# Contents

**About This Book**  
- xxi
- xxi
- xxi
- xxi
- xxii
- xxii

**Base Operating System (BOS) Runtime Services (A-P)**

<table>
<thead>
<tr>
<th>Function</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>a64l or l64a Subroutine</td>
<td>1</td>
</tr>
<tr>
<td>abort Subroutine</td>
<td>1</td>
</tr>
<tr>
<td>abs, div, labs, ldiv, imul_dbl, umul_dbl, llabs, or lldiv Subroutine</td>
<td>2</td>
</tr>
<tr>
<td>access, accessx, or faccessx Subroutine</td>
<td>3</td>
</tr>
<tr>
<td>acct Subroutine</td>
<td>4</td>
</tr>
<tr>
<td>acl_chg or acl_fchg Subroutine</td>
<td>7</td>
</tr>
<tr>
<td>acl_get or acl_fget Subroutine</td>
<td>8</td>
</tr>
<tr>
<td>acl_put or acl_fput Subroutine</td>
<td>10</td>
</tr>
<tr>
<td>acl_set or acl_fset Subroutine</td>
<td>12</td>
</tr>
<tr>
<td>acl_convert Subroutine</td>
<td>14</td>
</tr>
<tr>
<td>aclx_get or aclx_fget Subroutine</td>
<td>16</td>
</tr>
<tr>
<td>aclx_gettypeinfo Subroutine</td>
<td>17</td>
</tr>
<tr>
<td>aclx_gettypes Subroutine</td>
<td>20</td>
</tr>
<tr>
<td>aclx_print or aclx_printStr Subroutine</td>
<td>21</td>
</tr>
<tr>
<td>aclx_put or aclx_fput Subroutine</td>
<td>23</td>
</tr>
<tr>
<td>aclx_scan or aclx_scanStr Subroutine</td>
<td>25</td>
</tr>
<tr>
<td>acos, acosl Subroutine</td>
<td>27</td>
</tr>
<tr>
<td>acosh, acosl Subroutine</td>
<td>29</td>
</tr>
<tr>
<td>acosl, acosfl, or acosfl Subroutine</td>
<td>30</td>
</tr>
<tr>
<td>addproj Subroutine</td>
<td>31</td>
</tr>
<tr>
<td>addprojdb Subroutine</td>
<td>32</td>
</tr>
<tr>
<td>addssys Subroutine</td>
<td>33</td>
</tr>
<tr>
<td>adjtime Subroutine</td>
<td>35</td>
</tr>
<tr>
<td>agg_proc_stat, agg_lpar_stat, agg_arm_stat, or free_agg_list Subroutine</td>
<td>36</td>
</tr>
<tr>
<td>aio_cancel or aio_cancel64 Subroutine</td>
<td>38</td>
</tr>
<tr>
<td>aio_error or aio_error64 Subroutine</td>
<td>42</td>
</tr>
<tr>
<td>aio_fsync Subroutine</td>
<td>45</td>
</tr>
<tr>
<td>aio_nwait Subroutine</td>
<td>47</td>
</tr>
<tr>
<td>aio_nwait_timeout Subroutine</td>
<td>49</td>
</tr>
<tr>
<td>aio_read or aio_read64 Subroutine</td>
<td>50</td>
</tr>
<tr>
<td>aio_return or aio_return64 Subroutine</td>
<td>52</td>
</tr>
<tr>
<td>aio_suspend or aio_suspend64 Subroutine</td>
<td>56</td>
</tr>
<tr>
<td>aio_write or aio_write64 Subroutine</td>
<td>58</td>
</tr>
<tr>
<td>alloc, dealloc, print, read_data, read_regs, symbol_addr, write_data, and write_regs Subroutine</td>
<td>60</td>
</tr>
<tr>
<td>allocclmb Subroutine</td>
<td>62</td>
</tr>
<tr>
<td>arm_end Subroutine</td>
<td>66</td>
</tr>
<tr>
<td>arm_end Dual Call Subroutine</td>
<td>68</td>
</tr>
<tr>
<td>arm_getid Subroutine</td>
<td>69</td>
</tr>
<tr>
<td>arm_getid Dual Call Subroutine</td>
<td>70</td>
</tr>
<tr>
<td>arm_init Subroutine</td>
<td>71</td>
</tr>
<tr>
<td>arm_init Dual Call Subroutine</td>
<td>72</td>
</tr>
<tr>
<td>arm_start Subroutine</td>
<td>73</td>
</tr>
<tr>
<td>arm_start Dual Call Subroutine</td>
<td>74</td>
</tr>
<tr>
<td>arm_stop Subroutine</td>
<td>75</td>
</tr>
<tr>
<td>arm_stop Dual Call Subroutine</td>
<td>76</td>
</tr>
<tr>
<td>arm_update Subroutine</td>
<td>77</td>
</tr>
</tbody>
</table>

© Copyright IBM Corp. 1994, 2006
<table>
<thead>
<tr>
<th>Subroutine Name</th>
<th>Page Number</th>
</tr>
</thead>
<tbody>
<tr>
<td>arm_update Dual Call</td>
<td>89</td>
</tr>
<tr>
<td>asinh, asinhf, or asinhl</td>
<td>90</td>
</tr>
<tr>
<td>asinf, asinl, or asin</td>
<td>91</td>
</tr>
<tr>
<td>assert Macro</td>
<td>92</td>
</tr>
<tr>
<td>atan2f, atan2l, or atan2</td>
<td>93</td>
</tr>
<tr>
<td>atan, atanf, or atanfl</td>
<td>94</td>
</tr>
<tr>
<td>atanh, atanhf, or atanhl</td>
<td>95</td>
</tr>
<tr>
<td>atof, atoff</td>
<td>96</td>
</tr>
<tr>
<td>audit Subroutine</td>
<td>98</td>
</tr>
<tr>
<td>auditbin Subroutine</td>
<td>100</td>
</tr>
<tr>
<td>auditevents Subroutine</td>
<td>102</td>
</tr>
<tr>
<td>auditlog Subroutine</td>
<td>104</td>
</tr>
<tr>
<td>auditobj Subroutine</td>
<td>105</td>
</tr>
<tr>
<td>auditpack Subroutine</td>
<td>108</td>
</tr>
<tr>
<td>auditproc Subroutine</td>
<td>109</td>
</tr>
<tr>
<td>auditread, auditread_r</td>
<td>111</td>
</tr>
<tr>
<td>auditwrite Subroutine</td>
<td>112</td>
</tr>
<tr>
<td>authenticate Subroutine</td>
<td>113</td>
</tr>
<tr>
<td>authenticatex Subroutine</td>
<td>115</td>
</tr>
<tr>
<td>basename Subroutine</td>
<td>117</td>
</tr>
<tr>
<td>bcrypt, bcmp, bzero or ffs</td>
<td>118</td>
</tr>
<tr>
<td>bessel: j0, j1, jn, y0, y1, or yn</td>
<td>119</td>
</tr>
<tr>
<td>bindprocessor Subroutine</td>
<td>120</td>
</tr>
<tr>
<td>brk or sbrk</td>
<td>122</td>
</tr>
<tr>
<td>bsearch Subroutine</td>
<td>123</td>
</tr>
<tr>
<td>btowc</td>
<td>124</td>
</tr>
<tr>
<td>buildproclist Subroutine</td>
<td>125</td>
</tr>
<tr>
<td>buildtranlist or freetranslist</td>
<td>126</td>
</tr>
<tr>
<td>_check_lock Subroutine</td>
<td>127</td>
</tr>
<tr>
<td>_clear_lock Subroutine</td>
<td>128</td>
</tr>
<tr>
<td>cabs, cabsf, or cabsfl</td>
<td>129</td>
</tr>
<tr>
<td>cacos, cacosf, or cacosfl</td>
<td>129</td>
</tr>
<tr>
<td>cacosh, cacoshf, or cacoshl</td>
<td>130</td>
</tr>
<tr>
<td>carg, cargf, or cargfl</td>
<td>131</td>
</tr>
<tr>
<td>casin, casinf, or casinfl</td>
<td>131</td>
</tr>
<tr>
<td>casinh, casinfl, or casinhl</td>
<td>132</td>
</tr>
<tr>
<td>catan, catanf, or catanfl</td>
<td>132</td>
</tr>
<tr>
<td>catanh, catanhf, or catanhl</td>
<td>133</td>
</tr>
<tr>
<td>catclose Subroutine</td>
<td>134</td>
</tr>
<tr>
<td>catget Subroutine</td>
<td>135</td>
</tr>
<tr>
<td>catopen Subroutine</td>
<td>136</td>
</tr>
<tr>
<td>cbrr, cbrrl, or cbrrl</td>
<td>137</td>
</tr>
<tr>
<td>ccoss, ccosf, or ccosl</td>
<td>138</td>
</tr>
<tr>
<td>ccosh, ccoshf, or ccoshl</td>
<td>139</td>
</tr>
<tr>
<td>ccsdtocs or cstonecsid</td>
<td>139</td>
</tr>
<tr>
<td>ceil, ceilf, or ceilll</td>
<td>140</td>
</tr>
<tr>
<td>cexp, cexpf, or cexpl</td>
<td>141</td>
</tr>
<tr>
<td>cfgetospeed, cfgetospeed, cfgetispeed, or cfsetispeed Subroutine</td>
<td>142</td>
</tr>
<tr>
<td>chacl or fchacl Subroutine</td>
<td>144</td>
</tr>
<tr>
<td>chdir Subroutine</td>
<td>147</td>
</tr>
<tr>
<td>chmod or fchmod Subroutine</td>
<td>148</td>
</tr>
<tr>
<td>chown, fchown, lchown, chownx, or fchownx Subroutine</td>
<td>151</td>
</tr>
<tr>
<td>chpass Subroutine</td>
<td>154</td>
</tr>
<tr>
<td>chpassx Subroutine</td>
<td>156</td>
</tr>
<tr>
<td>chprojattr Subroutine</td>
<td>158</td>
</tr>
<tr>
<td>Subroutine</td>
<td>Page</td>
</tr>
<tr>
<td>----------------------------------------------------------------------------</td>
<td>------</td>
</tr>
<tr>
<td>getpaginfo Subroutine</td>
<td>395</td>
</tr>
<tr>
<td>getpagvalue or getpagvalue64 Subroutine</td>
<td>396</td>
</tr>
<tr>
<td>getpass Subroutine</td>
<td>397</td>
</tr>
<tr>
<td>getcred Subroutine</td>
<td>398</td>
</tr>
<tr>
<td>getpeereid Subroutine</td>
<td>400</td>
</tr>
<tr>
<td>getpenv Subroutine</td>
<td>400</td>
</tr>
<tr>
<td>getpgid Subroutine</td>
<td>402</td>
</tr>
<tr>
<td>getpid, getpgrp, or getppid Subroutine</td>
<td>402</td>
</tr>
<tr>
<td>getportattr or putportattr Subroutine</td>
<td>403</td>
</tr>
<tr>
<td>getpri Subroutine</td>
<td>406</td>
</tr>
<tr>
<td>getpriority, setpriority, or nice Subroutine</td>
<td>407</td>
</tr>
<tr>
<td>getproclist, getlparlist, or getarmlist Subroutine</td>
<td>409</td>
</tr>
<tr>
<td>getprocs Subroutine</td>
<td>410</td>
</tr>
<tr>
<td>getproj Subroutine</td>
<td>413</td>
</tr>
<tr>
<td>getprojdb Subroutine</td>
<td>414</td>
</tr>
<tr>
<td>getprojs Subroutine</td>
<td>415</td>
</tr>
<tr>
<td>getpw Subroutine</td>
<td>416</td>
</tr>
<tr>
<td>getwent, getpwuid, getpwnam, putwent, setwent, or endwent Subroutine</td>
<td>417</td>
</tr>
<tr>
<td>getrlimit, getrlimit64, setrlimit, setrlimit64, or vlimit Subroutine</td>
<td>419</td>
</tr>
<tr>
<td>getrpcent, getrpcbyname, getrpcbyname, setrpcent, or endrpcent Subroutine</td>
<td>422</td>
</tr>
<tr>
<td>getrusage, getrusage64, times, or vtimes Subroutine</td>
<td>423</td>
</tr>
<tr>
<td>getroleattr or putroleattr Subroutine</td>
<td>426</td>
</tr>
<tr>
<td>gets or fgets Subroutine</td>
<td>429</td>
</tr>
<tr>
<td>getfsent_r, getfsspec_r, getfsfile_r, getfstype_r, setfsent_r, or endfsent_r Subroutine</td>
<td>430</td>
</tr>
<tr>
<td>getsid Subroutine</td>
<td>431</td>
</tr>
<tr>
<td>getssys Subroutine</td>
<td>432</td>
</tr>
<tr>
<td>getsobut Subroutine</td>
<td>433</td>
</tr>
<tr>
<td>getsubsvr Subroutine</td>
<td>434</td>
</tr>
<tr>
<td>gettcattr or puttcattr Subroutine</td>
<td>435</td>
</tr>
<tr>
<td>getthrds Subroutine</td>
<td>438</td>
</tr>
<tr>
<td>gettimeofday, settimeofday, or ftime Subroutine</td>
<td>440</td>
</tr>
<tr>
<td>gettimer, settimer, restimer, stime, or time Subroutine</td>
<td>441</td>
</tr>
<tr>
<td>gettimerid Subroutine</td>
<td>444</td>
</tr>
<tr>
<td>gettyent, gettynam, settyent, or endtyent Subroutine</td>
<td>445</td>
</tr>
<tr>
<td>getuid, geteuid, or getuidx Subroutine</td>
<td>447</td>
</tr>
<tr>
<td>getuserinfo Subroutine</td>
<td>448</td>
</tr>
<tr>
<td>getuserinfox Subroutine</td>
<td>448</td>
</tr>
<tr>
<td>getUserAuths Subroutine</td>
<td>449</td>
</tr>
<tr>
<td>getUserattr, IDtouser, nextuser, or putuserattr Subroutine</td>
<td>455</td>
</tr>
<tr>
<td>getUserattrs Subroutine</td>
<td>462</td>
</tr>
<tr>
<td>getuserpw, putuserpw, or putuserwhist Subroutine</td>
<td>463</td>
</tr>
<tr>
<td>getuserpwx Subroutine</td>
<td>465</td>
</tr>
<tr>
<td>getusraclattr, nextusracl or putusraclattr Subroutine</td>
<td>467</td>
</tr>
<tr>
<td>getutent, getutid, getutline, pututline, setutent, endutent, or utmpname Subroutine</td>
<td>469</td>
</tr>
<tr>
<td>getvfsent, getvfsbytype, getvfsbyname, getvfsbyflag, setvfsent, or endvfsent Subroutine</td>
<td>471</td>
</tr>
<tr>
<td>getwd Subroutine</td>
<td>472</td>
</tr>
<tr>
<td>getws or fgetws Subroutine</td>
<td>474</td>
</tr>
<tr>
<td>glob Subroutine</td>
<td>475</td>
</tr>
<tr>
<td>globfree Subroutine</td>
<td>477</td>
</tr>
<tr>
<td>grantpt Subroutine</td>
<td>480</td>
</tr>
<tr>
<td>HBA_CloseAdapter Subroutine</td>
<td>481</td>
</tr>
<tr>
<td>HBA_FreeLibrary Subroutine</td>
<td>482</td>
</tr>
<tr>
<td>Function Name</td>
<td>Page</td>
</tr>
<tr>
<td>---------------------------------------------------</td>
<td>------</td>
</tr>
<tr>
<td>HBA_GetEventBuffer Subroutine</td>
<td>486</td>
</tr>
<tr>
<td>HBA_GetFC4Statistics Subroutine</td>
<td>487</td>
</tr>
<tr>
<td>HBA_GetFcpPersistentBinding Subroutine</td>
<td>488</td>
</tr>
<tr>
<td>HBA_GetFCPStatistics Subroutine</td>
<td>489</td>
</tr>
<tr>
<td>HBA_GetFcpTargetMappingV2 Subroutine</td>
<td>490</td>
</tr>
<tr>
<td>HBA_GetFcpTargetMapping Subroutine</td>
<td>491</td>
</tr>
<tr>
<td>HBA_GetNumberOfAdapters Subroutine</td>
<td>492</td>
</tr>
<tr>
<td>HBA_GetPersistentBindingV2 Subroutine</td>
<td>493</td>
</tr>
<tr>
<td>HBA_GetPortStatistics Subroutine</td>
<td>494</td>
</tr>
<tr>
<td>HBA_GetRNIDMgmtInfo Subroutine</td>
<td>495</td>
</tr>
<tr>
<td>HBA_GetVersion Subroutine</td>
<td>496</td>
</tr>
<tr>
<td>HBA_LoadLibrary Subroutine</td>
<td>497</td>
</tr>
<tr>
<td>HBA_OpenAdapter Subroutine</td>
<td>498</td>
</tr>
<tr>
<td>HBA_OpenAdapterByWWN Subroutine</td>
<td>498</td>
</tr>
<tr>
<td>HBA_RefreshInformation Subroutine</td>
<td>499</td>
</tr>
<tr>
<td>HBA_ScsiInquiryV2 Subroutine</td>
<td>500</td>
</tr>
<tr>
<td>HBA_ScsiReadCapacityV2 Subroutine</td>
<td>501</td>
</tr>
<tr>
<td>HBA_ScsiReportLunsV2 Subroutine</td>
<td>504</td>
</tr>
<tr>
<td>HBA_SendCTPassThru Subroutine</td>
<td>505</td>
</tr>
<tr>
<td>HBA_SendCTPassThruV2 Subroutine</td>
<td>506</td>
</tr>
<tr>
<td>HBA_SendReadCapacity Subroutine</td>
<td>507</td>
</tr>
<tr>
<td>HBA_SendReportLUNS Subroutine</td>
<td>508</td>
</tr>
<tr>
<td>HBA_SendRLS Subroutine</td>
<td>509</td>
</tr>
<tr>
<td>HBA_SendRNID Subroutine</td>
<td>510</td>
</tr>
<tr>
<td>HBA_SendRNIDV2 Subroutine</td>
<td>511</td>
</tr>
<tr>
<td>HBA_SendRPL Subroutine</td>
<td>512</td>
</tr>
<tr>
<td>HBA_SendRPS Subroutine</td>
<td>514</td>
</tr>
<tr>
<td>HBA_SendScsilquiry Subroutine</td>
<td>515</td>
</tr>
<tr>
<td>HBA_SetRNIDMgmtInfo Subroutine</td>
<td>516</td>
</tr>
<tr>
<td>hpmInit, f_hpminit, hpmStart, f_hpmstart, hpmStop, f_hpmstop, hpmTstart, f_hpmtstart, hpmTstop, f_hpmsttop, f_hpmgettimeandcounters, hpmGetCounters, f_hpmgetcounters, hpmTerminate, and f_hpmterminate Subroutine</td>
<td>519</td>
</tr>
<tr>
<td>hsearch, hcreate, or hdestroy Subroutine</td>
<td>522</td>
</tr>
<tr>
<td>hypot, hypotf, or hypotl Subroutine</td>
<td>523</td>
</tr>
<tr>
<td>iconv Subroutine</td>
<td>524</td>
</tr>
<tr>
<td>iconv_close Subroutine</td>
<td>526</td>
</tr>
<tr>
<td>iconv_open Subroutine</td>
<td>527</td>
</tr>
<tr>
<td>ilogbf, ilogbl, or ilogb Subroutine</td>
<td>529</td>
</tr>
<tr>
<td>imaxabs Subroutine</td>
<td>530</td>
</tr>
<tr>
<td>imaxdiv Subroutine</td>
<td>530</td>
</tr>
<tr>
<td>IMAXMapping Subroutine</td>
<td>531</td>
</tr>
<tr>
<td>IMAuxCreate Callback Subroutine</td>
<td>532</td>
</tr>
<tr>
<td>IMAuxDestroy Callback Subroutine</td>
<td>532</td>
</tr>
<tr>
<td>IMAuxDraw Callback Subroutine</td>
<td>533</td>
</tr>
<tr>
<td>IMAuxHide Callback Subroutine</td>
<td>534</td>
</tr>
<tr>
<td>IMBeep Callback Subroutine</td>
<td>534</td>
</tr>
<tr>
<td>IMClose Subroutine</td>
<td>535</td>
</tr>
<tr>
<td>IMCreate Subroutine</td>
<td>536</td>
</tr>
<tr>
<td>IMDestroy Subroutine</td>
<td>536</td>
</tr>
<tr>
<td>IMFIlter Subroutine</td>
<td>537</td>
</tr>
<tr>
<td>IMFreeKeymap Subroutine</td>
<td>538</td>
</tr>
<tr>
<td>IMIndicatorDraw Callback Subroutine</td>
<td>538</td>
</tr>
<tr>
<td>IMIndicatorHide Callback Subroutine</td>
<td>539</td>
</tr>
<tr>
<td>IMInitialize Subroutine</td>
<td>540</td>
</tr>
<tr>
<td>IMInitializeKeymap Subroutine</td>
<td>541</td>
</tr>
<tr>
<td>IMLocall Subroutine</td>
<td>542</td>
</tr>
</tbody>
</table>
Technical Reference, Volume 1: Base Operating System and Extensions
LAPI_Probe Subroutine .................................................. 624
LAPI_Purge_totask Subroutine ............................................ 625
LAPI_Put Subroutine .......................................................... 627
LAPI_Putv Subroutine .......................................................... 629
LAPI_Qenv Subroutine .......................................................... 633
LAPI.Resume_totask Subroutine .............................................. 636
LAPI_RMw Subroutine ........................................................... 637
LAPI_RMw64 Subroutine ........................................................ 641
LAPI_Senv Subroutine .......................................................... 645
LAPI_Setcntr Subroutine ...................................................... 647
LAPI.Setcntr_wstatus Subroutine .......................................... 649
LAPI_Term Subroutine .......................................................... 650
LAPI_Util Subroutine ........................................................... 652
LAPI_Waitcntr Subroutine ..................................................... 653
LAPI_Xfer Subroutine ........................................................... 655
layout_object_create Subroutine ............................................ 678
layout_object_editshape or wclayout_object_editshape Subroutine 679
layout_object_getvalue Subroutine .......................................... 682
layout_object_setvalue Subroutine .......................................... 684
layout_object_shapeboxchars Subroutine ................................ 686
layout_object_transform or wclayout_object_transform Subroutine 687
layout_object_free Subroutine ................................................. 690
ldahread Subroutine ............................................................ 691
ldclose or ldaclose Subroutine ............................................. 692
ldexp, ldexpf, or ldexpl Subroutine ....................................... 693
ldfhread Subroutine ............................................................ 694
ldgetname Subroutine ........................................................... 696
ldlread, ldlinit, or ldlitem Subroutine ................................... 698
ldlseek or ldnlseek Subroutine ............................................. 700
ldohseek Subroutine ............................................................ 701
ldopen or ldaopen Subroutine ................................................ 703
lstrstr or ldnrseek Subroutine ............................................... 704
ldshread or ldnshread Subroutine ........................................... 706
ldtbindex Subroutine .......................................................... 707
ldtbread Subroutine ............................................................ 708
ldtsseek or ldnsseek Subroutine ............................................ 709
lgamma, lgammaf, or lgammal Subroutine ................................ 710
lineout Subroutine ............................................................. 711
link Subroutine ................................................................. 712
lio_listio or lio_listio64 Subroutine ..................................... 713
liste Subroutine ................................................................. 718
llrint, llrintf, or llrintl Subroutine ....................................... 719
llround, llroundf, or llroundl Subroutine ................................ 720
load Subroutine ................................................................ 721
loadbind Subroutine ............................................................. 725
loadquery Subroutine ............................................................ 726
localeconv Subroutine ........................................................... 728
lockx, lockf, flock, or lockf64 Subroutine ................................ 733
log10, log10f, or log10l Subroutine ....................................... 736
log1p, log1pf, or log1pl Subroutine ....................................... 737
log2, log2f, or log2l Subroutine ............................................ 738
logbf, logbl, or logb Subroutine ............................................ 739
log, logf, or logl Subroutine .................................................. 740
loginfailed Subroutine .......................................................... 741
loginrestrictions Subroutine .................................................. 743
loginrestrictionsx Subroutine .................................................. 746
loginsuccess Subroutine ......................................................... 748
lpar_get_info Subroutine ....................................................... 750
lpar_set_resources Subroutine ............................................... 751
lrint, lrintf, or lrintl Subroutine ......................................... 752
lround, lroundf, or lroundl Subroutine .................................... 753
lsearch or lfind Subroutine .................................................. 754
lseek, llseek, or lseek64 Subroutine ........................................ 756
lvm_querylv Subroutine ........................................................ 757
lvm_querypv Subroutine ........................................................ 761
lvm_queryvg Subroutine ........................................................ 765
lvm_queryvgs Subroutine ....................................................... 768
malloc, free, realloc, calloc, mallopt, mallinfo, mallinfo_heap, alloca, valloc, or posix_memalign Subroutine .......................................................... 769
madd, msub, mult, mdiv, pow, gcd, invert, rpow, msqrt, mcmp, move, min, omin, fmin, m_in, mout, omout, fmout, m_out, sdv, or itom Subroutine ................................................. 776
madvise Subroutine ................................................................ 777
makecontext or swapcontext Subroutine .................................... 779
matherr Subroutine ................................................................ 780
MatchAllAuths, MatchAnyAuths, MatchAllAuthsList, or MatchAnyAuthsList Subroutine .................................................. 781
mblen Subroutine .................................................................... 782
mbrlen Subroutine .................................................................. 783
mbrtocw Subroutine ................................................................ 784
mbsadvance Subroutine ......................................................... 785
mbcat, mbscmp, or mbscpy Subroutine ...................................... 786
mbschr Subroutine .................................................................. 787
mbsinit Subroutine .................................................................. 788
mbsinvalid Subroutine ........................................................... 789
mbslen Subroutine .................................................................. 789
mbsncat, mbsncmp, or mbsncpy Subroutine ................................ 790
mbspbrk Subroutine ................................................................ 791
mbschr Subroutine .................................................................. 792
mbstowcs Subroutine ............................................................. 793
mbstomb Subroutine ............................................................... 794
mbstowcs Subroutine ............................................................. 795
mbswidth Subroutine ................................................................ 796
mbtocw Subroutine .................................................................. 796
memccpy, memchr, memcmp, memcpy, memset or memmove Subroutine .......................................................... 798
mincore Subroutine .................................................................. 799
MIO_aio_read64 Subroutine ...................................................... 800
MIO_aio_suspend64 Subroutine ............................................... 801
MIO_aio_write64 Subroutine .................................................... 802
MIO_close Subroutine ............................................................. 803
MIO_fcntl Subroutine ............................................................. 807
MIO_finfo Subroutine ............................................................. 808
MIO_fstat64 Subroutine ........................................................... 809
MIO_fsync Subroutine ............................................................. 810
MIO_ftruncate64 Subroutine ..................................................... 810
MIO_ftruncate64 Subroutine ..................................................... 811
MIO_iio_listio64 Subroutine .................................................... 811
MIO_lseek64 Subroutine ........................................................... 812
MIO_open64 Subroutine ........................................................... 813
MIO_open Subroutine .............................................................. 819
MIO_read Subroutine ............................................................... 824
MIO_write Subroutine ............................................................. 825
mkdir Subroutine ....................................................................... 826
mknod or mkfifo Subroutine ................................................... 828
mkt temp or mkstemp Subroutine

mlock and munlock Subroutine

mlockall and munlockall Subroutine

mmap or mmap64 Subroutine

mntctl Subroutine

mod, modf, or modfl Subroutine

moncontrol Subroutine

monitor Subroutine

monstartup Subroutine

mprotect Subroutine

mq_close Subroutine

mq_getattr Subroutine

mq_notify Subroutine

mq_open Subroutine

mq_receive Subroutine

mq_send Subroutine

mq_setattr Subroutine

mq_unlink Subroutine

msem_init Subroutine

msem_lock Subroutine

msem_remove Subroutine

msem_unlock Subroutine

msgctl Subroutine

msgget Subroutine

msgrcv Subroutine

msgsnd Subroutine

msgxrcv Subroutine

msleep Subroutine

msync Subroutine

mt__trce Subroutine

nftw or nftw64 Subroutine

nl_langinfo Subroutine

nlist, nlist64 Subroutine

ns_addr Subroutine

ns_ntoa Subroutine

odm_add_obj Subroutine

odm_change_obj Subroutine

odm_close_class Subroutine

odm_create_class Subroutine

odm_err_msg Subroutine

odm_free_list Subroutine

odm_get_by_id Subroutine

odm_get_list Subroutine

odm_get_obj, odm_get_first, or odm_get_next Subroutine

odm_initialize Subroutine

odm_lock Subroutine
<table>
<thead>
<tr>
<th>Subroutine</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>odm_mount_class Subroutine</td>
<td>915</td>
</tr>
<tr>
<td>odm_open_class or odm_open_class_rdonly Subroutine</td>
<td>916</td>
</tr>
<tr>
<td>odm_rm_by_id Subroutine</td>
<td>917</td>
</tr>
<tr>
<td>odm_rm_class Subroutine</td>
<td>918</td>
</tr>
<tr>
<td>odm_rm_obj Subroutine</td>
<td>919</td>
</tr>
<tr>
<td>odm_run_method Subroutine</td>
<td>921</td>
</tr>
<tr>
<td>odm_set_path Subroutine</td>
<td>922</td>
</tr>
<tr>
<td>odm_set_perms Subroutine</td>
<td>923</td>
</tr>
<tr>
<td>odm_terminate Subroutine</td>
<td>923</td>
</tr>
<tr>
<td>odm_unlock Subroutine</td>
<td>924</td>
</tr>
<tr>
<td>open, openx, open64, creat, or creat64 Subroutine</td>
<td>925</td>
</tr>
<tr>
<td>opendir, readdir, telldir, seekdir, rewinddir, closedir, opendir64, readdir64, telldir64, seekdir64, rewinddir64, or closedir64 Subroutine</td>
<td>933</td>
</tr>
<tr>
<td>pam_acct_mgmt Subroutine</td>
<td>936</td>
</tr>
<tr>
<td>pam_authenticate Subroutine</td>
<td>937</td>
</tr>
<tr>
<td>pam_chauthtok Subroutine</td>
<td>939</td>
</tr>
<tr>
<td>pam_close_session Subroutine</td>
<td>940</td>
</tr>
<tr>
<td>pam_end Subroutine</td>
<td>941</td>
</tr>
<tr>
<td>pam_get_data Subroutine</td>
<td>942</td>
</tr>
<tr>
<td>pam_get_item Subroutine</td>
<td>943</td>
</tr>
<tr>
<td>pam_get_user Subroutine</td>
<td>944</td>
</tr>
<tr>
<td>pam_getenv Subroutine</td>
<td>945</td>
</tr>
<tr>
<td>pam_getenvlist Subroutine</td>
<td>946</td>
</tr>
<tr>
<td>pam_open_session Subroutine</td>
<td>947</td>
</tr>
<tr>
<td>pam_putenv Subroutine</td>
<td>948</td>
</tr>
<tr>
<td>pam_set_data Subroutine</td>
<td>949</td>
</tr>
<tr>
<td>pam_set_item Subroutine</td>
<td>950</td>
</tr>
<tr>
<td>pam_setcred Subroutine</td>
<td>951</td>
</tr>
<tr>
<td>pam_sm_acct_mgmt Subroutine</td>
<td>953</td>
</tr>
<tr>
<td>pam_sm_authenticate Subroutine</td>
<td>954</td>
</tr>
<tr>
<td>pam_sm_chauthtok Subroutine</td>
<td>955</td>
</tr>
<tr>
<td>pam_sm_close_session Subroutine</td>
<td>957</td>
</tr>
<tr>
<td>pam_sm_open_session Subroutine</td>
<td>958</td>
</tr>
<tr>
<td>pam_sm_setcred Subroutine</td>
<td>959</td>
</tr>
<tr>
<td>pam_strerror Subroutine</td>
<td>960</td>
</tr>
<tr>
<td>passwdexpired Subroutine</td>
<td>963</td>
</tr>
<tr>
<td>passwdexpiredx Subroutine</td>
<td>963</td>
</tr>
<tr>
<td>passwdpolicy Subroutine</td>
<td>964</td>
</tr>
<tr>
<td>passwdstrength Subroutine</td>
<td>966</td>
</tr>
<tr>
<td>pathconf or fpathconf Subroutine</td>
<td>968</td>
</tr>
<tr>
<td>pause Subroutine</td>
<td>972</td>
</tr>
<tr>
<td>pcap_close Subroutine</td>
<td>972</td>
</tr>
<tr>
<td>pcap_compile Subroutine</td>
<td>973</td>
</tr>
<tr>
<td>pcap_datalink Subroutine</td>
<td>974</td>
</tr>
<tr>
<td>pcap_dispatch Subroutine</td>
<td>974</td>
</tr>
<tr>
<td>pcap_dump Subroutine</td>
<td>974</td>
</tr>
<tr>
<td>pcap_dump_close Subroutine</td>
<td>976</td>
</tr>
<tr>
<td>pcap_dump_open Subroutine</td>
<td>976</td>
</tr>
<tr>
<td>pcap_fileno Subroutine</td>
<td>977</td>
</tr>
<tr>
<td>pcap_fileno Subroutine</td>
<td>978</td>
</tr>
<tr>
<td>pcap_geterr Subroutine</td>
<td>978</td>
</tr>
<tr>
<td>pcap_is_swapped Subroutine</td>
<td>979</td>
</tr>
<tr>
<td>pcap_lookupdev Subroutine</td>
<td>980</td>
</tr>
<tr>
<td>pcap_lookupnet Subroutine</td>
<td>980</td>
</tr>
<tr>
<td>pcap_loop Subroutine</td>
<td>981</td>
</tr>
</tbody>
</table>
pcap_major_version Subroutine .................................................. 983
pcap_minor_version Subroutine ................................................. 984
pcap_next Subroutine ............................................................. 985
pcap_open_live Subroutine ....................................................... 985
pcap_open_offline Subroutine ................................................ 986
pcap_error Subroutine ............................................................ 987
pcap_setfilter Subroutine ....................................................... 988
pcap_snapshot Subroutine ........................................................ 989
pcap_stats Subroutine ............................................................ 989
pcap_strerror Subroutine ........................................................ 990
pcap_strerror Subroutine ........................................................ 991
pclose Subroutine ..................................................................... 992
perfstat_cpu Subroutine ............................................................ 992
perfstat_cpu_total Subroutine .................................................. 994
perfstat_disk Subroutine .......................................................... 995
perfstat_diskadapter Subroutine ................................................. 997
perfstat_diskpath Subroutine .................................................... 998
perfstat_memory_total Subroutine .............................................. 1000
perfstat_netbuffer Subroutine .................................................. 1001
perfstat_netinterface Subroutine .............................................. 1003
perfstat_netinterface_total Subroutine ..................................... 1004
perfstat_netinterface Subroutine .............................................. 1006
perfstat_pagingspace Subroutine ............................................... 1007
perfstat_partial_reset Subroutine ............................................. 1008
perfstat_partition_total Subroutine .......................................... 1010
perfstat_protocol Subroutine .................................................... 1010
perfstat_reset Subroutine ....................................................... 1012
pm_cycles Subroutine ............................................................. 1017
pm_delete_program Subroutine ............................................... 1018
pm_delete_program_group Subroutine ....................................... 1019
pm_delete_program_mygroup Subroutine .................................... 1019
pm_delete_program_mythread Subroutine ................................... 1020
pm_delete_program_pgroup Subroutine ...................................... 1021
pm_delete_program_pthread Subroutine ..................................... 1022
pm_delete_program_thread Subroutine ....................................... 1023
pm_error Subroutine .............................................................. 1024
pm_get_data, pm_get_tdata, pm_get_TData, pm_get_data_cpu, pm_get_tdata_cpu,
            pm_get_TData_cpu, pm_get_data_lcpu, pm_get_tdata_lcpu and pm_get_Tdata_lcpu Subroutine ..... 1025
pm_get_data_group, pm_get_tdata_group and pm_get_Tdata_group Subroutine .................................................. 1028
pm_get_data_group_mx and pm_get_tdata_group_mx Subroutine ......... 1030
pm_get_data_mx, pm_get_tdata_mx, pm_get_data_cpu_mx, pm_get_tdata_cpu_mx,
            pm_get_data_lcpu_mx and pm_get_tdata_lcpu_mx Subroutine ......................................................... 1031
pm_get_data_mygroup, pm_get_tdata_mygroup or pm_get_Tdata_mygroup Subroutine .............................................. 1032
pm_get_data_mygroup_mx or pm_get_tdata_mygroup_mx Subroutine .... 1035
pm_get_data_mythread, pm_get_tdata_mythread or pm_get_Tdata_mythread Subroutine .............................................. 1036
pm_get_data_mythread_mx or pm_get_tdata_mythread_mx Subroutine .... 1037
pm_get_data_pgroup, pm_get_tdata_pgroup and pm_get_Tdata_pgroup Subroutine ...................................................... 1039
pm_get_data_pgroup_mx or pm_get_tdata_pgroup_mx Subroutine ........ 1039
pm_get_data_pthread, pm_get_tdata_pthread or pm_get_Tdata_pthread Subroutine .................................................... 1041
pm_get_data_pthread_mx or pm_get_tdata_pthread_mx Subroutine ....... 1041
pm_get_data_thread, pm_get_tdata_thread or pm_get_Tdata_thread Subroutine ...................................................... 1043
pm_get_data_thread_mx or pm_get_tdata_thread_mx Subroutine .......... 1045
pm_get_program Subroutine ........................................................ 1047
pm_get_program Group Subroutine .......................................... 1049

pm_get_program_group_mx Subroutine .......................................................... 1052
pm_get_program_mx Subroutine ...................................................................... 1053
pm_get_program_mygroup Subroutine ............................................................ 1055
pm_get_program_mygroup_mx Subroutine ......................................................... 1056
pm_get_program_mythread Subroutine ............................................................ 1057
pm_get_program_mythread_mx Subroutine ....................................................... 1058
pm_get_program_pgroup Subroutine ............................................................... 1060
pm_get_program_pgroup_mx Subroutine ......................................................... 1061
pm_get_program pthread Subroutine ............................................................. 1063
pm_get_program_pthread_mx Subroutine ......................................................... 1064
pm_get_program_thread Subroutine ............................................................... 1066
pm_get_program_thread_mx Subroutine .......................................................... 1067
pm_init Subroutine ......................................................................................... 1069
pm_initialize Subroutine ................................................................................ 1071
pm_reset_data Subroutine ................................................................................ 1073
pm_reset_data_group Subroutine ..................................................................... 1073
pm_reset_data_mygroup Subroutine ................................................................. 1075
pm_reset_data_mythread Subroutine ............................................................... 1076
pm_reset_data_pgroup Subroutine ................................................................. 1076
pm_reset_data_pthread Subroutine ................................................................. 1078
pm_reset_data_thread Subroutine ................................................................... 1079
pm_set_program Subroutine ............................................................................ 1080
pm_set_program_group Subroutine .................................................................. 1081
pm_set_program_group_mx Subroutine ............................................................ 1083
pm_set_program_mx Subroutine ....................................................................... 1085
pm_set_program_mygroup Subroutine ............................................................. 1086
pm_set_program_mygroup_mx Subroutine ....................................................... 1087
pm_set_program_mythread Subroutine ............................................................ 1089
pm_set_program_mythread_mx Subroutine ....................................................... 1090
pm_set_program_pgroup Subroutine ............................................................... 1092
pm_set_program_pgroup_mx Subroutine .......................................................... 1093
pm_set_program_pthread Subroutine ............................................................. 1095
pm_set_program_pthread_mx Subroutine ....................................................... 1097
pm_set_program_thread Subroutine .................................................................. 1098
pm_set_program_thread_mx Subroutine .......................................................... 1099
pm_start and pm_tstart Subroutine .................................................................. 1101
pm_start_group and pm_tstart_group Subroutine ............................................ 1102
pm_start_mygroup and pm_tstart_mygroup Subroutine ..................................... 1104
pm_start_mythread and pm_tstart_mythread Subroutine ................................... 1105
pm_start_pgroup and pm_tstart_pgroup Subroutine ........................................ 1106
pm_start pthread and pm_tstart_pthread Subroutine ..................................... 1107
pm_start_thread and pm_tstart_thread Subroutine .......................................... 1109
pm_stop and pm_tstop Subroutine ................................................................... 1110
pm_stop_group and pm_tstop_group Subroutine ............................................. 1111
pm_stop_mygroup and pm_tstop_mygroup Subroutine ...................................... 1112
pm_stop_mythread and pm_tstop_mythread Subroutine ................................... 1113
pm_stop_pgroup and pm_tstop_pgroup Subroutine .......................................... 1114
pm_stoppthread and pm_tstop_pthread Subroutine ......................................... 1116
pm_stop_thread and pm_tstop_thread Subroutine ........................................... 1117
poll Subroutine .............................................................................................. 1118
pollset_create, pollset_ctl, pollset_destroy, pollset_poll, and pollset_query Subroutines .......................................................... 1121
popen Subroutine ........................................................................................... 1124
posix_fadvise Subroutine ................................................................................ 1125
posix_fallocate Subroutine .............................................................................. 1126
posix_madvise Subroutine ............................................................................... 1127
posix_openpt Subroutine ................................................................................. 1128
posix_spawn or posix_spawnp Subroutine .................................................. 1129
posix_spawn_file_actions_addclose or posix_spawn_file_actions_addopen Subroutine .................................................. 1133
posix_spawn_file_actions_adddup2 Subroutine ........................................... 1134
posix_spawn_file_actions_destroy or posix_spawn_file_actions_init Subroutine .................................................. 1135
posix_spawnattr_destroy or posix_spawnattr_init Subroutine .................................................. 1136
posix_spawnattr_getflags or posix_spawnattr_setflags Subroutine .................................................. 1137
posix_spawnattr_getpgroup or posixSpawnattr_setpgroup Subroutine .................................................. 1138
posix_spawnattr_getschedparam or posix_spawnattr_setschedparam Subroutine .................................................. 1139
posix_spawnattr_getschedpolicy or posix_spawnattr_setschedpolicy Subroutine .................................................. 1140
posix_spawnattr_getsigdefault or posix_spawnattr_setsigdefault Subroutine .................................................. 1141
posix_spawnattr_getsigmask or posix_spawnattr_setsigmask Subroutine .................................................. 1142
posix_trace_getnext_event, posix_trace_timedgetnext_event, posix_trace_trygetnext_event Subroutine .................................................. 1143
powf, powl, or pow Subroutine .................................................. 1146
printf, fprintf, sprintf, snprintf, wprintf, vprintf, fprintf, vfprintf, vsprintf, or vwsprintf Subroutine .................................................. 1148
prof Subroutine .................................................. 1155
proj_execve Subroutine .................................................. 1157
projdballoc Subroutine .................................................. 1158
projdbfinit Subroutine .................................................. 1159
projdbfree Subroutine .................................................. 1160
psdanger Subroutine .................................................. 1161
psignal Subroutine or sys_siglist Vector .................................................. 1162
pthdb_attr, pthdb_cond, pthdb_condattr, pthdb_key, pthdb_mutex, pthdb_mutexattr, pthdb_pthread, pthdb_pthread_key, pthdb_rwlock, or pthdb_rwlockattr Subroutine .................................................. 1163
pthdb_attr_detachstate, pthdb_attr_addr, pthdb_attr_guardsize, pthdb_attr_inheritsched, pthdb_attr_schedparam, pthdb_attr_schedpolicy, pthdb_attr_scheduledpriority, pthdb_attr_scope, pthdb_attr_stackaddr, pthdb_attr_stacksize, or pthdb_attr_suspendstate Subroutine .................................................. 1165
pthdb_condattr_pshared, or pthdb_condattr_addr Subroutine .................................................. 1167
pthdb_cond_addr, pthdb_cond_mutex or pthdb_cond_pshared Subroutine .................................................. 1168
pthdb_mutexattr_addr, pthdb_mutexattr_prioceiling, pthdb_mutexattr_protocol, pthdb_mutexattr_pshared or pthdb_mutexattr_type Subroutine .................................................. 1169
pthdb_mutex_addr, pthdb_mutex_lock_count, pthdb_mutex_owner, pthdb_mutex_pshared, pthdb_mutex_prioceiling, pthdb_mutex_protcol, pthdb_mutex_state or pthdb_mutex_type Subroutine .................................................. 1171
pthdb_mutex_waiter, pthdb_cond_waiter, pthdb_rwlock_read_waiter or pthdb_rwlock_write_waiter Subroutine .................................................. 1172
pthdb_pthread_arg Subroutine .................................................. 1174
pthdb_pthread_context or pthdb_pthread_setcontext Subroutine .................................................. 1177
pthdb_pthread_hold, pthdb_pthread_holdstate or pthdb_pthread_unhold Subroutine .................................................. 1178
pthdb_pthread_sigmask, pthdb_pthread_sigsuspend or pthdb_pthread_sigwait Subroutine .................................................. 1179
pthdb_pthread_specific Subroutine .................................................. 1180
pthdb_pthread_tid or pthdb_tid_pthread Subroutine .................................................. 1181
pthdb_rwlockattr_addr, or pthdb_rwlockattr_pshared Subroutine .................................................. 1182
pthdb_rwlock_addr, pthdb_rwlock_lock_count, pthdb_rwlock_owner, pthdb_rwlock_pshared or pthdb_rwlock_state Subroutine .................................................. 1183
pthdb_session_committed Subroutines .................................................. 1185
pthread_atfork Subroutine .................................................. 1188
pthread_attr_destroy Subroutine .................................................. 1189
pthread_attr_getguardsize or pthread_attr_setguardsize Subroutines .................................................. 1190
pthread_attr_getinheritsched, pthread_attr_setinheritsched Subroutine .................................................. 1192
pthread_attr_getschedparam Subroutine .................................................. 1193
pthread_attr_getschedpolicy, pthread_attr_setschedpolicy Subroutine .................................................. 1194
pthread_attr_getstackaddr Subroutine .................................................. 1195
pthread_attr_getstacksize Subroutine .................................................. 1196
pthread_attr_init Subroutine .................................................. 1197
pthread_attr_getdetachstate or pthread_attr_setdetachstate Subroutines .................................................. 1198

Contents xvii
pthread_attr_getscope and pthread_attr_setscope Subroutines

pthread_attr_setschedparam Subroutine

pthread_attr_setstackaddr Subroutine

pthread_attr_setstacksize Subroutine

pthread_attr_setsuspendstate_np and pthread_attr_getsuspendstate_np Subroutine

pthread_barrier_destroy or pthread_barrier_init Subroutine

pthread_barrier_wait Subroutine

pthread_barrierattr_destroy or pthread_barrierattr_init Subroutine

pthread_barrierattr_getpshared or pthread_barrierattr_setpshared Subroutine

pthread_cleanup_pop or pthread_cleanup_push Subroutine

pthread_cond_destroy or pthread_cond_init Subroutine

PTHREAD_COND_INITIALIZER Macro

pthread_cond_signal or pthread_cond_broadcast Subroutine

pthread_cond_wait or pthread_cond_timedwait Subroutine

pthread_condattr_destroy or pthread_condattr_init Subroutine

pthread_condattr_getclock, pthread_condattr_setclock Subroutine

pthread_condattr_getpshared Subroutine

pthread_condattr_setpshared Subroutine

pthread_create Subroutine

pthread_create_withcred_np Subroutine

pthread_delay_np Subroutine

pthread_equal Subroutine

pthread_exit Subroutine

pthread_get_expiration_np Subroutine

pthread_getconcurrency or pthread_setconcurrency Subroutine

pthread_getcpuclockid Subroutine

pthread_getrusage_np Subroutine

pthread_getschedparam Subroutine

pthread_getspecific or pthread_setspecific Subroutine

pthread_getthrds_np Subroutine

pthread_getunique_np Subroutine

pthread_join or pthread_detach Subroutine

pthread_key_create Subroutine

pthread_key_delete Subroutine

pthread_kill Subroutine

pthread_lock_global_np Subroutine

pthread_mutex_init or pthread_mutex_destroy Subroutine

pthread_mutex_getprioceiling or pthread_mutex_setprioceiling Subroutine

PTHREAD_MUTEX_INITIALIZER Macro

pthread_mutex_lock, pthread_mutex_trylock, or pthread_mutex_unlock Subroutine

pthread_mutex_timedlock Subroutine

pthread_mutexattr_destroy or pthread_mutexattr_init Subroutine

pthread_mutexattr_getkind_np Subroutine

pthread_mutexattr_getprioceiling or pthread_mutexattr_setprioceiling Subroutine

pthread_mutexattr_getprotocol or pthread_mutexattr_setprotocol Subroutine

pthread_mutexattr_getpshared or pthread_mutexattr_setpshared Subroutine

pthread_mutexattr_gettype or pthread_mutexattr_settype Subroutine

pthread_mutexattr_setkind_np Subroutine

pthread_once Subroutine

PTHREAD_ONCE_INITIALIZER Macro

pthread_rwlock_init or pthread_rwlock_destroy Subroutine

pthread_rwlock_rdlock or pthread_rwlock_tryrdlock Subroutines

pthread_rwlock_timedrdlock Subroutine

pthread_rwlock_timedwrlock Subroutine

pthread_rwlock_unlock Subroutine
About This Book

This book provides experienced C programmers with complete detailed information about Base Operating System runtime services for the AIX® operating system. Runtime services are listed alphabetically, and complete descriptions are given for them. This volume contains AIX services that begin with the letters A through P. To use the book effectively, you should be familiar with commands, system calls, subroutines, file formats, and special files. This publication is also available on the documentation CD that is shipped with the operating system.

This book is part of the six-volume technical reference set, AIX 5L Version 5.3 Technical Reference, that provides information on system calls, kernel extension calls, and subroutines in the following volumes:

- AIX 5L Version 5.3 Technical Reference: Base Operating System and Extensions Volume 1 and AIX 5L Version 5.3 Technical Reference: Base Operating System and Extensions Volume 2 provide information on system calls, subroutines, functions, macros, and statements associated with base operating system runtime services.

- AIX 5L Version 5.3 Technical Reference: Communications Volume 1 and AIX 5L Version 5.3 Technical Reference: Communications Volume 2 provide information on entry points, functions, system calls, subroutines, and operations related to communications services.

- AIX 5L Version 5.3 Technical Reference: Kernel and Subsystems Volume 1 and AIX 5L Version 5.3 Technical Reference: Kernel and Subsystems Volume 2 provide information about kernel services, device driver operations, file system operations, subroutines, the configuration subsystem, the communications subsystem, the low function terminal (LFT) subsystem, the logical volume subsystem, the M-audio capture and playback adapter subsystem, the printer subsystem, the SCSI subsystem, and the serial DASD subsystem.

Highlighting

The following highlighting conventions are used in this book:

**Bold**

Identifies commands, subroutines, keywords, files, structures, directories, and other items whose names are predefined by the system. Also identifies graphical objects such as buttons, labels, and icons that the user selects.

*Italics*

Identifies parameters whose actual names or values are to be supplied by the user.

Monospace

Identifies examples of specific data values, examples of text similar to what you might see displayed, examples of portions of program code similar to what you might write as a programmer, messages from the system, or information you should actually type.

Case-Sensitivity in AIX

Everything in the AIX operating system is case-sensitive, which means that it distinguishes between uppercase and lowercase letters. For example, you can use the `ls` command to list files. If you type `LS`, the system responds that the command is “not found.” Likewise, `FILEA`, `FILEa`, and `filea` are three distinct file names, even if they reside in the same directory. To avoid causing undesirable actions to be performed, always ensure that you use the correct case.

ISO 9000

ISO 9000 registered quality systems were used in the development and manufacturing of this product.
32-Bit and 64-Bit Support for the Single UNIX Specification

Beginning with Version 5.2, the operating system is designed to support The Open Group’s Single UNIX Specification Version 3 (UNIX 03) for portability of UNIX-based operating systems. Many new interfaces, and some current ones, have been added or enhanced to meet this specification, making Version 5.2 even more open and portable for applications, while remaining compatible with previous releases of AIX.

To determine the proper way to develop a UNIX 03-portable application, you may need to refer to The Open Group’s UNIX 03 specification, which can be accessed online or downloaded from http://www.unix.org/.

Related Publications

The following books contain information about or related to application programming interfaces:

- Operating system and device management
- Networks and communication management
- AIX 5L Version 5.3 General Programming Concepts: Writing and Debugging Programs
- AIX 5L Version 5.3 Communications Programming Concepts
- AIX 5L Version 5.3 Kernel Extensions and Device Support Programming Concepts
- AIX 5L Version 5.3 Files Reference
a64l or l64a Subroutine

Purpose
Converts between long integers and base-64 ASCII strings.

Library
Standard C Library (libc.a)

Syntax
#include <stdlib.h>

long a64l (String)
char *String;

char *l64a (LongInteger)
long LongInteger;

Description
The a64l and l64a subroutines maintain numbers stored in base-64 ASCII characters. This is a notation in which long integers are represented by up to 6 characters, each character representing a digit in a base-64 notation.

The following characters are used to represent digits:

<table>
<thead>
<tr>
<th>Character</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>.</td>
<td>Represents 0.</td>
</tr>
<tr>
<td>/</td>
<td>Represents 1.</td>
</tr>
<tr>
<td>0 - 9</td>
<td>Represents the numbers 2-11.</td>
</tr>
<tr>
<td>A - Z</td>
<td>Represents the numbers 12-37.</td>
</tr>
<tr>
<td>a - z</td>
<td>Represents the numbers 38-63.</td>
</tr>
</tbody>
</table>

Parameters

String Specifies the address of a null-terminated character string.
LongInteger Specifies a long value to convert.

Return Values
The a64l subroutine takes a pointer to a null-terminated character string containing a value in base-64 representation and returns the corresponding long value. If the string pointed to by the String parameter contains more than 6 characters, the a64l subroutine uses only the first 6.

Conversely, the l64a subroutine takes a long parameter and returns a pointer to the corresponding base-64 representation. If the LongInteger parameter is a value of 0, the l64a subroutine returns a pointer to a null string.

The value returned by the l64a subroutine is a pointer into a static buffer, the contents of which are overwritten by each call.
If the '*String parameter is a null string, the a64l subroutine returns a value of 0L.

If LongInteger is 0L, the l64a subroutine returns a pointer to a null string.

### Related Information

Subroutines Overview in AIX 5L Version 5.3 General Programming Concepts: Writing and Debugging Programs.

List of Multithread Subroutines in AIX 5L Version 5.3 General Programming Concepts: Writing and Debugging Programs.

---

**abort Subroutine**

**Purpose**

Sends a SIGIOT signal to end the current process.

**Library**

Standard C Library (libc.a)

**Syntax**

```c
#include <stdlib.h>
int abort (void)
```

**Description**

The abort subroutine sends a SIGIOT signal to the current process to terminate the process and produce a memory dump. If the signal is caught and the signal handler does not return, the abort subroutine does not produce a memory dump.

If the SIGIOT signal is neither caught nor ignored, and if the current directory is writable, the system produces a memory dump in the core file in the current directory and prints an error message.

The abnormal-termination processing includes the effect of the fclose subroutine on all open streams and message-catalog descriptors, and the default actions defined as the SIGIOT signal. The SIGIOT signal is sent in the same manner as that sent by the raise subroutine with the argument SIGIOT.

The status made available to the wait or waitpid subroutine by the abort subroutine is the same as a process terminated by the SIGIOT signal. The abort subroutine overrides blocking or ignoring the SIGIOT signal.

**Note:** The SIGABRT signal is the same as the SIGIOT signal.

**Return Values**

The abort subroutine does not return a value.

**Related Information**

The exit ("exit, atexit, unatexit, exit, or Exit Subroutine" on page 242), atexit ("exit, atexit, unatexit, exit, or Exit Subroutine" on page 242), or Exit ("exit, atexit, unatexit, exit, or Exit Subroutine" on page 242) subroutine, fclose ("fclose or fflush Subroutine" on page 252) subroutine, kill ("kill or killpg Subroutine" on page 578), or killpg ("kill or killpg Subroutine" on page 578) subroutine, raise subroutine, sigaction subroutine, sigvec subroutine, wait or waitpid subroutine.

The dbx command.
Subroutines Overview in AIX 5L Version 5.3 General Programming Concepts: Writing and Debugging Programs.

abs, div, labs, ldiv, imul_dbl, umul_dbl, llabs, or lldiv Subroutine

Purpose
Computes absolute value, division, and double precision multiplication of integers.

Library
Standard C Library (libc.a)

Syntax
#include <stdlib.h>
int abs (i)
int i;

#include <stdlib.h>
long labs (i)
long i;

#include <stdlib.h>

div_t div (Numerator, Denominator)
int Numerator: Denominator;
#include <stdlib.h>

void imul_dbl (i, j, Result)
long i, j;
long *Result;
#include <stdlib.h>

ldiv_t ldiv (Numerator, Denominator)
long Numerator: Denominator;
#include <stdlib.h>

void umul_dbl (i, j, Result)
unsigned long i, j;
unsigned long *Result;
#include <stdlib.h>

long long labs(i)
long long int i;
#include <stdlib.h>

ldiv_t lldiv (Numerator, Denominator)
long long int Numerator, Denominator;

Description
The abs subroutine returns the absolute value of its integer operand.

Note: A twos-complement integer can hold a negative number whose absolute value is too large for the integer to hold. When given this largest negative value, the abs subroutine returns the same value.

The div subroutine computes the quotient and remainder of the division of the number represented by the Numerator parameter by that specified by the Denominator parameter. If the division is inexact, the sign of the resulting quotient is that of the algebraic quotient, and the magnitude of the resulting quotient is the largest integer less than the magnitude of the algebraic quotient. If the result cannot be represented (for example, if the denominator is 0), the behavior is undefined.
The labs and ldiv subroutines are included for compatibility with the ANSI C library, and accept long integers as parameters, rather than as integers.

The imul_dbl subroutine computes the product of two signed longs, \( i \) and \( j \), and stores the double long product into an array of two signed longs pointed to by the Result parameter.

The umul_dbl subroutine computes the product of two unsigned longs, \( i \) and \( j \), and stores the double unsigned long product into an array of two unsigned longs pointed to by the Result parameter.

The llabs and lldiv subroutines compute the absolute value and division of long long integers. These subroutines operate under the same restrictions as the abs and div subroutines.

**Note:** When given the largest negative value, the llabs subroutine (like the abs subroutine) returns the same value.

### Parameters

- **\( i \)** Specifies, for the abs subroutine, some integer; for labs and imul_dbl, some long integer; for the umul_dbl subroutine, some unsigned long integer; for the llabs subroutine, some long long integer.

- **Numerator** Specifies, for the div subroutine, some integer; for the ldiv subroutine, some long integer; for lldiv, some long long integer.

- **\( j \)** Specifies, for the imul_dbl subroutine, some long integer; for the umul_dbl subroutine, some unsigned long integer.

- **Denominator** Specifies, for the div subroutine, some integer; for the ldiv subroutine, some long integer; for lldiv, some long long integer.

- **Result** Specifies, for the imul_dbl subroutine, some long integer; for the umul_dbl subroutine, some unsigned long integer.

### Return Values

The abs, labs, and llabs subroutines return the absolute value. The imul_dbl and umul_dbl subroutines have no return values. The div subroutine returns a structure of type div_t. The ldiv subroutine returns a structure of type ldiv_t, comprising the quotient and the remainder. The structure is displayed as:

```c
struct ldiv_t {
    int quot; /* quotient */
    int rem; /* remainder */
};
```

The lldiv subroutine returns a structure of type lldiv_t, comprising the quotient and the remainder.

---

### access, accessx, or faccessx Subroutine

**Purpose**

Determines the accessibility of a file.

**Library**

Standard C Library (libc.a)

**Syntax**

```c
#include <unistd.h>

int access (PathName, Mode)
char *PathName;
```
int Mode;

int accessx (PathName, Mode, Who);

int faccessx (FileDescriptor, Mode, Who);

Description
The access, accessx, and faccessx subroutines determine the accessibility of a file system object. The accessx and faccessx subroutines allow the specification of a class of users or processes for whom access is to be checked.

The caller must have search permission for all components of the PathName parameter.

Parameters

PathName
Specifies the path name of the file. If the PathName parameter refers to a symbolic link, the access subroutine returns information about the file pointed to by the symbolic link.

FileDescriptor
Specifies the file descriptor of an open file.

Mode
Specifies the access modes to be checked. This parameter is a bit mask containing 0 or more of the following values, which are defined in the sys/access.h file:

R_OK   Check read permission.
W_OK   Check write permission.
X_OK   Check execute or search permission.
F_OK   Check the existence of a file.

If none of these values are specified, the existence of a file is checked.

Who
Specifies the class of users for whom access is to be checked. This parameter must be one of the following values, which are defined in the sys/access.h file:

ACC_SELF
Determines if access is permitted for the current process. The effective user and group IDs, the concurrent group set and the privilege of the current process are used for the calculation.

ACC_INVOKE
Determines if access is permitted for the invoker of the current process. The real user and group IDs, the concurrent group set, and the privilege of the invoker are used for the calculation.

Note: The expression access (PathName, Mode) is equivalent to accessx (PathName, Mode, ACC_INVOKER).

ACC_OTHERS
Determines if the specified access is permitted for any user other than the object owner. The Mode parameter must contain only one of the valid modes. Privilege is not considered in the calculation.

ACC_ALL
Determines if the specified access is permitted for all users. The Mode parameter must contain only one of the valid modes. Privilege is not considered in the calculation.

Note: The accessx subroutine shows the same behavior by both the user and root with ACC_ALL.
Return Values
If the requested access is permitted, the access, accessx, and faccessx subroutines return a value of 0. If the requested access is not permitted or the function call fails, a value of -1 is returned and the errno global variable is set to indicate the error.

The access subroutine indicates success for X_OK even if none of the execute file permission bits are set.

Error Codes
The access and accessx subroutines fail if one or more of the following are true:

<table>
<thead>
<tr>
<th>Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EACCES</td>
<td>Search permission is denied on a component of the PathName prefix.</td>
</tr>
<tr>
<td>EFAULT</td>
<td>The PathName parameter points to a location outside the allocated address space of the process.</td>
</tr>
<tr>
<td>ELOOP</td>
<td>Too many symbolic links were encountered in translating the PathName parameter.</td>
</tr>
<tr>
<td>ENOENT</td>
<td>A component of the PathName does not exist or the process has the disallow truncation attribute set.</td>
</tr>
<tr>
<td>ENOTDIR</td>
<td>A component of the PathName is not a directory.</td>
</tr>
<tr>
<td>ESTALE</td>
<td>The process root or current directory is located in a virtual file system that has been unmounted.</td>
</tr>
<tr>
<td>EPERM</td>
<td>The named file does not exist.</td>
</tr>
<tr>
<td>EPERM</td>
<td>The PathName parameter was null.</td>
</tr>
<tr>
<td>ENAMETOOLONG</td>
<td>A symbolic link was named, but the file to which it refers does not exist.</td>
</tr>
<tr>
<td>ENAMETOOLONG</td>
<td>A component of the PathName parameter exceeded 255 characters or the entire PathName parameter exceeded 1022 characters.</td>
</tr>
</tbody>
</table>

The faccessx subroutine fails if the following is true:

<table>
<thead>
<tr>
<th>Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EBADF</td>
<td>The value of the FileDescriptor parameter is not valid.</td>
</tr>
</tbody>
</table>

The access, accessx, and faccessx subroutines fail if one or more of the following is true:

<table>
<thead>
<tr>
<th>Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EIO</td>
<td>An I/O error occurred during the operation.</td>
</tr>
<tr>
<td>EACCES</td>
<td>The file protection does not allow the requested access.</td>
</tr>
<tr>
<td>EROFS</td>
<td>Write access is requested for a file on a read-only file system.</td>
</tr>
</tbody>
</table>

If Network File System (NFS) is installed on your system, the accessx and faccessx subroutines can also fail if the following is true:

<table>
<thead>
<tr>
<th>Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ETIMEDOUT</td>
<td>The connection timed out.</td>
</tr>
<tr>
<td>ETXTBSY</td>
<td>Write access is requested for a shared text file that is being executed.</td>
</tr>
<tr>
<td>EINVAL</td>
<td>The value of the Mode argument is invalid.</td>
</tr>
</tbody>
</table>

Related Information
The acl_get subroutine, chacl subroutine, chown command, chmod command.

Files, Directories, and File Systems for Programmers in AIX 5L Version 5.3 General Programming Concepts: Writing and Debugging Programs.
acct Subroutine

Purpose
Enables and disables process accounting.

Library
Standard C Library (libc.a)

Syntax
```
int acct (Path)
char *Path;
```

Description
The acct subroutine enables the accounting routine when the Path parameter specifies the path name of the file to which an accounting record is written for each process that terminates. When the Path parameter is a 0 or null value, the acct subroutine disables the accounting routine.

If the Path parameter refers to a symbolic link, the acct subroutine causes records to be written to the file pointed to by the symbolic link.

If Network File System (NFS) is installed on your system, the accounting file can reside on another node.

Note: To ensure accurate accounting, each node must have its own accounting file. Although no two nodes should share accounting files, a node’s accounting files can be located on any node in the network.

The calling process must have root user authority to use the acct subroutine.

Parameters

Path Specifies a pointer to the path name of the file or a null pointer.

Return Values
Upon successful completion, the acct subroutine returns a value of 0. Otherwise, a value of -1 is returned and the global variable errno is set to indicate the error.

Error Codes
The acct subroutine is unsuccessful if one or more of the following are true:

- EACCES Write permission is denied for the named accounting file.
- EACCES The file named by the Path parameter is not an ordinary file.
- EBUSY An attempt is made to enable accounting when it is already enabled.
- ENOENT The file named by the Path parameter does not exist.
- EPERM The calling process does not have root user authority.
- EROFS The named file resides on a read-only file system.

If NFS is installed on the system, the acct subroutine is unsuccessful if the following is true:

- ETIMEDOUT The connection timed out.
acl_chg or acl_fchg Subroutine

Purpose
Changes the AIXC ACL type access control information on a file.

Library
Security Library (libc.a)

Syntax
#include <sys/access.h>

int acl_chg (Path, How, Mode, Who)
char *Path;
int How;
int Mode;
int Who;

int acl_fchg (FileDescriptor, How, Mode, Who)
int FileDescriptor;
int How;
int Mode;
int Who;

Description
The acl_chg and acl_fchg subroutines modify the AIXC ACL-type-based access control information of a specified file. This call can fail for file system objects with any non-AIXC ACL.

Parameters

FileDescriptor
Specifies the file descriptor of an open file.

How
Specifies how the permissions are to be altered for the affected entries of the Access Control List (ACL). This parameter takes one of the following values:

- **ACC_PERMIT**
  Allows the types of access included in the Mode parameter.

- **ACC_DENY**
  Denies the types of access included in the Mode parameter.

- **ACC_SPECIFY**
  Grants the access modes included in the Mode parameter and restricts the access modes not included in the Mode parameter.

Mode
Specifies the access modes to be changed. The Mode parameter is a bit mask containing zero or more of the following values:

- **R_ACC**
  Allows read permission.

- **W_ACC**
  Allows write permission.

- **X_ACC**
  Allows execute or search permission.

Path
Specifies a pointer to the path name of a file.
Who Specifies which entries in the ACL are affected. This parameter takes one of the following values:

- **ACC_OBJ_OWNER**
  Changes the owner entry in the base ACL.

- **ACC_OBJ_GROUP**
  Changes the group entry in the base ACL.

- **ACC_OTHERS**
  Changes all entries in the ACL except the base entry for the owner.

- **ACC_ALL**
  Changes all entries in the ACL.

Return Values
On successful completion, the `acl_chg` and `acl_fchg` subroutines return a value of 0. Otherwise, a value of -1 is returned and the `errno` global variable is set to indicate the error.

Error Codes
The `acl_chg` subroutine fails and the access control information for a file remains unchanged if one or more of the following is true:

- **EACCES**
  Search permission is denied on a component of the `Path` prefix.

- **EFAULT**
  The `Path` parameter points to a location outside of the allocated address space of the process.

- **ELOOP**
  Too many symbolic links were encountered in translating the `Path` parameter.

- **ENAMETOOLONG**
  A component of the `Path` parameter exceeded 255 characters, or the entire `Path` parameter exceeded 1023 characters.

- **ENOENT**
  A component of the `Path` does not exist or has the `disallow truncation` attribute (see the `ulimit` subroutine).

- **ENOENT**
  The `Path` parameter was null.

- **ENOENT**
  A symbolic link was named, but the file to which it refers does not exist.

- **ENOTDIR**
  A component of the `Path` prefix is not a directory.

- **ESTALE**
  The process' root or current directory is located in a virtual file system that has been unmounted.

The `acl_fchg` subroutine fails and the file permissions remain unchanged if the following is true:

- **EBADF**
  The `FileDescriptor` value is not valid.

The `acl_chg` or `acl_fchg` subroutine fails and the access control information for a file remains unchanged if one or more of the following is true:

- **EINVAL**
  The `How` parameter is not one of `ACC_PERMIT`, `ACC_DENY`, or `ACC_SPECIFY`.

- **EINVAL**
  The `Who` parameter is not `ACC_OWNER`, `ACC_GROUP`, `ACC_OTHERS`, or `ACC_ALL`.

- **EROFS**
  The named file resides on a read-only file system.

The `acl_chg` or `acl_fchg` subroutine fails and the access control information for a file remains unchanged if one or more of the following is true:

- **EIO**
  An I/O error occurred during the operation.

- **EPERM**
  The effective user ID does not match the ID of the owner of the file and the invoker does not have root user authority.
If Network File System (NFS) is installed on your system, the `acl_chg` and `acl_fchg` subroutines can also fail if the following is true:

**ETIMEDOUT** The connection timed out.

**Related Information**

The `acl_get` ("acl_get or acl_fget Subroutine" on page 12) subroutine, `acl_put` ("acl_put or acl_fput Subroutine" on page 14) subroutine, `chac1` ("chac1 or fchac1 Subroutine" on page 144) subroutine, `chmod` ("chmod or fchmod Subroutine" on page 148) subroutine, `stat` subroutine, `statacl` subroutine.

The `aclget` command, `aclput` command, `chmod` command.

List of Security and Auditing Subroutines and Subroutines Overview in AIX 5L Version 5.3 General Programming Concepts: Writing and Debugging Programs.

---

**acl_get or acl_fget Subroutine**

**Purpose**

Gets the access control information of a file if the ACL associated is of the AIXC type.

**Library**

Security Library (libc.a)

**Syntax**

```
#include <sys/access.h>

char *acl_get (Path)
char *Path;

char *acl_fget (FileDescriptor)
int FileDescriptor;
```

**Description**

The `acl_get` and `acl_fget` subroutines retrieve the access control information for a file system object. This information is returned in a buffer pointed to by the return value. The structure of the data in this buffer is unspecified. The value returned by these subroutines should be used only as an argument to the `acl_put` or `acl_fput` subroutines to copy or restore the access control information. Note that `acl_get` and `acl_fget` subroutines could fail if the ACL associated with the file system object is of a different type than AIXC. It is recommended that applications make use of `aclx_get` and `aclx_fget` subroutines to retrieve the ACL.

The buffer returned by the `acl_get` and `acl_fget` subroutines is in allocated memory. After usage, the caller should deallocate the buffer using the `free` subroutine.

**Parameters**

- **Path** Specifies the path name of the file.
- **FileDescriptor** Specifies the file descriptor of an open file.
Return Values
On successful completion, the acl_get and acl_fget subroutines return a pointer to the buffer containing the access control information. Otherwise, a null pointer is returned and the errno global variable is set to indicate the error.

Error Codes
The acl_get subroutine fails if one or more of the following are true:

- **EACCESS** Search permission is denied on a component of the Path prefix.
- **EFAULT** The Path parameter points to a location outside of the allocated address space of the process.
- **ELOOP** Too many symbolic links were encountered in translating the Path parameter.
- **ENAMETOOLONG** A component of the Path parameter exceeded 255 characters, or the entire Path parameter exceeded 1023 characters.
- **ENOTDIR** A component of the Path prefix is not a directory.
- **ENOENT** A component of the Path does not exist or the process has the disallow truncation attribute (see the ulimit subroutine).
- **ENOENT** The Path parameter was null.
- **ENOENT** A symbolic link was named, but the file to which it refers does not exist.
- **ESTALE** The process' root or current directory is located in a virtual file system that has been unmounted.

The acl_fget subroutine fails if the following is true:

- **EBADF** The FileDescriptor parameter is not a valid file descriptor.

The acl_get or acl_fget subroutine fails if the following is true:

- **EIO** An I/O error occurred during the operation.

If Network File System (NFS) is installed on your system, the acl_get and acl_fget subroutines can also fail if the following is true:

- **ETIMEDOUT** The connection timed out.

Security
Access Control The invoker must have search permission for all components of the Path prefix.
Audit Events None.

Related Information
The acl_chg or acl_fchg subroutine, acl_put or acl_fput subroutine, acl_set or acl_fset subroutine, aclx_get or aclx_fget subroutine, aclx_put or aclx_fput subroutine, chmod subroutine, stat subroutine, and the aclget command, aclput command, and chmod command.

List of Security and Auditing Subroutines and Subroutines Overview in AIX 5L Version 5.3 General Programming Concepts: Writing and Debugging Programs.
acl_put or acl_fput Subroutine

Purpose
Sets AIXC ACL type access control information of a file.

Library
Security Library (libc.a)

Syntax
```
#include <sys/access.h>

int acl_put (Path, Access, Free)
char * Path;
char * Access;
int Free;

int acl_fput (FileDescriptor, Access, Free)
int FileDescriptor;
char * Access;
int Free;
```

Description
The acl_put and acl_fput subroutines set the access control information of a file system object. This information is contained in a buffer returned by a call to the acl_get or acl_fget subroutine. The structure of the data in this buffer is unspecified. However, the entire Access Control List (ACL) for a file cannot exceed one memory page (4096 bytes) in size. Note that acl_put/acl_fput operation could fail if the existing ACL associated with the file system object is of a different kind or if the underlying physical file system does not support AIXC ACL type. It is recommended that applications make use of aclx_put and aclx_fput subroutines to set the ACL instead of acl_put/acl_fput routines.

Parameters
- **Path**
  Specifies the path name of a file.
- **FileDescriptor**
  Specifies the file descriptor of an open file.
- **Access**
  Specifies a pointer to the buffer containing the access control information.
- **Free**
  Specifies whether the buffer space is to be deallocated. The following values are valid:
  - 0  Space is not deallocated.
  - 1  Space is deallocated.

Return Values
On successful completion, the acl_put and acl_fput subroutines return a value of 0. Otherwise, -1 is returned and the errno global variable is set to indicate the error.

Error Codes
The acl_put subroutine fails and the access control information for a file remains unchanged if one or more of the following are true:

- **EACCES**
  Search permission is denied on a component of the Path prefix.
- **EFAULT**
  The Path parameter points to a location outside of the allocated address space of the process.
- **ELOOP**
  Too many symbolic links were encountered in translating the Path parameter.
ENAMETOOLONG A component of the \texttt{Path} parameter exceeded 255 characters, or the entire \texttt{Path} parameter exceeded 1023 characters.

ENOENT A component of the \texttt{Path} does not exist or has the \texttt{disallow truncation} attribute (see the \texttt{ulimit} subroutine).

ENOENT The \texttt{Path} parameter was null.

ENOENT A symbolic link was named, but the file to which it refers does not exist.

ENOTDIR A component of the \texttt{Path} prefix is not a directory.

ESTALE The process’ root or current directory is located in a virtual file system that has been unmounted.

The \texttt{acl_fput} subroutine fails and the file permissions remain unchanged if the following is true:

EBADF The \texttt{FileDescriptor} parameter is not a valid file descriptor.

The \texttt{acl_put} or \texttt{acl_fput} subroutine fails and the access control information for a file remains unchanged if one or more of the following are true:

EINVAL The \texttt{Access} parameter does not point to a valid access control buffer.

EINVAL The \texttt{Free} parameter is not 0 or 1.

EIO An I/O error occurred during the operation.

EROFS The named file resides on a read-only file system.

If Network File System (NFS) is installed on your system, the \texttt{acl_put} and \texttt{acl_fput} subroutines can also fail if the following is true:

ETIMEDOUT The connection timed out.

Security
Access Control: The invoker must have search permission for all components of the \texttt{Path} prefix.

Auditing Events:

<table>
<thead>
<tr>
<th>Event</th>
<th>Information</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>chacl</td>
<td>Path</td>
</tr>
<tr>
<td>fchacl</td>
<td>FileDescriptor</td>
</tr>
</tbody>
</table>

Related Information

The \texttt{acl_chg} ("acl_chg or acl_fchg Subroutine" on page 8) subroutine, \texttt{acl_get} ("acl_get or acl_fget Subroutine" on page 10) subroutine, \texttt{acl_set} ("acl_set or acl_fset Subroutine" on page 14) subroutine, \texttt{chacl} ("chacl or fchacl Subroutine" on page 144) subroutine, \texttt{chmod} ("chmod or fchmod Subroutine" on page 148) subroutine, \texttt{stat} subroutine, \texttt{statacl} subroutine.

"aclx_get or aclx_fget Subroutine" on page 17, "aclx_put or aclx_fput Subroutine" on page 25.

The \texttt{aclget} command, \texttt{aclput} command, \texttt{chmod} command.

[List of Security and Auditing Subroutines and Subroutines Overview in AIX 5L Version 5.3 General Programming Concepts: Writing and Debugging Programs.]
acl_set or acl_fset Subroutine

Purpose
Sets the AIXC ACL type access control information of a file.

Library
Security Library (libc.a)

Syntax
#include <sys/access.h>

int acl_set (char *Path, int OwnerMode, int GroupMode, int DefaultMode);
int acl_fset (int *FileDescriptor, int OwnerMode, int GroupMode, int DefaultMode);

Description
The acl_set and acl_fset subroutines set the base entries of the Access Control List (ACL) of the file. All other entries are discarded. Other access control attributes are left unchanged. Note that if the file system object is associated with any other ACL type access control information, it will be replaced with just the Base mode bits information. It is strongly recommended that applications stop using these interfaces and instead make use of aclx_put and aclx_fput subroutines to set the ACL.

Parameters
DefaultMode
Specifies the access permissions for the default class.

FileDescriptor
Specifies the file descriptor of an open file.

GroupMode
Specifies the access permissions for the group of the file.

OwnerMode
Specifies the access permissions for the owner of the file.

Path
Specifies a pointer to the path name of a file.

The mode parameters specify the access permissions in a bit mask containing zero or more of the following values:

R_ACC
Authorize read permission.

W_ACC
Authorize write permission.

X_ACC
Authorize execute or search permission.

Return Values
Upon successful completion, the acl_set and acl_fset subroutines return the value 0. Otherwise, the value -1 is returned and the errno global variable is set to indicate the error.
Error Codes
The acl_set subroutine fails and the access control information for a file remains unchanged if one or more of the following are true:

<table>
<thead>
<tr>
<th>Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EACCES</td>
<td>Search permission is denied on a component of the Path prefix.</td>
</tr>
<tr>
<td>EFAULT</td>
<td>The Path parameter points to a location outside of the allocated address space of the process.</td>
</tr>
<tr>
<td>ELOOP</td>
<td>Too many symbolic links were encountered in translating the Path parameter.</td>
</tr>
<tr>
<td>ENAMETOOLONG</td>
<td>A component of the Path parameter exceeded 255 characters, or the entire Path parameter exceeded 1023 characters.</td>
</tr>
<tr>
<td>ENOENT</td>
<td>A component of the Path does not exist or has the disallow truncation attribute (see the ulimit subroutine).</td>
</tr>
<tr>
<td>ENOENT</td>
<td>The Path parameter was null.</td>
</tr>
<tr>
<td>ENOENT</td>
<td>A symbolic link was named, but the file to which it refers does not exist.</td>
</tr>
<tr>
<td>ENOTDIR</td>
<td>A component of the Path prefix is not a directory.</td>
</tr>
<tr>
<td>ESTALE</td>
<td>The process' root or current directory is located in a virtual file system that has been unmounted.</td>
</tr>
</tbody>
</table>

The acl_fset subroutine fails and the file permissions remain unchanged if the following is true:

<table>
<thead>
<tr>
<th>Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EBADF</td>
<td>The file descriptor FileDescriptor is not valid.</td>
</tr>
</tbody>
</table>

The acl_set or acl_fset subroutine fails and the access control information for a file remains unchanged if one or more of the following are true:

<table>
<thead>
<tr>
<th>Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EIO</td>
<td>An I/O error occurred during the operation.</td>
</tr>
<tr>
<td>EPERM</td>
<td>The effective user ID does not match the ID of the owner of the file and the invoker does not have root user authority.</td>
</tr>
<tr>
<td>EROFS</td>
<td>The named file resides on a read-only file system.</td>
</tr>
</tbody>
</table>

If Network File System (NFS) is installed on your system, the acl_set and acl_fset subroutines can also fail if the following is true:

<table>
<thead>
<tr>
<th>Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ETIMEDOUT</td>
<td>The connection timed out.</td>
</tr>
</tbody>
</table>

Security
Access Control: The invoker must have search permission for all components of the Path prefix.

Auditing Events:

<table>
<thead>
<tr>
<th>Event</th>
<th>Information</th>
</tr>
</thead>
<tbody>
<tr>
<td>changl</td>
<td>Path</td>
</tr>
<tr>
<td>fchangl</td>
<td>FileDescriptor</td>
</tr>
</tbody>
</table>

Related Information
The acl_chg subroutine, acl_get subroutine, acl_put subroutine, chmod subroutine, chmod subroutine, stat subroutine, statacl subroutine.
aclx_convert Subroutine

Purpose
Converts the access control information from one ACL type to another.

Library
Security Library (libc.a)

Syntax
#include <sys/acl.h>

int aclx_convert (from_acl, from_sz, from_type, to_acl, to_sz, to_type, fs_obj_path)
void * from_acl;
size_t from_sz;
acl_type_t from_type;
void * to_acl;
size_t * to_sz;
acl_type_t to_type;
char * fs_obj_path;

Description
The aclx_convert subroutine converts the access control information from the binary input given in from_acl of the ACL type from_type into a binary ACL of the type to_type and stores it in to_acl. Values from_type and to_type can be any ACL types supported in the system.

The ACL conversion takes place with the help of an ACL type-specific algorithm. Because the conversion is approximate, it can result in a potential loss of access control. Therefore, the user of this call must make sure that the converted ACL satisfies the required access controls. The user can manually review the access control information after the conversion for the file system object to ensure that the conversion was successful and satisfied the requirements of the intended access control.

Parameters

from_acl Points to the ACL that has to be converted.
from_sz Indicates the size of the ACL information pointed to by from_acl.
from_type Indicates the ACL type information of the ACL. The acl_type is 64 bits in size and is unique on the system. If the given acl_type is not supported in the system, this function fails and errno is set to EINVAL.
to_acl Points to a buffer in which the target binary ACL has to be stored. The amount of memory available in this buffer is indicated by the to_sz parameter.
to_sz Indicates the amount of memory, in bytes, available in to_acl. If to_sz contains less than the required amount of memory for storing the converted ACL, *to_sz is set to the required amount of memory and ENOSPC is returned by errno.
to_type Indicates the ACL type to which conversion needs to be done. The ACL type is 64 bits in size and is unique on the system. If the given acl_type is not supported in the system, this function fails and errno is set to EINVAL.
fs_obj_path File System Object Path for which the ACL conversion is being requested. Gets information about the object, such as whether it is file or directory.
Return Values
On successful completion, the aclx_convert subroutine returns a value of 0. Otherwise, -1 is returned and the errno global variable is set to indicate the error.

Error Codes
The aclx_convert subroutine fails if one or more of the following is true:

EINVAL    Invalid input parameter. The same error can be returned if an invalid acl_type is specified as input to this routine, either in from_type or in to_type. This errno could also be returned if the binary ACL given in from_acl is not the type specified by from_type.
ENOSPC    Insufficient storage space is available in to_acl.

Security
Access Control: The invoker must have search permission for all components of the Path prefix.

Auditing Events: If the auditing subsystem has been properly configured and is enabled, the aclx_convert subroutine generates the following audit record (event) every time the command is executed:

<table>
<thead>
<tr>
<th>Event</th>
<th>Information</th>
</tr>
</thead>
<tbody>
<tr>
<td>FILE_Acl</td>
<td>Lists access controls.</td>
</tr>
</tbody>
</table>

Related Information
The aclget command, aclput command, aclconvert command.

List of Security and Auditing Subroutines and Subroutines Overview in AIX 5L Version 5.3 General Programming Concepts: Writing and Debugging Programs.

aclx_get or aclx_fget Subroutine

Purpose
Gets the access control information for a file system object.

Library
Security Library (libc.a)

Syntax
#include <sys/acl.h>

int aclx_get (Path, ctl_flags, acl_type, acl, acl_sz, mode_info)
char *Path;
uint64_t ctl_flags;
acl_type_t *acl_type;
void *acl;
size_t *acl_sz;
mode_t *mode_info;

int aclx_fget (FileDescriptor, ctl_flags, acl_type, acl, acl_sz, mode_info)
int FileDescriptor;
uint64_t ctl_flags;
acl_type_t *acl_type;
Description

The `aclx_get` and `aclx_fget` subroutines retrieve the access control information for a file system object in the native ACL format. Native ACL format is the format as defined for the particular ACL type in the system. These subroutines are advanced versions of the `acl_get` and `acl_fget` subroutines and should be used instead of the older versions. The `aclx_get` and `aclx_fget` subroutines provide for more control for the user to interact with the underlying file system directly.

In the earlier versions (`acl_get` or `acl_fget`), OS libraries found out the ACL size from the file system and allocated the required memory buffer space to hold the ACL information. The caller does all this now with the `aclx_get` and `aclx_fget` subroutines. Callers are responsible for finding out the size and allocating memory for the ACL information, and later freeing the same memory after it is used. These subroutines allow for an `acl_type` input and output argument. The data specified in this argument can be set to a particular ACL type and a request for the ACL on the file system object of the same type. Some physical file systems might do emulation to return the ACL type requested, if the ACL type that exists on the file system object is different. If the `acl_type` pointer points to a data area with a value of `ACL_ANY` or 0, then the underlying physical file system has to return the type of the ACL associated with the file system object.

The `ctl_flags` parameter is a bit mask that allows for control over the `aclx_get` requests.

The value returned by these subroutines can be use as an argument to the `aclx_get` or `aclx_fget` subroutines to copy or restore the access control information.

Parameters

- **Path**: Specifies the path name of the file system object.
- **FileDescriptor**: Specifies the file descriptor of an open file.
- **ctl_flags**: This 64-bit sized bit mask provides control over the ACL retrieval. The following flag values are defined:
  - **GET_ACLINFO_ONLY**: Gets only the ACL type and length information from the underlying file system. When this bit is set, arguments such as `acl` and `mode_info` can be set to NULL. In all other cases, these should be valid buffer pointers (or else an error is returned). If this bit is not specified, then all the other information about the ACL, such as ACL data and mode information, is returned.
- **acl_type**: Points to a buffer that will hold ACL type information. The ACL type is 64 bits in size and is unique on the system. The caller can provide an ACL type in this area and a request for the ACL on the file system object of the same type. If the ACL type requested does not match the one on the file system object, the physical file system might return an error or emulate and provide the ACL information in the ACL type format requested. If the caller does not know the ACL type and wants to retrieve the ACL associated with the file system object, then the caller should set the buffer value pointed to by `acl_type` to `ACL_ANY` or 0.
- **acl**: Points to a buffer where the ACL retrieved is stored. The size of this buffer is indicated by the `acl_sz` parameter.
- **acl_sz**: Indicates the size of the buffer area passed through the `acl` parameter.
- **mode_info**: Pointer to a buffer where the mode word associated with the file system object is returned. Note that this mode word’s meaning and formations depend entirely on the ACL type concerned.
Return Values
On successful completion, the aclx_put and aclx_fput subroutines return a value of 0. Otherwise, -1 is returned and the errno global variable is set to indicate the error.

Error Codes
The aclx_get subroutine fails if one or more of the following is true:

- **EACCES**: Search permission is denied on a component of the Path prefix.
- **EFAULT**: The Path parameter points to a location outside of the allocated address space of the process.
- **ELOOP**: Too many symbolic links were encountered in translating the Path parameter.
- **ENAMETOOLONG**: A component of the Path parameter exceeded 255 characters, or the entire Path parameter exceeded 1023 characters.
- **ENOENT**: A component of the Path does not exist or has the disallow truncation attribute (see the ulimit subroutine).
- **ENOENT**: The Path parameter was null.
- **ENOENT**: A symbolic link was named, but the file to which it refers does not exist.
- **ENOTDIR**: A component of the Path prefix is not a directory.
- **ESTALE**: The process' root or current directory is located in a virtual file system that has been unmounted.

The aclx_fget subroutine fails if the following is true:

- **EBADF**: The FileDescriptor parameter is not a valid file descriptor.

The aclx_get or aclx_fget subroutine fails if one or more of the following is true:

- **EINVAL**: Invalid input parameter. The same error can be returned if an invalid acl_type is specified as input to this routine.
- **EIO**: An I/O error occurred during the operation.
- **ENOSPC**: Input buffer size acl_sz is not sufficient to store the ACL data in acl.

If Network File System (NFS) is installed on your system, the aclx_get and aclx_fget subroutines can also fail if the following condition is true:

- **ETIMEDOUT**: The connection timed out.

Security
Access Control: The invoker must have search permission for all components of the Path prefix.

Auditing Events: None

Related Information
The acl_chg ("acl_chg or acl_fchg Subroutine" on page 8) subroutine, acl_put ("acl_get or acl_fget Subroutine" on page 10) subroutine, acl_set ("acl_set or acl_fset Subroutine" on page 14) subroutine, chacl ("chacl or fchacl Subroutine" on page 144) subroutine, chmod ("chmod or fchmod Subroutine" on page 148) subroutine, stat ("stat subroutine," statacl ("statacl subroutine," "aclx_convert Subroutine" on page 16).

The aclget command, aclput command, chmod command.

List of Security and Auditing Subroutines and Subroutines Overview in AIX 5L Version 5.3 General Programming Concepts: Writing and Debugging Programs.
aclx_gettypeinfo Subroutine

Purpose
Retrieves the ACL characteristics given to an ACL type.

Library
Security Library (libc.a)

Syntax
#include <sys/acl.h>

int aclx_gettypeinfo (Path, acl_type, buffer, buffer_sz)

Parameters
Path Specifies the path name of the file.
acl_type ACL type for which the characteristics are sought.
buffer Specifies the pointer to a buffer space, where the characteristics of acl_type for the file system is returned. The structure of data returned is ACL type-specific. Refer to the ACL type-specific documentation for more details.
buffer_sz Points to an area that specifies the length of the buffer buffer in which the characteristics of acl_type are returned by the file system. This is an input/output parameter. If the length of the buffer provided is not sufficient to store all the ACL type characteristic information, then the file system returns an error and indicates the length of the buffer required in this variable. The length is specified in number of bytes.

Return Values
On successful completion, the aclx_gettypeinfo subroutine returns a value of 0. Otherwise, -1 is returned and the errno global variable is set to indicate the error.

Error Codes
The aclx_gettypeinfo subroutine fails and the access control information for a file remains unchanged if one or more of the following is true:

EACCES Search permission is denied on a component of the Path prefix.
EFAULT The Path parameter points to a location outside of the allocated address space of the process.
ELOOP Too many symbolic links were encountered in translating the Path parameter.
ENAMETOOLONG A component of the Path parameter exceeded 255 characters, or the entire Path parameter exceeded 1023 characters.
ENOENT A component of the Path does not exist or has the disallow truncation attribute (see the ulimit subroutine).
ENOENT  The \textit{Path} parameter was null.
ENOENT  A symbolic link was named, but the file to which it refers does not exist.
ENOSPC  Buffer space provided is not enough to store all the \textit{acl_type} characteristics of the file system.
ENOTDIR  A component of the \textit{Path} prefix is not a directory.
ESTALE  The process' root or current directory is located in a virtual file system that has been unmounted.

If Network File System (NFS) is installed on your system, the \texttt{aclx_gettypeinfo} subroutine can also fail if the following condition is true:

ETIMEDOUT  The connection timed out.

\section*{Security}
Auditing Events: None

\section*{Related Information}
The \texttt{"aclx\_get or aclx\_fget Subroutine" on page 17, \"aclx\_put or aclx\_fput Subroutine" on page 25.}
The \texttt{aclget} command, \texttt{aclput} command.

\begin{center}
List of Security and Auditing Subroutines and \texttt{Subroutines Overview} in AIX 5L Version 5.3 General Programming Concepts: Writing and Debugging Programs.
\end{center}

### aclx\_gettypes Subroutine

\section*{Purpose}
Retrieves the list of ACL types supported for the file system associated with the path provided.

\section*{Library}
Security Library (libc.a)

\section*{Syntax}
\begin{verbatim}
#include <sys/acl.h>

int aclx_gettypes (Path, acl_type_list, acl_type_list_len)
char * Path;
acl_types_list_t * acl_type_list;
size_t * acl_type_list_len;
\end{verbatim}

\section*{Description}
The \texttt{aclx\_gettypes} subroutine helps obtain the list of ACL types supported on the particular file system. A file system can implement policies to support one to many ACL types simultaneously. The first ACL type in the list is the default ACL type for the file system. This default ACL type is used in ACL conversions if the target ACL type is not supported on the file system. Each file system object in the file system is associated with only one piece of ACL data of a particular ACL type.
Parameters

Path
Specifies the path name of the file system object within the file system for which the list of supported ACLs are being requested.

acl_type_list
Specifies the pointer to a buffer space, where the list of ACL types is returned. The size of this buffer is indicated using the acl_type_list_len argument in bytes.

acl_type_list_len
Pointer to a buffer that specifies the length of the buffer acl_type_list in which the list of ACLs is returned by the file system. This is an input/output parameter. If the length of the buffer is not sufficient to store all the ACL types, the file system returns an error and indicates the length of the buffer required in this same area. The length is specified in bytes.
If the subroutine call is successful, this field contains the number of bytes of information stored in the acl_type_list buffer. This information can be used by the caller to get the number of ACL type entries returned.

Return Values
On successful completion, the aclx_gettypes subroutine returns a value of 0. Otherwise, -1 is returned and the errno global variable is set to indicate the error.

Error Codes
The aclx_gettypes subroutine fails and the access control information for a file remains unchanged if one or more of the following is true:

EACCES
Search permission is denied on a component of the Path prefix.

EFAULT
The Path parameter points to a location outside of the allocated address space of the process.

ELOOP
Too many symbolic links were encountered in translating the Path parameter.

ENAMETOOLONG
A component of the Path parameter exceeded 255 characters, or the entire Path parameter exceeded 1023 characters.

ENOENT
A component of the Path does not exist or has the disallow truncation attribute (see the ulimit subroutine).

ENOENT
The Path parameter was null.

ENOENT
A symbolic link was named, but the file to which it refers does not exist.

ENOSPC
The acl_type_list buffer provided is not enough to store all the ACL types supported by this file system.

ENOTDIR
A component of the Path prefix is not a directory.

ESTALE
The process' root or current directory is located in a virtual file system that has been unmounted.

If Network File System (NFS) is installed on your system, the aclx_gettypes subroutine can also fail if the following condition is true:

ETIMEDOUT
The connection timed out.

Security
Access Control: Caller must have search permission for all components of the Path prefix.

Auditing Events: None

Related Information
The aclget command, aclput command.
aclx_print or aclx_printStr Subroutine

Purpose
Converts the binary access control information into nonbinary, readable format.

Library
Security Library (libc.a)

Syntax
#include <sys/acl.h>

int aclx_print (acl_file, acl, acl_sz, acl_type, fs_obj_path, flags)
FILE * acl_file;
void * acl;
size_t acl_sz;
acl_type_t acl_type;
char * fs_obj_path;
int32_t flags;

int aclx_printStr (str, str_sz, acl, acl_sz, acl_type, fs_obj_path, flags)
char * str;
size_t * str_sz;
void * acl;
size_t acl_sz;
acl_type_t acl_type;
char * fs_obj_path;
int32_t flags;

Description
The aclx_print and aclx_printStr subroutines print the access control information in a nonbinary, readable text format. These subroutines take the ACL information in binary format as input, convert it into text format, and print that text format output to either a file or a string. The aclx_print subroutine prints the ACL text to the file specified by acl_file. The aclx_printStr subroutine prints the ACL text to str. The amount of space available in str is specified in str_sz. If this memory is insufficient, the subroutine sets str_sz to the needed amount of memory and returns an ENOSPC error.

Parameters
- acl_file: Points to the file into which the textual output is printed.
- str: Points to the string into which the textual output should be printed.
- str_sz: Indicates the amount of memory in bytes available in str. If the text representation of acl requires more space than str_sz, this subroutine updates the str_sz with the amount of memory required and fails by setting errno to ENOSPC.
- acl: Points to a buffer which contains the binary ACL data that has to be printed. The size of this buffer is indicated by the acl_sz parameter.
- acl_sz: Indicates the size of the buffer area passed through the acl parameter.
- acl_type: Indicates the ACL type information of the acl. The ACL type is 64 bits in size and is unique on the system. If the given ACL type is not supported in the system, this function fails and errno is set to EINVAL.
fs_obj_path
File System Object Path for which the ACL data format and print are being requested.

Gets information about the object (such as whether the object is a file or directory, who
the owner is, and the associated group ID).

flags
Allows for control over the print operation. A value of ACL_VERBOSE indicates whether
additional information has to be printed in text format in comments. This bit is set when
the aclget command is issued with the -v (verbose) option.

Return Values
On successful completion, the aclx_print and aclx_printStr subroutines return a value of 0. Otherwise, -1
is returned and the errno global variable is set to indicate the error.

Error Codes
The aclx_print subroutine fails if one or more of the following is true:

Note: The errors in the following list occur only because aclx_print calls the fprintf subroutine internally. For more information about these errors, refer to the fprintf subroutine.

EAGAIN The O_NONBLOCK flag is set for the file descriptor underlying the file specified by the acl_file parameter, and the process would be delayed in the write operation.

EBADF The file descriptor underlying the file specified by the acl_file parameter is not a valid file descriptor open for writing.

EFBIG An attempt was made to write to a file that exceeds the file size limit of this process or the maximum file size. For more information, refer to the ulimit subroutine.

EINTR The write operation terminated because of a signal was received, and either no data
was transferred or a partial transfer was not reported.

EIO The process is a member of a background process group attempting to perform a
write to its controlling terminal, the TOSTOP flag is set, the process is neither ignoring
nor blocking the SIGTTOU signal, and the process group of the process has no
parent process.

ENOSPC No free space remains on the device that contains the file.

ENOSPC Insufficient storage space is available.

ENXIO A request was made of a nonexistent device, or the request was outside the
capabilities of the device.

EPIPE An attempt was made to write to a pipe or first-in-first-out (FIFO) that is not open for
reading by any process. A SIGPIPE signal is sent to the process.

The aclx_printStr subroutine fails if the following is true:

ENOSPC Input buffer size strSz is not sufficient to store the text representation of acl in str.

ENOSPC Insufficient storage space is available. This error is returned by sprintf, which is called by the
aclx_printStr subroutine internally.

The aclx_print or aclx_printStr subroutine fails if the following is true:

EINVAL Invalid input parameter. The same error can be returned if an invalid acl_type is specified as input
to this routine. This errno can also be returned if the acl is not of the type specified by acl_type.

Related Information
The "printf, fprintf, sprintf, snprintf, wsprintf, vprintf, vfprintf, vsprintf, or vwsprintf Subroutine" on page 1148,
"aclx_scan or aclx_scanStr Subroutine" on page 27.

The aclget command, aclput command.
aclx_put or aclx_fput Subroutine

Purpose
Stores the access control information for a file system object.

Library
Security Library (libc.a)

Syntax
```
#include <sys/acl.h>

int aclx_put (Path, ctl_flags, acl_type, acl, acl_sz, mode_info)
char *Path;
uint64_t ctl_flags;
acl_type_t acl_type;
void *acl;
size_t acl_sz;
mode_t mode_info;

int aclx_fput (FileDescriptor, ctl_flags, acl_type, acl, acl_sz, mode_info)
int FileDescriptor;
uint64_t ctl_flags;
acl_type_t acl_type;
void *acl;
size_t acl_sz;
mode_t mode_info;
```

Description
The aclx_put and aclx_fput subroutines store the access control information for a file system object in the native ACL format. Native ACL format is the format as defined for the particular ACL type in the system. These subroutines are advanced versions of the acl_put and acl_fput subroutines and should be used instead of the older versions. The aclx_put and aclx_fput subroutines provide for more control for the user to interact with the underlying file system directly.

A caller specifies the ACL type in the acl_type argument and passes the ACL information in the acl argument. The acl_sz parameter indicates the size of the ACL data. The ctl_flags parameter is a bitmask that allows for variation of aclx_put requests.

The value provided to these subroutines can be obtained by invoking aclx_get or aclx_fget subroutines to copy or restore the access control information.

The aclx_put and aclx_fput subroutines can also be used to manage the special bits (such as SGID and SUID) in the mode word associated with the file system object. For example, you can set the mode_info value to any special bit mask (as in the mode word defined for the file system), and a request can be made to set the same bits using the ctl_flags argument. Note that special privileges (such as root) might be required to set these bits.

Parameters

Path
Specifications the path name of the file system object.
Specifies the file descriptor of an open file system object. This 64-bit sized bit mask provides control over the ACL retrieval. These bits are divided as follows:

**Lower 16 bits**
- System-wide (nonphysical file-system-specific) ACL control flags

**32 bits**
- Reserved.

**Last 16 bits**
- Any physical file-system-defined options (that are specific to physical file system ACL implementation).

**ctl_flags**
- Bit mask with the following system-wide flag values defined:
  - **SET_MODE_S_BITS**
    - Indicates that the mode_info value is set by the caller and the ACL put operation needs to consider this value while completing the ACL put operation.
  - **SET_ACL**
    - Indicates that the acl argument points to valid ACL data that needs to be considered while the ACL put operation is being performed.

**Note:** Both of the preceding values can be specified by the caller by ORing the two masks.

**acl_type**
- Indicates the type of ACL to be associated with the file object. If the acl_type specified is not among the ACL types supported for the file system, then an error is returned.

**acl**
- Points to a buffer where the ACL information exists. This ACL information is associated with the file system object specified. The size of this buffer is indicated by the acl_sz parameter.

**acl_sz**
- Indicates the size of the ACL information sent through the acl parameter.

**mode_info**
- This value indicates any mode word information that needs to be set for the file system object in question as part of this ACL put operation. When mode bits are being altered by specifying the SET_MODE_S_BITS flag (in ctl_flags) ACL put operation fails if the caller does not have the required privileges.

### Return Values

On successful completion, the aclx_put and aclx_fput subroutines return a value of 0. Otherwise, -1 is returned and the errno global variable is set to indicate the error.

### Error Codes

The aclx_put subroutine fails and the access control information for a file remains unchanged if one or more of the following are true:

- **EACCES**
  - Search permission is denied on a component of the Path prefix.

- **EFAULT**
  - The Path parameter points to a location outside of the allocated address space of the process.

- **ELOOP**
  - Too many symbolic links were encountered in translating the Path parameter.

- **ENAMETOOLONG**
  - A component of the Path parameter exceeded 255 characters, or the entire Path parameter exceeded 1023 characters.

- **ENOENT**
  - A component of the Path does not exist or has the disallow truncation attribute (see the ulimit subroutine).

- **ENOENT**
  - The Path parameter was null.

- **ENOENT**
  - A symbolic link was named, but the file to which it refers does not exist.

- **ENOTDIR**
  - A component of the Path prefix is not a directory.

- **ESTALE**
  - The process' root or current directory is located in a virtual file system that has been unmounted.
The `aclx_fput` subroutine fails and the file permissions remain unchanged if the following is true:

**EBADF**

The `FileDescriptor` parameter is not a valid file descriptor.

The `aclx_put` or `aclx_fput` subroutine fails if one or more of the following is true:

**EINVAL** 
Invalid input parameter. The same error can be returned if an invalid `acl_type` is specified as input to this routine.

**EIO** 
An I/O error occurred during the operation.

**EROFS**
The named file resides on a read-only file system.

If Network File System (NFS) is installed on your system, the `acl_put` and `acl_fput` subroutines can also fail if the following condition is true:

**ETIMEDOUT**
The connection timed out.

**Security**

Access Control: The invoker must have search permission for all components of the `Path` prefix.

Auditing Events:

<table>
<thead>
<tr>
<th>Event</th>
<th>Information</th>
</tr>
</thead>
<tbody>
<tr>
<td>chacl</td>
<td><code>Path</code>-based event</td>
</tr>
<tr>
<td>fchacl</td>
<td><code>FileDescriptor</code>-based event</td>
</tr>
</tbody>
</table>

**Related Information**

The `acl_chg` ("acl_chg or acl_fchg Subroutine" on page 8) subroutine, `acl_get` ("acl_get or acl_fget Subroutine" on page 10) subroutine, `acl_set` ("acl_set or acl_fset Subroutine" on page 14) subroutine, `chacl` ("chacl or fchacl Subroutine" on page 144) subroutine, `chmod` ("chmod or fchmod Subroutine" on page 148) subroutine, `stat` subroutine, `statacl` subroutine.

The `aclget` command, `aclput` command, `chmod` command.

[List of Security and Auditing Subroutines and Subroutines Overview](#) in AIX 5L Version 5.3 General Programming Concepts: Writing and Debugging Programs.

### aclx_scan or aclx_scanStr Subroutine

**Purpose**

Reads the access control information that is in nonbinary, readable text format, and converts it into ACL type-specific native format binary ACL data.

**Library**

Security Library (`libc.a`)

**Syntax**

```c
#include <sys/acl.h>

int aclx_scan (acl_file, acl, acl_sz, acl_type, err_file)
FILE * acl_file;
void * acl;
```
size_t * acl_sz;
 acl_type_t acl_type;
 FILE * err_file;

int aclx_scanStr (str, acl, acl_sz, acl_type)
char * str;
void * acl;
size_t * acl_sz;
acl_type_t acl_type;

Description
The aclx_scan and aclx_scanStr subroutines read the access control information from the input given in nonbinary, readable text format and return a binary ACL data in the ACL type-specific native format. The aclx_scan subroutine provides the ACL data text in the file specified by acl_file. In the case of aclx_scanStr, the ACL data text is provided in the string pointed to by str. When the err_file parameter is not Null, it points to a file to which any error messages are written out by the aclx_scan subroutine in case of syntax errors in the input ACL data. The errors can occur if the syntax of the input text data does not adhere to the required ACL type-specific data specifications.

Parameters
- acl_file: Points to the file from which the ACL text output is read.
- str: Points to the string from which the ACL text output is printed.
- acl: Points to a buffer in which the binary ACL data has to be stored. The amount of memory available in this buffer is indicated by the acl_sz parameter.
- acl_sz: Indicates the amount of memory, in bytes, available in the acl parameter.
- acl_type: Indicates the ACL type information of the acl. The ACL type is 64 bits in size and is unique on the system. If the given ACL type is not supported in the system, this function fails and errno is set to EINVAL.
- err_file: File pointer to an error file. When this pointer is supplied, the subroutines write out any errors in the syntax/composition of the ACL input data.

Return Values
On successful completion, the aclx_scan and aclx_scanStr subroutines return a value of 0. Otherwise, -1 is returned and the errno global variable is set to indicate the error.

Error Codes
The aclx_scan subroutine fails if one or more of the following is true:

Note: The errors in the following list occur only because aclx_scan calls the fscanf subroutine internally. For more information about these errors, refer to the fscanf subroutine.

- EAGAIN: The O_NONBLOCK flag is set for the file descriptor underlying the file specified by the acl_file parameter, and the process would be delayed in the write operation.
- EBADF: The file descriptor underlying the file specified by the acl_file parameter is not a valid file descriptor open for writing.
- EINTR: The write operation terminated because of a signal was received, and either no data was transferred or a partial transfer was not reported.
- EIO: The process is a member of a background process group attempting to perform a write to its controlling terminal, the TOSTOP flag is set, the process is neither ignoring nor blocking the SIGTTOU signal, and the process group of the process has no parent process.
- ENOSPC: Insufficient storage space is available.
The `aclx_scanStr` subroutine fails if the following is true:

**ENOSPC** Insufficient storage space is available. This error is returned by `sscanf`, which is called by the `aclx_scanStr` subroutine internally.

The `aclx_scan` or `aclx_scanStr` subroutine fails if the following is true:

**EINVAL** Invalid input parameter. The same error can be returned if an invalid `acl_type` is specified as input to this routine. This `errno` can also be returned if the text ACL given in the input/file string is not of the type specified by `acl_type`.

**Related Information**
The “aclx_print or aclx_printStr Subroutine” on page 23, `fscanf` Subroutine.
The `aclget` command, `aclput` command.

List of Security and Auditing Subroutines and Subroutines Overview in AIX 5L Version 5.3 General Programming Concepts: Writing and Debugging Programs.

---

### acos, acosf, or acosl Subroutine

**Purpose**
Computes the inverse cosine of a given value.

**Syntax**
```c
#include <math.h>

float acosf (float x);  
long double acosl (long double x);  
double acosl (double x);  
```

**Description**
The `acosf`, `acosl`, and `acos` subroutines compute the principal value of the arc cosine of the `x` parameter. The value of `x` should be in the range [-1,1].

An application wishing to check for error situations should set the `errno` global variable to zero and call `fetestexcept(FE_ALL_EXCEPT)` before calling these functions. On return, if `errno` is nonzero or `fetestexcept(FE_INVALID | FE_DIVBYZERO | FE_OVERFLOW | FE_UNDERFLOW)` is nonzero, an error has occurred.

**Parameters**

`x` Specifies the value to be computed.
Return Values
Upon successful completion, these subroutines return the arc cosine of \( x \), in the range \([0, \pi]\) radians.

For finite values of \( x \) not in the range \([-1,1]\), a domain error occurs, and a NaN is returned.

If \( x \) is NaN, a NaN is returned.

If \( x \) is +1, 0 is returned.

If \( x \) is ±Inf, a domain error occurs, and a NaN is returned.

Related Information
The "acosh, acoshf, or acoshl Subroutine."

**acosh, acoshf, or acoshl Subroutine**

**Purpose**
Computes the inverse hyperbolic cosine.

**Syntax**
```c
#include <math.h>

float acoshf (f)
float x;

long double acoshl (x)
long double x;

double acosh (x)
double x;
```

**Description**
The `acoshf` or `acoshl` subroutine computes the inverse hyperbolic cosine of the \( x \) parameter.

The `acosh` subroutine returns the hyperbolic arc cosine specified by the \( x \) parameter, in the range 1 to the +HUGE_VAL value.

An application wishing to check for error situations should set `errno` to zero and call `fetestexcept(FE_ALL_EXCEPT)` before calling these subroutines. Upon return, if the `errno` global variable is nonzero or `fetestexcept(FE_INVALID | FE_DIVBYZERO | FE_OVERFLOW | FE_UNDERFLOW)` is nonzero, an error has occurred.

**Parameters**

\( x \) Specifies the value to be computed.

**Return Values**
Upon successful completion, the `acoshf`, or `acoshl` subroutine returns the inverse hyperbolic cosine of the given argument.

For finite values of \( x < 1 \), a domain error occurs, and a NaN is returned.
If \( x \) is NaN, a NaN is returned.

If \( x \) is +1, 0 is returned.

If \( x \) is +\( \infty \), +\( \infty \) is returned.

If \( x \) is −\( \infty \), a domain error occurs, and a NaN is returned.

**Error Codes**

The \texttt{acosh} subroutine returns \texttt{NaNQ} (not-a-number) and sets \texttt{errno} to \texttt{EDOM} if the \( x \) parameter is less than the value of 1.

**Related Information**

\texttt{math.h} in AIX 5L Version 5.3 Files Reference.

---

**addproj Subroutine**

**Purpose**

Adds an API-based project definition to the kernel project registry.

**Library**

The \texttt{libaacct.a} library.

**Syntax**

\texttt{<sys/aacct.h>}

\texttt{addproj(struct project *)}

**Description**

The \texttt{addproj} subroutine defines the application-based project definition to the kernel repository. An application can assign a project defined in this way using the \texttt{proj_execve} system call.

Projects that are added this way are marked as being specified by applications so that they do not overlap with system administrator-specified projects defined using the \texttt{projctl} command. The \texttt{PROJFLAG_API} flag is turned on in the structure \texttt{project} to indicate that the project definition was added by an application.

Projects added by a system administrator using the \texttt{projctl} command are flagged as being derived from the local or LDAP-based project repositories by the \texttt{PROJFLAGS_LDAP} or \texttt{PROJFLAGS_PDF} flag. If one of these flags is specified, the \texttt{addproj} subroutine fails with \texttt{EPERM}.

The \texttt{getproj} routine can be used to determine the origin of a loaded project.

The \texttt{addproj} validates the input project number to ensure that it is within the expected range of 0x00000001 - 0x00ffffff. It also validates that the project name is a POSIX compliant alphanumeric character string. If any invalid input is found \texttt{errno} will be set to \texttt{EINVAL} and the \texttt{addproj} subroutine returns -1.

**Parameters**

\texttt{project} Points to a project structure that holds the definition of the project to be added.
Security
Only for privileged users. Privilege can be extended to nonroot users by granting the CAP_AACCT capability to a user.

Return Values

<table>
<thead>
<tr>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Success</td>
</tr>
<tr>
<td>-1</td>
<td>Failure</td>
</tr>
</tbody>
</table>

Error Codes

<table>
<thead>
<tr>
<th>Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EINVAL</td>
<td>Invalid Project Name / Number or the passed pointer is NULL</td>
</tr>
<tr>
<td>EEXIST</td>
<td>Project Definition exists</td>
</tr>
<tr>
<td>EPERM</td>
<td>Permission Denied, not a privileged user</td>
</tr>
</tbody>
</table>

Related Information

The "addprojdb Subroutine," "chprojattr Subroutine" on page 158,"getproj Subroutine" on page 413,"getprojs Subroutine" on page 415,"rmproj Subroutine"

addprojdb Subroutine

Purpose
Adds a project definition to the specified project database.

Library
The libaacct.a library.

Syntax
<sys/aacct.h>
addprojdb(void *handle, struct project *project, char *comment)

Description
The addprojdb subroutine appends the project definition stored in the struct project variable into the project database named by the handle parameter. The project database must be initialized before calling this subroutine. The projdballoc subroutine is provided for this purpose. This routine verifies whether the supplied project definition already exists. If it does exist, the addprojdb subroutine sets errno to EEXIST and returns -1.

The addprojdb subroutine validates the input project number to ensure that it is within the expected range 0x00000001 - 0x00ffffff and validates that the project name is a POSIX-compliant alphanumeric character string. If any invalid input is found, the addprojdb subroutine sets errno to EINVAL and returns -1.

If the user does not have privilege to add an entry to project database, the addprojdb subroutine sets errno to EACCES and returns -1.

There is an internal state (that is, the current project) associated with the project database. When the project database is initialized, the current project is the first project in the database. The addprojdb subroutine appends the specified project to the end of the database. It advances the current project assignment to the next project in the database, which is the end of the project data base. At this point, a call to the getnextprojdb subroutine would fail, because there are no additional project definitions. To read...
the project definition that was just added, use the getprojdb subroutine. To read other projects, first call getfirstprojdb subroutine to reset the internal current project assignment so that subsequent reads can be performed.

The format of the records added to the project database are given as follows:

```
ProjectName:ProjectNumber:AggregationStatus:Comment::
```

Example:

```
Biology:4756:no:Project Created by projctl command::
```

**Parameters**

- **handle**: Pointer to project database handle
- **project**: Pointer to a project structure that holds the definition of the project to be added
- **comment**: Pointer to a character string that holds the comments about the project

**Security**

Only for privileged users. Privilege can be extended to nonroot users by granting the CAP_AACCT capability to a user.

**Return Values**

- **0**: Success
- **-1**: Failure

**Error Codes**

- **EINVAL**: Invalid project name or number, or the passed pointer is NULL.
- **EEXIST**: Project definition already exists.
- **EPERM**: Permission denied. The user is not a privileged user.

**Related Information**

The "addproj Subroutine" on page 31, "chprojattrdb Subroutine" on page 159, "getfirstprojdb Subroutine" on page 363, "getnextprojdb Subroutine" on page 391, "getprojdb Subroutine" on page 414, "projdballoc Subroutine" on page 1158, "projdbfinit Subroutine" on page 1159, "projdbfree Subroutine" on page 1160, "rmprojdb Subroutine".

**addssys Subroutine**

**Purpose**

Adds the **SRCsubsys** record to the subsystem object class.

**Library**

System Resource Controller Library (**libsri.a**)
int addssys (SRCSubsystem)
struct SRCsubsys *SRCSubsystem;

Description
The **addssys** subroutine adds a record to the subsystem object class. You must call the **defssys** subroutine to initialize the **SRCSubsystem** buffer before your application program uses the **SRCsubsys** structure. The **SRCsubsys** structure is defined in the */usr/include/sys/srcobj.h* file.

The executable running with this subroutine must be running with the group system.

Parameters

**SRCSubsystem**
A pointer to the **SRCsubsys** structure.

Return Values
Upon successful completion, the **addssys** subroutine returns a value of 0. Otherwise, it returns a value of -1 and the **odmerrno** variable is set to indicate the error, or an SRC error code is returned.

Error Codes
The **addssys** subroutine fails if one or more of the following are true:

- **SRC_BADFSIG**: Invalid stop force signal.
- **SRC_BADNSIG**: Invalid stop normal signal.
- **SRC_CMDARG2BIG**: Command arguments too long.
- **SRC_GRPNAME2BIG**: Group name too long.
- **SRC_NOCONTACT**: Contact not signal, sockets, or message queue.
- **SRC_NONAME**: No subsystem name specified.
- **SRC_NOPATH**: No subsystem path specified.
- **SRC_PATH2BIG**: Subsystem path too long.
- **SRC_STDERR2BIG**: stderr path too long.
- **SRC_STDIN2BIG**: stdin path too long.
- **SRCSTDOUT2BIG**: stdout path too long.
- **SRC_SUBEXIST**: New subsystem name already on file.
- **SRC_SUBSYS2BIG**: Subsystem name too long.
- **SRC_SYNAMEXIST**: New subsystem synonym name already on file.
- **SRC_SYNAME2BIG**: Synonym name too long.

Security
Privilege Control: This command has the Trusted Path attribute. It has the following kernel privilege:

SET_PROC_AUDIT
Files Accessed:

<table>
<thead>
<tr>
<th>Mode</th>
<th>File</th>
</tr>
</thead>
<tbody>
<tr>
<td>644</td>
<td>/etc/objrepos/SRCsubsys</td>
</tr>
</tbody>
</table>

Auditing Events:

If the auditing subsystem has been properly configured and is enabled, the **addssys** subroutine generates the following audit record (event) each time the subroutine is executed:
Event Information

SRC_addssys  Lists the SRCsubsys records added.

See “Setting Up Auditing” in Security for details about selecting and grouping audit events, and configuring audit event data collection.

Files

/etc/objrepos/SRCsubsys  SRC Subsystem Configuration object class.
/dev/SRC  Specifies the AF_UNIX socket file.
/dev/SRC-unix  Specifies the location for temporary socket files.
/usr/include/spc.h  Defines external interfaces provided by the SRC subroutines.
/usr/include/sys/srcobj.h  Defines object structures used by the SRC.

Related Information

The chssys (“chssys Subroutine” on page 162) subroutine, defssys (“defssys Subroutine” on page 211) subroutine, delssys (“delssys Subroutine” on page 211) subroutine.

The auditpr command, chssys command, mkssys command, rmssys command.

Auditing Overview (“audit Subroutine” on page 98) and System Resource Controller in Operating system and device management.

Defining Your Subsystem to the SRC System Resource Controller (SRC) Overview for Programmers in AIX 5L Version 5.3 General Programming Concepts: Writing and Debugging Programs.

List of SRC Subroutines in AIX 5L Version 5.3 General Programming Concepts: Writing and Debugging Programs.

adjtime Subroutine

Purpose

Corrects the time to allow synchronization of the system clock.

Library

Standard C Library (libc.a)

Syntax

```c
#include <sys/time.h>
int adjtime ([Delta], [Olddelta])
struct timeval *Delta;
struct timeval *Olddelta;
```

Description

The adjtime subroutine makes small adjustments to the system time, as returned by the gettimeofday subroutine, advancing or retarding it by the time specified by the Delta parameter of the timeval structure. If the Delta parameter is negative, the clock is slowed down by periodically subtracting a small amount from it until the correction is complete. If the Delta parameter is positive, a small amount is periodically added to the clock until the correction is complete. The skew used to perform the correction is generally ten percent. If the clock is sampled frequently enough, an application program can see time apparently jump backwards. For information on a way to avoid this, see gettimeofday, settimeofday, or ftime.
A time correction from an earlier call to the `adjtime` subroutine may not be finished when the `adjtime` subroutine is called again. If the `Olddelta` parameter is nonzero, then the structure pointed to will contain, upon return, the number of microseconds still to be corrected from the earlier call.

This call may be used by time servers that synchronize the clocks of computers in a local area network. Such time servers would slow down the clocks of some machines and speed up the clocks of others to bring them to the average network time.

The `adjtime` subroutine is restricted to the users with root user authority.

### Parameters

- **Delta**: Specifies the amount of time to be altered.
- **Olddelta**: Contains the number of microseconds still to be corrected from an earlier call.

### Return Values

A return value of 0 indicates that the `adjtime` subroutine succeeded. A return value of -1 indicates than an error occurred, and `errno` is set to indicate the error.

### Error Codes

The `adjtime` subroutine fails if the following are true:

- **EFAULT**: An argument address referenced invalid memory.
- **EPERM**: The process’s effective user ID does not have root user authority.

### agg_proc_stat, agg_lpar_stat, agg_arm_stat, or free_agg_list Subroutine

#### Purpose

Aggregate advanced accounting data.

#### Library

The `libaacct.a` library.

#### Syntax

```c
#define <sys/aacct.h>
int agg_arm_stat(tran_list, arm_list);
struct aacct_tran_rec *tran_list
struct agg_arm_stat **arm_list
int agg_proc_stat(sortcrit1, sortcrit2, sortcrit3, sortcrit4, tran_list, proc_list);
int sortcrit1, sortcrit2, sortcrit3, sortcrit4
struct aacct_tran_rec *tran_list
struct agg_proc_stat **proc_list
int agg_lpar_stat(l_type, *tran_list, l_list);
int l_type
struct aacct_tran_rec *tran_list
union agg_lpar_rec *l_list
void free_agg_list(list);
void *list
```

Description
The `agg_proc_stat`, `agg_lpar_stat`, and `agg_arm_stat` subroutines return a linked list of aggregated transaction records for process, LPAR, and ARM, respectively.

The `agg_proc_stat` subroutine performs the process record aggregation based on the criterion values passed as input parameters. The aggregated process transaction records are sorted based on the sorting criteria values `sortcrit1`, `sortcrit2`, `sortcrit3`, and `sortcrit4`. These four can be one of the following values defined in the `sys/aacct.h` file:

- `CRIT_UID`
- `CRIT_GID`
- `CRIT_PROJ`
- `CRIT_CMD`
- `CRIT_NONE`

The order of their usage determines the sorting order applied to the retrieved aggregated list of process transaction records. For example, the sort criteria values of `PROJ_GID`, `PROJ_PROJ`, `PROJ_UID`, `PROJ_NONE` first sorts the aggregated list on group IDs, which are further sorted based on project IDs, followed by another level of sorting based on user IDs.

Some of the process transaction records (of type `TRID_agg_proc`) cannot be aggregated based on group IDs and command names. For such records, `agg_proc_stat` returns an asterisk (*) character as the command name and a value of -2 as the group ID. This indicates to the caller that these records cannot be aggregated.

If the aggregation is not necessary on a specific criteria, `agg_proc_stat` returns a value of -1 in the respective field. For example, if the aggregation is not necessary on the group ID (`CRIT_GID`), the retrieved list of aggregation records has a value of -1 filled in the group ID fields.

The `agg_lpar_stat` retrieves an aggregated list of LPAR transaction records. Because there are several types of LPAR transaction records, the caller must specify the type of LPAR transaction record that is to be aggregated. The transaction record type can be one of the following values, defined in the `sys/aacct.h` file:

- `AGG_CPUMEM`
- `AGG_FILESYS`
- `AGG_NETIF`
- `AGG_DISK`
- `AGG_VTARGET`
- `AGG_VCLIENT`

The `agg_lpar_stat` subroutine uses a union argument of type `struct agg_lpar_rec`. For this argument, the caller must provide the address of the linked list to which the aggregated records should be returned.

The `agg_arm_list` retrieves an aggregated list of ARM transaction records from the list of transaction records provided as input. The aggregated transaction records are returned to the caller through the structure pointer of type `struct agg_arm_stat`.

The `free_agg_list` subroutine frees the memory allocated to the aggregated records returned by the `agg_proc_stat`, `agg_lpar_stat`, or `agg_arm_stat` subroutine.

Parameters

`arm_list`  
Pointer to the linked list of `struct agg_arm_stat` nodes to be returned.
l_list
Pointer to union agg_lpar_rec address to which the aggregated LPAR records are returned.

l_type
Integer value that represents the type of LPAR resource to be aggregated.

list
Pointer to the aggregated list to be freed.

proc_list
Pointer to the linked list of struct agg_proc_stat nodes to be returned.

sortcrit1, sortcrit2, sortcrit3, sortcrit4
Integer values that represent the sorting criteria to be passed to agg_proc_stat.

call_list
Pointer to the input list of transaction records

Security
No restrictions. Any user can call this function.

Return Values

0 The call to the subroutine was successful.
-1 The call to the subroutine failed.

Error Codes

EINVAL The passed pointer is NULL.
ENOMEM Insufficient memory.
EPERM Permission denied. Unable to read the data file.

Related Information

The "buildproclist Subroutine" on page 125, "buildtranlist or freetranlist Subroutine" on page 126, "getproclist, getlparlist, or getarmlist Subroutine" on page 409.

Understanding the Advanced Accounting Subsystem

aio_cancel or aio_cancel64 Subroutine

The aio_cancel or aio_cancel64 subroutine includes information for the POSIX AIO aio_cancel subroutine (as defined in the IEEE std 1003.1-2001), and the Legacy AIO aio_cancel subroutine.

POSIX AIO aio_cancel Subroutine

Purpose
Cancels one or more outstanding asynchronous I/O requests.

Library
Standard C Library (libc.a)

Syntax

```c
#include <aio.h>

int aio_cancel (fildes, aiocbp)  
int fildes;  
struct aiocb *aiocbp;
```

38 Technical Reference, Volume 1: Base Operating System and Extensions
**Description**

The `aio_cancel` subroutine cancels one or more asynchronous I/O requests currently outstanding against the `fildes` parameter. The `aiocbp` parameter points to the asynchronous I/O control block for a particular request to be canceled. If `aiocbp` is NULL, all outstanding cancelable asynchronous I/O requests against `fildes` are canceled.

Normal asynchronous notification occurs for asynchronous I/O operations that are successfully canceled. If there are requests that cannot be canceled, the normal asynchronous completion process takes place for those requests when they are completed.

For requested operations that are successfully canceled, the associated error status is set to `ECANCELED`, and a -1 is returned. For requested operations that are not successfully canceled, the `aiocbp` parameter is not modified by the `aio_cancel` subroutine.

If `aiocbp` is not NULL, and if `fildes` does not have the same value as the file descriptor with which the asynchronous operation was initiated, unspecified results occur.

The implementation of the subroutine defines which operations are cancelable.

**Parameters**

- **fildes** Identifies the object to which the outstanding asynchronous I/O requests were originally queued.
- **aiocbp** Points to the `aiocb` structure associated with the I/O operation.

**aiocb Structure**

The `aiocb` structure is defined in the `/usr/include/aio.h` file and contains the following members:

```c
int aio_fildes
off_t aio_offset
char *aio_buf
size_t aio_nbytes
int aio_reqprio
struct sigevent aio_sigevent
int aio_lio_opcode
```

**Execution Environment**

The `aio_cancel` and `aio_cancel64` subroutines can be called from the process environment only.

**Return Values**

The `aio_cancel` subroutine returns `AIO_CANCELED` to the calling process if the requested operation(s) were canceled. `AIO_NOTCANCELED` is returned if at least one of the requested operations cannot be canceled because it is in progress. In this case, the state of the other operations, referenced in the call to `aio_cancel` is not indicated by the return value of `aio_cancel`. The application may determine the state of affairs for these operations by using the `aio_error` subroutine. `AIO_ALLDONE` is returned if all of the operations are completed. Otherwise, the subroutine returns -1 and sets the `errno` global variable to indicate the error.

**Error Codes**

- **EBADF** The `fildes` parameter is not a valid file descriptor.

**Related Information**

- [“aio_error or aio_error64 Subroutine” on page 42](#)
- [“aio_nwait Subroutine” on page 47](#)
- [“aio_nwait_timeout Subroutine” on page 49](#)
- [“aio_read or aio_read64 Subroutine” on page 50](#)
- [“aio_return or aio_return64 Subroutine” on page 50](#)
Legacy AIO aio_cancel Subroutine

Purpose
Cancels one or more outstanding asynchronous I/O requests.

Library
Standard C Library (libc.a)

Syntax
#include <aio.h>

aio_cancel (int FileDescriptor, struct aiocb *aiocbp)

aio_cancel64 (int FileDescriptor, struct aiocb64 *aiocbp)

Description
The aio_cancel subroutine attempts to cancel one or more outstanding asynchronous I/O requests issued on the file associated with the FileDescriptor parameter. If the pointer to the aio control block (aiocb) structure (the aiocbp parameter) is not null, then an attempt is made to cancel the I/O request associated with this aiocb. The aiocbp parameter used by the thread calling aix_cancel must have had its request initiated by this same thread. Otherwise, a -1 is returned and errno is set to EINVAL. However, if the aiocbp parameter is null, then an attempt is made to cancel all outstanding asynchronous I/O requests associated with the FileDescriptor parameter without regard to the initiating thread.

The aio_cancel64 subroutine is similar to the aio_cancel subroutine except that it attempts to cancel outstanding large file enabled asynchronous I/O requests. Large file enabled asynchronous I/O requests make use of the aiocb64 structure instead of the aiocb structure. The aiocb64 structure allows asynchronous I/O requests to specify offsets in excess of OFF_MAX (2 gigabytes minus 1).

In the large file enabled programming environment, aio_cancel is redefined to be aio_cancel64.

When an I/O request is canceled, the aio_error subroutine called with the handle to the corresponding aiocb structure returns ECANCELED.

Note: The _AIO_AIX_SOURCE macro used in aio.h must be defined when using aio.h to compile an aio application with the Legacy AIO function definitions. The default compile using the aio.h file is for an application with the POSIX AIO definitions. In the source file enter:

#define _AIO_AIX_SOURCE
#include <sys/aio.h>
or, on the command line when compiling enter:
->xlc ... -D_AIO_AIX_SOURCE ... legacy_aio_program.c

Parameters

**FileDescriptor** Identifies the object to which the outstanding asynchronous I/O requests were originally queued.

**aiocbp** Points to the **aiocb** structure associated with the I/O operation.

**aiocb Structure**
The **aiocb** structure is defined in the `/usr/include/aio.h` file and contains the following members:

```
struct aiocb {
    int aio_whence;
    off_t aio_offset;
    char *aio_buf;
    ssize_t aio_return;
    int aio_errno;
    size_t aio_nbytes;
    union {
        int reqprio;
        struct {
            int version:8;
            int priority:8;
            int cache_hint:16;
        } ext;
    } aio_u1;
    int aio_flag;
    int aio_iocpfd;
    aio_handle_t aio_handle;
}
```

#define aio_reqprio aio_u1.reqprio
#define aio_version aio_u1.ext.version
#define aio_priority aio_u1.ext.priority
#define aio_cache_hint aio_u1.ext.cache_hint

Execution Environment

The **aio_cancel** and **aio_cancel64** subroutines can be called from the process environment only.

Return Values

**AIO_CANCELED** Indicates that all of the asynchronous I/O requests were canceled successfully. The **aio_error** subroutine call with the handle to the **aiocb** structure of the request will return **ECANCELED**.

**AIO_NOTCANCELED** Indicates that the **aio_cancel** subroutine did not cancel one or more outstanding I/O requests. This may happen if an I/O request is already in progress. The corresponding error status of the I/O request is not modified.

**AIO_ALLDONE** Indicates that none of the I/O requests is in the queue or in progress.

-1 Indicates that the subroutine was not successful. Sets the **errno** global variable to identify the error.

A return code can be set to the following **errno** value:

**EBADF** Indicates that the **FileDescriptor** parameter is not valid.
Related Information
"aio_error or aio_error64 Subroutine," "aio_nwait Subroutine" on page 47, "aio_nwait_timeout Subroutine" on page 49, "aio_read or aio_read64 Subroutine" on page 50, "aio_return or aio_return64 Subroutine" on page 55, "aio_suspend or aio_suspend64 Subroutine" on page 58, and "aio_write or aio_write64 Subroutine" on page 61, "lio_listio or lio_listio64 Subroutine" on page 713.


The Input and Output Handling in AIX 5L Version 5.3 General Programming Concepts: Writing and Debugging Programs describes the files, commands, and subroutines used for low-level, stream, terminal, and asynchronous I/O interfaces.

aio_error or aio_error64 Subroutine

The aio_error or aio_error64 subroutine includes information for the POSIX AIO aio_error subroutine (as defined in the IEEE std 1003.1-2001), and the Legacy AIO aio_error subroutine.

POSIX AIO aio_error Subroutine

Purpose
Retrieves error status for an asynchronous I/O operation.

Library
Standard C Library (libc.a)

Syntax
#include <aio.h>

int aio_error (aiocbp);
const struct aiocb *aiocbp;

Description
The aio_error subroutine returns the error status associated with the aiocb structure. This structure is referenced by the aiocbp parameter. The error status for an asynchronous I/O operation is the synchronous I/O errno value that would be set by the corresponding read, write, or fsync subroutine. If the subroutine has not yet completed, the error status is equal to EINPROGRESS.

Parameters
aiocbp Points to the aiocb structure associated with the I/O operation.

aiocb Structure
The aiocb structure is defined in the /usr/include/aio.h file and contains the following members:

int aio_fildes
off_t aio_offset
char *aio_buf
size_t aio_nbytes
int aio_reqprio
struct sigevent aio_sigevent
int aio_lio_opcode

42 Technical Reference, Volume 1: Base Operating System and Extensions
Execution Environment
The **aio_error** and **aio_error64** subroutines can be called from the process environment only.

Return Values
If the asynchronous I/O operation has completed successfully, the **aio_error** subroutine returns a 0. If unsuccessful, the error status (as described for the **read**, **write**, and **fsync** subroutines) is returned. If the asynchronous I/O operation has not yet completed, **EINPROGRESS** is returned.

Error Codes
**EINVAL**  The **aiocbp** parameter does not refer to an asynchronous operation whose return status has not yet been retrieved.

Related Information
“**aio_cancel or aio_cancel64 Subroutine**” on page 38, “**aio_fsync Subroutine**” on page 45, “**aio_nwait Subroutine**” on page 47, “**aio_nwait_timeout Subroutine**” on page 49, “**aio_read or aio_read64 Subroutine**” on page 50, “**aio_return or aio_return64 Subroutine**” on page 55, “**aio_write or aio_write64 Subroutine**” on page 61, “**close Subroutine**” on page 175, “**exec, execl, execle, execvp, execv, execvp, or execvpe Subroutine**” on page 235, “**exit, atexit, unatexit, _exit, or _Exit Subroutine**” on page 242, “**fork, f_fork, or vfork Subroutine**” on page 287, “**fsync or fsync_range Subroutine**” on page 317, “**lio_listio or lio_listio64 Subroutine**” on page 713, and “**lseek, llseek or lseek64 Subroutine**” on page 756.

Legacy AIO aio_error Subroutine

Purpose
Retrieves the error status of an asynchronous I/O request.

Library
Standard C Library (**libc.a**)

Syntax
```
#include <aio.h>

int aio_error(handle);
handle_t handle;

int aio_error64(handle);
handle_t handle;
```

Description
The **aio_error** subroutine retrieves the error status of the asynchronous request associated with the **handle** parameter. The error status is the **errno** value that would be set by the corresponding I/O operation. The error status is **EINPROG** if the I/O operation is still in progress.

The **aio_error64** subroutine is similar to the **aio_error** subroutine except that it retrieves the error status associated with an **aiocb64** control block.
Note: The _AIO_AIX_SOURCE macro used in aio.h must be defined when using aio.h to compile an aio application with the Legacy AIO function definitions. The default compile using the aio.h file is for an application with the POSIX AIO definitions. In the source file enter:

```c
#define _AIO_AIX_SOURCE
#include <sys/aio.h>
```
or, on the command line when compiling enter:
```
->xlc ... -D_AIO_AIX_SOURCE ... legacy_aio_program.c
```

Parameters

**handle** The handle field of an aio control block (aiocb or aiocb64) structure set by a previous call of the aio_read, aio_read64, aio_write, aio_write64, lio_listio, lio_listio64 subroutine. If a random memory location is passed in, random results are returned.

aiocb Structure

The aiocb structure is defined in the /usr/include/aio.h file and contains the following members:

```c
struct aiocb {
    int aio_whence;
    off_t aio_offset;
    char *aio_buf;
    ssize_t aio_return;
    int aio_errno;
    size_t aio_nbytes;
    union {
        int reqprio;
        struct {
            int version:8;
            int priority:8;
            int cache_hint:16;
        } ext;
    } aio_u1;
    int aio_flag;
    int aio_iocpfd;
    aio_handle_t aio_handle;
}
```

```c
#define aio_reqprio aio_u1.reqprio
#define aio_version aio_u1.ext.version
#define aio_priority aio_u1.ext.priority
#define aio_cache_hint aio_u1.ext.cache_hint
```

Execution Environment

The aio_error and aio_error64 subroutines can be called from the process environment only.

Return Values

0 Indicates that the operation completed successfully.

ECANCELED Indicates that the I/O request was canceled due to an aio_cancel subroutine call.
Indicates that the I/O request has not completed.

An **errno** value described in the `aio_read` (**aio_read or aio_read64 Subroutine** on page 50), `aio_write` (**aio_write or aio_write64 Subroutine** on page 61), and `lio_listio` (**lio_listio or lio_listio64 Subroutine** on page 713) subroutines: Indicates that the operation was not queued successfully. For example, if the `aio_read` subroutine is called with an unusable file descriptor, it (**aio_read**) returns a value of -1 and sets the **errno** global variable to **EBADF**. A subsequent call of the `aio_error` subroutine with the handle of the unsuccessful **aio control block** (**aiocb**) structure returns **EBADF**.

An **errno** value of the corresponding I/O operation: Indicates that the operation was initiated successfully, but the actual I/O operation was unsuccessful. For example, calling the `aio_write` subroutine on a file located in a full file system returns a value of 0, which indicates the request was queued successfully. However, when the I/O operation is complete (that is, when the `aio_error` subroutine no longer returns **EINPROG**), the `aio_error` subroutine returns **ENOSPC**. This indicates that the I/O was unsuccessful.

**Related Information**

"**aio_cancel or aio_cancel64 Subroutine** on page 38," **aio_read or aio_read64 Subroutine** on page 50, "**aio_nwait Subroutine** on page 47," **aio_nwait_timeout Subroutine** on page 49, "**aio_return or aio_return64 Subroutine** on page 55," **aio_suspend or aio_suspend64 Subroutine** on page 58, "**aio_write or aio_write64 Subroutine** on page 61," **lio_listio or lio_listio64 Subroutine** on page 713, and **lio_listio or lio_listio64 Subroutine** on page 713.

The **Asynchronous I/O Overview** and the **Communications I/O Subsystem: Programming Introduction** in AIX 5L Version 5.3 Kernel Extensions and Device Support Programming Concepts.

The **Input and Output Handling Programmer’s Overview** in AIX 5L Version 5.3 General Programming Concepts: Writing and Debugging Programs describes the files, commands, and subroutines used for low-level, stream, terminal, and asynchronous I/O interfaces.

### **aio_fsasync Subroutine**

**Purpose**

Synchronizes asynchronous files.

**Library**

Standard C Library (**libc.a**)

**Syntax**

```c
#include <aio.h>

int aio_fsync(int op, struct aiocb *aiocbp);
```

**Description**

The **aio_fsasync** subroutine asynchronously forces all I/O operations to the synchronized I/O completion state. The function call returns when the synchronization request has been initiated or queued to the file or device (even when the data cannot be synchronized immediately).

If the **op** parameter is set to O_DSYNC, all currently queued I/O operations are completed as if by a call to the **fdatasync** subroutine. If the **op** parameter is set to O_SYNC, all currently queued I/O operations are
completed as if by a call to the `fsync` subroutine. If the `aio_fsync` subroutine fails, or if the operation queued by `aio_fsync` fails, outstanding I/O operations are not guaranteed to be completed.

If `aio_fsync` succeeds, it is only the I/O that was queued at the time of the call to `aio_fsync` that is guaranteed to be forced to the relevant completion state. The completion of subsequent I/O on the file descriptor is not guaranteed to be completed in a synchronized fashion.

The `aiocbp` parameter refers to an asynchronous I/O control block. The `aiocbp` value can be used as an argument to the `aio_error` and `aio_return` subroutines in order to determine the error status and return status, respectively, of the asynchronous operation while it is proceeding. When the request is queued, the error status for the operation is `EINPROGRESS`. When all data has been successfully transferred, the error status is reset to reflect the success or failure of the operation. If the operation does not complete successfully, the error status for the operation is set to indicate the error. The `aio_sigevent` member determines the asynchronous notification to occur when all operations have achieved synchronized I/O completion. All other members of the structure referenced by the `aiocbp` parameter are ignored. If the control block referenced by `aiocbp` becomes an illegal address prior to asynchronous I/O completion, the behavior is undefined.

If the `aio_fsync` subroutine fails or `aiocbp` indicates an error condition, data is not guaranteed to have been successfully transferred.

**Parameters**

- `op` Determines the way all currently queued I/O operations are completed.
- `aiocbp` Points to the `aiocb` structure associated with the I/O operation.

**aiocb Structure**

The `aiocb` structure is defined in the `/usr/include/aio.h` file and contains the following members:

- `aio_fildes`
- `aio_offset`
- `*aio_buf`
- `aio_nbytes`
- `aio_reqprio`
- `aio_lio_opcode`
- `aio_sigevent`
- `aio_error`
- `aio_return`

**Execution Environment**

The `aio_error` and `aio_error64` subroutines can be called from the process environment only.

**Return Values**

The `aio_fsync` subroutine returns a 0 to the calling process if the I/O operation is successfully queued. Otherwise, it returns a -1, and sets the `errno` global variable to indicate the error.

**Error Codes**

- `EAGAIN` The requested asynchronous operation was not queued due to temporary resource limitations.
- `EBADF` The `aio_fildes` member of the `aiocb` structure referenced by the `aiocbp` parameter is not a valid file descriptor open for writing.

In the event that any of the queued I/O operations fail, the `aio_fsync` subroutine returns the error condition defined for the `read` and `write` subroutines. The error is returned in the error status for the asynchronous `fsync` subroutine, which can be retrieved using the `aio_error` subroutine.
Related Information

“fcntl, dup, or dup2 Subroutine” on page 254, “fsync or fsync_range Subroutine” on page 317, and “open, openx, open64, creat, or creat64 Subroutine” on page 925.

read, readx, readv, or pread Subroutine and write, writex, writev, writevx or pwrite Subroutines in AIX 5L Version 5.3 Technical Reference: Base Operating System and Extensions Volume 2.

aio_nwait Subroutine

Purpose
Suspends the calling process until a certain number of asynchronous I/O requests are completed.

Library
Standard C Library (libc.a)

Syntax
```c
#include <aio.h>

int aio_nwait (cnt, nwait, list);

int cnt;
int nwait;
struct aiocb **list;
```

Description
Although the `aio_nwait` subroutine is included with POSIX AIO, it is not part of the standard definitions for POSIX AIO.

The `aio_nwait` subroutine suspends the calling process until a certain number (`nwait`) of asynchronous I/O requests are completed. These requests are initiated at an earlier time by the `lio_listio` subroutine, which uses the LIO_NOWAIT_AIOWAIT `cmd` parameter. The `aio_nwait` subroutine fills in the `aiocb` pointers to the completed requests in `list` and returns the number of valid entries in `list`. The `cnt` parameter is the maximum number of `aiocb` pointers that `list` can hold (`cnt >= nwait`). The subroutine also returns when less than `nwait` number of requests are done if there are no more pending aio requests.

Note: If the `lio_listio64` subroutine is used, the `aiocb` structure changes to `aiocb64`.

Note: The aio control block’s `errno` field continues to have the value EINPROG until after the `aio_nwait` subroutine is completed. The `aio_nwait` subroutine updates this field when the `lio_listio` subroutine has run with the LIO_NOWAIT_AIOWAIT `cmd` parameter. No utility, such as `aio_error`, can be used to look at this value until after the `aio_nwait` subroutine is completed.

The `aio_suspend` subroutine returns after any one of the specified requests gets done. The `aio_nwait` subroutine returns after a certain number (`nwait` or more) of requests are completed.

There are certain limitations associated with the `aio_nwait` subroutine, and a comparison between it and the `aio_suspend` subroutine is necessary. The following table is a comparison of the two subroutines:

<table>
<thead>
<tr>
<th><code>aio_suspend</code></th>
<th><code>aio_nwait</code></th>
</tr>
</thead>
<tbody>
<tr>
<td>Requires users to build a list of control blocks, each associated with an I/O operation they want to wait for.</td>
<td>Requires the user to provide an array to put <code>aiocb</code> address information into. No specific aio control blocks need to be known.</td>
</tr>
<tr>
<td>Returns when any one of the specified control blocks indicates that the I/O associated with that control block completed.</td>
<td>Returns when <code>nwait</code> amount of requests are done or no other requests are to be processed.</td>
</tr>
</tbody>
</table>
aio_suspend:
The aio control blocks may be updated before the subroutine is called. Other polling methods (such as the aio_error subroutine) can also be used to view the aio control blocks.

aio_nwait:
Updates the aio control blocks itself when it is called. Other polling methods can't be used until after the aio_nwait subroutine is called enough times to cover all of the aio requests specified with the lio_listio subroutine. Is only used in accordance with the LIO_NOWAIT_AIOWAIT command, which is one of the commands associated with the lio_listio subroutine. If the lio_listio subroutine is not first used with the LIO_NOWAIT_AIOWAIT command, aio_nwait can not be called. The aio_nwait subroutine only affects those requests called by one or more lio_listio calls for a specified process.

Parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>cnt</td>
<td>Specifies the number of entries in the list array.</td>
</tr>
<tr>
<td>nwait</td>
<td>Specifies the minimal number of requests to wait on.</td>
</tr>
<tr>
<td>list</td>
<td>An array of pointers to aio control structures defined in the aio.h file.</td>
</tr>
</tbody>
</table>

Return Values

The return value is the total number of requests the aio_nwait subroutine has waited on to complete. It can not be more than cnt. Although nwait is the desired amount of requests to find, the actual amount returned could be less than, equal to, or greater than nwait. The return value indicates how much of the list array to access.

The return value may be greater than the nwait value if the lio_listio subroutine initiated more than nwait requests and the cnt variable is larger than nwait. The nwait parameter represents a minimal value desired for the return value, and cnt is the maximum value possible for the return.

The return value may be less than the nwait value if some of the requests initiated by the lio_listio subroutine occur at a time of high activity, and there is a lack of resources available for the number of requests. EAGAIN (error try again later) may be returned in some request's aio control blocks, but these requests will not be seen by the aio_nwait subroutine. In this situation aiocb addresses not found on the list have to be accessed by using the aio_error subroutine after the aio_nwait subroutine is called. You may need to increase the aio parameters max servers or max requests if this occurs. Increasing the parameters will ensure that the system is well tuned, and an EAGAIN error is less likely to occur.

In the event of an error, the aio_nwait subroutine returns a value of -1 and sets the errno global variable to identify the error. Return codes can be set to the following errno values:

EBUSY  An aio_nwait call is in process.
EINVAL  The application has retrieved all of the aiocb pointers, but the user buffer does not have enough space for them.
EINVAL  There are no outstanding async I/O calls.

Related Information

- "aio_cancel or aio_cancel64 Subroutine" on page 38
- "aio_error or aio_error64 Subroutine" on page 42
- "aio_nwait timeout Subroutine" on page 49
- "aio_read or aio_read64 Subroutine" on page 50
- "aio_return or aio_return64 Subroutine" on page 55
- "aio_suspend or aio_suspend64 Subroutine" on page 58
- "aio_write or aio_write64 Subroutine" on page 61
- "lio_listio or lio_listio64 Subroutine" on page 713

The Input and Output Handling Programmer's Overview in AIX 5L Version 5.3 General Programming Concepts: Writing and Debugging Programs describes the files, commands, and subroutines used for low-level, stream, terminal, and asynchronous I/O interfaces.

**aio_nwait_timeout Subroutine**

**Purpose**
Extends the capabilities of the **aio_nwait** subroutine by specifying timeout values.

**Library**
Standard C library (**libc.a**).

**Syntax**

```c
int aio_nwait_timeout (cnt, nwait, list, timeout)
int cnt;
int nwait;
struct aiocbp **list;
int timeout;
```

**Description**

The **aio_nwait_timeout** subroutine waits for a certain number of asynchronous I/O operations to complete as specified by the **nwait** parameter, or until the call has blocked for a certain duration specified by the **timeout** parameter.

**Parameters**

- **cnt** indicates the maximum number of pointers to the **aiocbp** structure that can be copied into the **list** array.
- **list** is an array of pointers to aio control structures defined in the **aio.h** file.
- **nwait** specifies the number of asynchronous I/O operations that must complete before the **aio_nwait_timeout** subroutine returns.
- **timeout** is specified in units of milliseconds.

A **timeout** value of -1 indicates that the subroutine behaves like the **aio_nwait** subroutine, blocking until all of the requested I/O operations complete or until there are no more asynchronous I/O requests pending from the process.

A **timeout** value of 0 indicates that the subroutine returns immediately with the current completed number of asynchronous I/O requests. All other positive **timeout** values indicate that the subroutine must block until either the **timeout** value is reached or the requested number of asynchronous I/O operations complete.

**Return Values**

The return value is the total number of requests the **aio_nwait** subroutine has waited on to complete. It can not be more than **cnt**. Although **nwait** is the desired amount of requests to find, the actual amount returned could be less than, equal to, or greater than **nwait**. The return value indicates how much of the list array to access.

The return value may be greater than the **nwait** value if the **lio_listio** subroutine initiated more than **nwait** requests and the **cnt** variable is larger than **nwait**. The **nwait** parameter represents a minimal value desired for the return value, and **cnt** is the maximum value possible for the return.

The return value may be less than the **nwait** value if some of the requests initiated by the **lio_listio** subroutine occur at a time of high activity, and there is a lack of resources available for the number of requests. The **EAGAIN** return code (error try again later) might be returned in some request’s aio control
blocks, but these requests will not be seen by the **aio_nwait** subroutine. In this situation, the **aiocb** structure addresses that are not found on the list must be accessed using the **aio_error** subroutine after the **aio_nwait** subroutine is called. You might need to increase the aio parameters max servers or max requests if this occurs. Increasing the parameters will ensure that the system is well tuned, and an **EAGAIN** error is less likely to occur. The return value might be less than the **nwait** value due to the setting of the new timeout parameter in the following cases:

- **timeout** > 0 and a timeout has occurred before **nwait** requests are done
- **timeout** = 0 and the current requests completed at the time of the **aio_nwait_timeout** call are less then **nwait** parameter

In the event of an error, the **aio_nwait** subroutine returns a value of -1 and sets the **errno** global variable to identify the error. Return codes can be set to the following **errno** values:

- EBUSY: An **aio_nwait** call is in process.
- EINVAL: The application has retrieved all of the **aiocb** pointers, but the user buffer does not have enough space for them.
- EINVAL: There are no outstanding async I/O calls.

### Related Information

- "**aio_nwait Subroutine**" on page 47, "**aio_suspend or aio_suspend64 Subroutine**" on page 58, "**aio_cancel or aio_cancel64 Subroutine**" on page 38, "**aio_error or aio_error64 Subroutine**" on page 42, "**aio_read or aio_read64 Subroutine**", "**aio_return or aio_return64 Subroutine**" on page 55, "**aio_write or aio_write64 Subroutine**" on page 61, and "**lio_listio or lio_listio64 Subroutine**" on page 713.


The Input and Output Handling Programmer’s Overview in AIX 5L Version 5.3 General Programming Concepts: Writing and Debugging Programs describes the files, commands, and subroutines used for low-level, stream, terminal, and asynchronous I/O interfaces.

---

**aio_read or aio_read64 Subroutine**

The **aio_read** or **aio_read64** subroutine includes information for the POSIX AIO **aio_read** subroutine (as defined in the IEEE std 1003.1-2001), and the Legacy AIO **aio_read** subroutine.

### POSIX AIO aio_read Subroutine

#### Purpose

Asynchronously reads a file.

#### Library

Standard C Library (**libc.a**)

#### Syntax

```c
#include <aio.h>

int aio_read (aiocbp);
struct aiocb *aiocbp;
```
Description
The **aio_read** subroutine reads `aio_nbytes` from the file associated with `aio_fildes` into the buffer pointed to by `aio_buf`. The subroutine returns when the read request has been initiated or queued to the file or device (even when the data cannot be delivered immediately).

The `aiocbp` value may be used as an argument to the **aio_error** and **aio_return** subroutines in order to determine the error status and return status, respectively, of the asynchronous operation while it is proceeding. If an error condition is encountered during queuing, the function call returns without having initiated or queued the request. The requested operation takes place at the absolute position in the file as given by `aio_offset`, as if the `lseek` subroutine were called immediately prior to the operation with an offset equal to `aio_offset` and a whence equal to SEEK_SET. After a successful call to enqueue an asynchronous I/O operation, the value of the file offset for the file is unspecified.

The `aio_lio_opcode` field is ignored by the **aio_read** subroutine.

If prioritized I/O is supported for this file, the asynchronous operation is submitted at a priority equal to the scheduling priority of the process minus `aiocbp->aio_reqprio`.

The `aiocbp` parameter points to an **aiocb** structure. If the buffer pointed to by `aio_buf` or the control block pointed to by `aiocbp` becomes an illegal address prior to asynchronous I/O completion, the behavior is undefined.

Simultaneous asynchronous operations using the same `aiocbp` produce undefined results.

If synchronized I/O is enabled on the file associated with `aio_fildes`, the behavior of this subroutine is according to the definitions of synchronized I/O data integrity completion and synchronized I/O file integrity completion.

For any system action that changes the process memory space while an asynchronous I/O is outstanding, the result of that action is undefined.

For regular files, no data transfer occurs past the offset maximum established in the open file description.

If you use the **aio_read** or **aio_read64** subroutine with a file descriptor obtained from a call to the **shm_open** subroutine, it will fail with `EINVAL`.

Parameters
```c
aiocbp    Points to the **aiocb** structure associated with the I/O operation.
```

**aiocb Structure**
The **aiocb** structure is defined in the `/usr/include/aio.h` file and contains the following members:

```c
    int    aio_fildes
    off_t  aio_offset
    char   *aio_buf
    size_t aio_nbytes
    int    aio_reqprio
    struct sigevent  aio_sigevent
    int    aio_lio_opcode
```

**Execution Environment**
The **aio_read** and **aio_read64** subroutines can be called from the **process environment** only.
Return Values

The `aio_read` subroutine returns 0 to the calling process if the I/O operation is successfully queued. Otherwise, it returns a -1 and sets the `errno` global variable to indicate the error.

Error Codes

**EAGAIN**  
The requested asynchronous I/O operation was not queued due to system resource limitations.

Each of the following conditions may be detected synchronously at the time of the call to the `aio_read` subroutine, or asynchronously. If any of the conditions below are detected synchronously, the `aio_read` subroutine returns -1 and sets the `errno` global variable to the corresponding value. If any of the conditions below are detected asynchronously, the return status of the asynchronous operation is set to -1, and the error status of the asynchronous operation is set to the corresponding value.

**EBADF**  
The `aio_fildes` parameter is not a valid file descriptor open for reading.

**EINVAL**  
The file offset value implied by `aio_offset` is invalid, `aio_reqprio` is an invalid value, or `aio_nbytes` is an invalid value. The `aio_read` or `aio_read64` subroutine was used with a file descriptor obtained from a call to the `shm_open` subroutine.

If the `aio_read` subroutine successfully queues the I/O operation but the operation is subsequently canceled or encounters an error, the return status of the asynchronous operation is one of the values normally returned by the `read` subroutine. In addition, the error status of the asynchronous operation is set to one of the error statuses normally set by the `read` subroutine, or one of the following values:

**EBADF**  
The `aio_fildes` argument is not a valid file descriptor open for reading.

**ECANCELED**  
The requested I/O was canceled before the I/O completed due to an explicit `aio_cancel` request.

**EINVAL**  
The file offset value implied by `aio_offset` is invalid.

The following condition may be detected synchronously or asynchronously:

**EOVERFLOW**  
The file is a regular file, `aio_nbytes` is greater than 0, and the starting offset in `aio_offset` is before the end-of-file and is at or beyond the offset maximum in the open file description associated with `aio_fildes`.

Related Information

- `aio_cancel` or `aio_cancel64 Subroutine` on page 38
- `aio_error or aio_error64 Subroutine` on page 42
- `aio_nwait` Subroutine” on page 47
- `aio_nwait_timeout Subroutine” on page 49
- `lio_listio or lio_listio64 Subroutine` on page 713
- `aio_return` or `aio_return64 Subroutine’ on page 55
- `aio_suspend` or `aio_suspend64 Subroutine’ on page 58
- `aio_write or aio_write64 Subroutine’ on page 61
- `close Subroutine” on page 175
- `exec, :execl, execlp, execv, execvp, or execlp Subroutine” on page 235
- `exit, atexit, unatexit, _exit, or _Exit Subroutine” on page 242
- `fork, f_fork, or vfork Subroutine” on page 287
- `lseek, lseek, or lseek64 Subroutine” on page 756

The `read, readx, readv, readvx, or pread Subroutine` in *AIX 5L Version 5.3 Technical Reference: Base Operating System and Extensions Volume 2.*

Legacy AIO `aio_read Subroutine`

**Purpose**

Reads asynchronously from a file.
Library
Standard C Library (libc.a)

Syntax
#include <aio.h>

int aio_read(int FileDescriptor, struct aiocb *aiocbp);

int FileDescriptor;
struct aiocb *aiocbp;

int aio_read64(int FileDescriptor, struct aiocb64 *aiocbp);

int FileDescriptor;
struct aiocb64 *aiocbp;

Description
The aio_read subroutine reads asynchronously from a file. Specifically, the aio_read subroutine reads from the file associated with the FileDescriptor parameter into a buffer.

The aio_read64 subroutine is similar to the aio_read subroutine except that it takes an aiocb64 reference parameter. This allows the aio_read64 subroutine to specify offsets in excess of OFF_MAX (2 gigabytes minus 1).

In the large file enabled programming environment, aio_read is redefined to be aio_read64.

If you use the aio_read or aio_read64 subroutine with a file descriptor obtained from a call to the shm_open subroutine, it will fail with EINVAL.

The details of the read are provided by information in the aiocb structure, which is pointed to by the aiocbp parameter. This information includes the following fields:

aio_buf Indicates the buffer to use.
aio_nbytes Indicates the number of bytes to read.

When the read request has been queued, the aio_read subroutine updates the file pointer specified by the aio_whence and aio_offset fields in the aiocb structure as if the requested I/O were already completed. It then returns to the calling program. The aio_whence and aio_offset fields have the same meaning as the whence and offset parameters in the lseek (“lseek, llseek or lseek64 Subroutine” on page 756) subroutine. The subroutine ignores them for file objects that are not capable of seeking.

If an error occurs during the call, the read request is not queued. To determine the status of a request, use the aio_error (“aio_error or aio_error64 Subroutine” on page 42) subroutine.

To have the calling process receive the SIGIO signal when the I/O operation completes, set the AIO_SIGNAL bit in the aio_flag field in the aiocb structure.

Note: The event structure in the aiocb structure is currently not in use but is included for future compatibility.

Note: The _AIO_AIX_SOURCE macro used in aio.h must be defined when using aio.h to compile an aio application with the Legacy AIO function definitions. The default compile using the aio.h file is for an application with the POSIX AIO definitions. In the source file enter:
#define _AIO_AIX_SOURCE
#include <sys/aio.h>
or, on the command line when compiling enter:

```
->xlc ... -D_AIO_AIX_SOURCE ... legacy_aio_program.c
```

Since prioritized I/O is not supported at this time, the `aio_reqprio` field of the structure is not presently used.

**Parameters**

- **FileDescriptor** Identifies the object to be read as returned from a call to open.
- **aiocbp** Points to the asynchronous I/O control block structure associated with the I/O operation.

**aiocb Structure**

The `aio_cb` and the `aiocb64` structures are defined in the `aio.h` file and contain the following members:

```c
struct aiocb
{
    int aio_whence;
    off_t aio_offset;
    char *aio_buf;
    ssize_t aio_return;
    int aio_errno;
    size_t aio_nbytes;
    union {
        int reqprio;
        struct {
            int version:8;
            int priority:8;
            int cache_hint:16;
        } ext;
    } aio_u1;
    int aio_flag;
    int aio_lrench;
    aiohandle_t aio_handle;
}
```

```
define aio_reqprio    aio_u1.reqprio
#define aio_version     aio_u1.ext.version
#define aio_priority    aio_u1.ext.priority
#define aio_cache_hint  aio_u1.ext.cache_hint
```

**Execution Environment**

The `aio_read` and `aio_read64` subroutines can be called from the process environment only.

**Return Values**

When the read request queues successfully, the `aio_read` subroutine returns a value of 0. Otherwise, it returns a value of -1 and sets the global variable `errno` to identify the error.

Return codes can be set to the following `errno` values:

- **EAGAIN** Indicates that the system resources required to queue the request are not available. Specifically, the transmit queue may be full, or the maximum number of opens may be reached.
- **EBADF** Indicates that the `FileDescriptor` parameter is not valid.
- **EFAULT** Indicates that the address specified by the `aiocbp` parameter is not valid.
- **EINVAL** Indicates that the `aio_whence` field does not have a valid value, or that the resulting pointer is not valid. The `aio_read` or `aio_read64` subroutine was used with a file descriptor obtained from a call to the `shm_open` subroutine.
When using I/O Completion Ports with AIO Requests, return codes can also be set to the following errno values:

- **EBADF** Indicates that the aio_iocpfdo field in the aiocb structure is not a valid I/O Completion Port file descriptor.
- **EINVAL** Indicates that an I/O Completion Port service failed when attempting to start the AIO Request.
- **EPERM** Indicates that I/O Completion Port services are not available.

**Note:** Other error codes defined in the `sys/errno.h` file can be returned by the `aio_error` subroutine if an error during the I/O operation is encountered.

**Related Information**

- "aio_cancel or aio_cancel64 Subroutine” on page 38.
- "aio_nwait Subroutine” on page 47.
- "aio_nwait_timeout Subroutine” on page 49.
- "aio_error or aio_error64 Subroutine” on page 42.
- "aio_return or aio_return64 Subroutine” on page 58.
- "aio_write or aio_write64 Subroutine” on page 61.
- "lio_listio or lio_listio64 Subroutine” on page 713.


The Input and Output Handling Programmer’s Overview in AIX 5L Version 5.3 General Programming Concepts: Writing and Debugging Programs describes the files, commands, and subroutines used for low-level, stream, terminal, and asynchronous I/O interfaces.

---

**aio_return or aio_return64 Subroutine**

The `aio_return` subroutine includes information for the POSIX AIO `aio_return` subroutine (as defined in the IEEE std 1003.1-2001), and the Legacy AIO `aio_return` subroutine.

**POSIX AIO aio_return Subroutine**

**Purpose**

Retrieves the return status of an asynchronous I/O operation.

**Library**

Standard C Library (`libc.a`)

**Syntax**

```c
#include <aio.h>

size_t aio_return (aiocbp);
struct aiocb *aiocbp;
```

**Description**

The `aio_return` subroutine returns the return status associated with the `aiocb` structure. The return status for an asynchronous I/O operation is the value that would be returned by the corresponding `read`, `write`, or `fsync` subroutine call. If the error status for the operation is equal to **EINPROGRESS**, the return status for the operation is undefined. The `aio_return` subroutine can be called once to retrieve the return status of a given asynchronous operation. After that, if the same `aiocb` structure is used in a call to `aio_return` or `aio_error`, an error may be returned. When the `aiocb` structure referred to by `aiocbp` is used to submit another asynchronous operation, the `aio_return` subroutine can be successfully used to retrieve the return status of that operation.
Parameters

aiocbp

Points to the aiocb structure associated with the I/O operation.

aiocb Structure

The aiocb structure is defined in the /usr/include/aio.h file and contains the following members:

- int aio_fildes
- off_t aio_offset
- char *aio_buf
- size_t aio_nbytes
- int aio_reqprio
- struct sigevent aio_sigevent
- int aio_lio_opcode

Execution Environment

The aio_return and aio_return64 subroutines can be called from the process environment only.

Return Values

If the asynchronous I/O operation has completed, the return status (as described for the read, write, and fsync subroutines) is returned. If the asynchronous I/O operation has not yet completed, the results of the aio_return subroutine are undefined.

Error Codes

EINVAL

The aiocbp parameter does not refer to an asynchronous operation whose return status has not yet been retrieved.

Related Information

*aio_cancel or aio_cancel64 Subroutine* on page 38, *aio_error or aio_error64 Subroutine* on page 42, *aio_nwait Subroutine* on page 47, *aio_nwait_timeout Subroutine* on page 49, *aio_read or aio_read64 Subroutine* on page 50, *aio_suspend or aio_suspend64 Subroutine* on page 58, *aio_write or aio_write64 Subroutine* on page 61, *close Subroutine* on page 175, *exec: exec, execl, exect, execvp, execv, execve, execvp, or exec Subroutine* on page 235, *exit, atexit, unatexit, exit, or Exit Subroutine* on page 242, *fork, _fork, or vfork Subroutine* on page 287, *lio_listio or lio_listio64 Subroutine* on page 713, and *lseek, llseek or lseek64 Subroutine* on page 756.


Legacy AIO aio_return Subroutine

Purpose

Retrieves the return status of an asynchronous I/O request.

Library

Standard C Library (libc.a)

Syntax

```c
#include <aio.h>

int aio_return(int handle);
aio_handle_t handle;
```
int aio_return64( handle)
    aio_handle_t handle;

Description
The aio_return subroutine retrieves the return status of the asynchronous I/O request associated with the aio_handle_t handle if the I/O request has completed. The status returned is the same as the status that would be returned by the corresponding read or write function calls. If the I/O operation has not completed, the returned status is undefined.

The aio_return64 subroutine is similar to the aio_return subroutine except that it retrieves the error status associated with an aiocb64 control block.

Note: The _AIO_AIX_SOURCE macro used in aio.h must be defined when using aio.h to compile an aio application with the Legacy AIO function definitions. The default compile using the aio.h file is for an application with the POSIX AIO definitions. In the source file enter:

```c
#define _AIO_AIX_SOURCE
#include <sys/aio.h>
```

or, on the command line when compiling enter:

```bash
->xlc ... -D_AIO_AIX_SOURCE ... legacy_aio_program.c
```

Parameters
handle The handle field of an aio control block (aiocb or aiocb64) structure is set by a previous call of the aio_read, aio_read64, aio_write, aio_write64, lio_listio, aio_listio64 subroutine. If a random memory location is passed in, random results are returned.

aiocb Structure
The aiocb structure is defined in the /usr/include/aio.h file and contains the following members:

```c
struct aiocb {
    int aio_whence;
    off_t aio_offset;
    char *aio_buf;
    ssize_t aio_return;
    int aio_errno;
    size_t aio_nbytes;
    union {
        int reqprio;
        struct {
            int version:8;
            int priority:8;
            int cache_hint:16;
        } ext;
    } aio_u1;
    int aio_flag;
    int aio_iocpfid;
    aio_handle_t aio_handle;
}
```

```c
#define aio_reqprio aio_u1.reqprio
#define aio_version aio_u1.ext.version
#define aio_priority aio_u1.ext.priority
#define aio_cache_hint aio_u1.ext.cache_hint
```

Execution Environment
The aio_return and aio_return64 subroutines can be called from the process environment only.
Return Values

The `aio_return` subroutine returns the status of an asynchronous I/O request corresponding to those returned by `read` or `write` functions. If the error status returned by the `aio_error` subroutine call is `EINPROG`, the value returned by the `aio_return` subroutine is undefined.

Examples

An `aio_read` request to read 1000 bytes from a disk device eventually, when the `aio_error` subroutine returns a 0, causes the `aio_return` subroutine to return 1000. An `aio_read` request to read 1000 bytes from a 500 byte file eventually causes the `aio_return` subroutine to return 500. An `aio_write` request to write to a read-only file system results in the `aio_error` subroutine eventually returning `ERofs` and the `aio_return` subroutine returning a value of -1.

Related Information

- "aio_cancel or aio_cancel64 Subroutine" on page 38
- "aio_error or aio_error64 Subroutine" on page 42
- "aio_nwait Subroutine" on page 47
- "aio_nwait_timeout Subroutine" on page 49
- "aio_read or aio_read64 Subroutine" on page 50
- "aio_suspend or aio_suspend64 Subroutine" on page 61
- "aio_write or aio_write64 Subroutine" on page 62
- "close Subroutine" on page 175
- "exec: execl, execle, execlp, execv, execve, execvp, or exect Subroutine" on page 235
- "fork, _fork, or vfork Subroutine" on page 287
- "lseek, llseek or lseek64 Subroutine" on page 756


The Input and Output Handling Programmer’s Overview in AIX 5L Version 5.3 General Programming Concepts: Writing and Debugging Programs describes the files, commands, and subroutines used for low-level, stream, terminal, and asynchronous I/O interfaces.

### aio_suspend or aio_suspend64 Subroutine

The `aio_suspend` subroutine includes information for the POSIX AIO `aio_suspend` subroutine (as defined in the IEEE std 1003.1-2001), and the Legacy AIO `aio_suspend` subroutine.

**POSIX AIO aio_suspend Subroutine**

**Purpose**

Waits for an asynchronous I/O request.

**Library**

Standard C Library (`libc.a`)

**Syntax**

```c
#include <aio.h>

int aio_suspend (list, nent, timeout);
const struct aiocb * const list[];
int nent;
const struct timespec *timeout;
```

**Description**

The `aio_suspend` subroutine suspends the calling thread until at least one of the asynchronous I/O operations referenced by the `list` parameter has completed, until a signal interrupts the function, or, if
timeout is not NULL, until the time interval specified by timeout has passed. If any of the aiocb structures in the list correspond to completed asynchronous I/O operations (the error status for the operation is not equal to EINPROGRESS) at the time of the call, the subroutine returns without suspending the calling thread. The list parameter is an array of pointers to asynchronous I/O control blocks. The nent parameter indicates the number of elements in the array. Each aiocb structure pointed to has been used in initiating an asynchronous I/O request through the aio_read, aio_write, or lio_listio subroutine. This array may contain NULL pointers, which are ignored. If this array contains pointers that refer to aiocb structures that have not been used in submitting asynchronous I/O, the effect is undefined.

If the time interval indicated in the timespec structure pointed to by timeout passes before any of the I/O operations referenced by list are completed, the aio_suspend subroutine returns with an error. If the Monotonic Clock option is supported, the clock that is used to measure this time interval is the CLOCK_MONOTONIC clock.

Parameters

- **list** Array of asynchronous I/O operations.
- **nent** Indicates the number of elements in the list array.
- **timeout** Specifies the time the subroutine has to complete the operation.

Execution Environment

The aio_suspend and aio_suspend64 subroutines can be called from the process environment only.

Return Values

If the aio_suspend subroutine returns after one or more asynchronous I/O operations have completed, it returns a 0. Otherwise, it returns a -1 and sets the errno global variable to indicate the error.

The application can determine which asynchronous I/O completed by scanning the associated error and returning status using the aio_error and aio_return subroutines, respectively.

Error Codes

- **EAGAIN** No asynchronous I/O indicated in the list referenced by list completed in the time interval indicated by timeout.
- **EINTR** A signal interrupted the aio_suspend subroutine. Since each asynchronous I/O operation may possibly provoke a signal when it completes, this error return may be caused by the completion of one (or more) of the very I/O operations being awaited.

Related Information

- “aio_cancel or aio_cancel64 Subroutine” on page 38
- “aio_error or aio_error64 Subroutine” on page 42
- “aio_nwait Subroutine” on page 47
- “aio_nwait_timeout Subroutine” on page 49
- “aio_read or aio_read64 Subroutine” on page 50
- “aio_return or aio_return64 Subroutine” on page 55
- “aio_write or aio_write64 Subroutine” on page 61
- “lio_listio or lio_listio64 Subroutine” on page 713

Legacy AIO aio_suspend Subroutine

Purpose

Suspends the calling process until one or more asynchronous I/O requests is completed.

Library

Standard C Library (libc.a)
Syntax

```c
#include <aio.h>

aio_suspend(count, aiocbpa)
int count;
struct aiocb *aiocbpa[];

aio_suspend64(count, aiocbpa)
int count;
struct aiocb64 *aiocbpa[];
```

**Description**

The **aio_suspend** subroutine suspends the calling process until one or more of the `count` parameter asynchronous I/O requests are completed or a signal interrupts the subroutine. Specifically, the **aio_suspend** subroutine handles requests associated with the **aio control block (aiocb)** structures pointed to by the `aiocbpa` parameter.

The **aio_suspend64** subroutine is similar to the **aio_suspend** subroutine except that it takes an array of pointers to **aiocb64** structures. This allows the **aio_suspend64** subroutine to suspend on asynchronous I/O requests submitted by either the **aio_read64**, **aio_write64**, or the **lio_listio64** subroutines.

In the large file enabled programming environment, **aio_suspend** is redefined to be **aio_suspend64**.

The array of **aiocb** pointers may include null pointers, which will be ignored. If one of the I/O requests is already completed at the time of the **aio_suspend** call, the call immediately returns.

**Note:** The `_AIO_AIX_SOURCE` macro used in **aio.h** must be defined when using **aio.h** to compile an aio application with the Legacy AIO function definitions. The default compile using the **aio.h** file is for an application with the POSIX AIO definitions. In the source file enter:

```c
#define _AIO_AIX_SOURCE
#include <sys/aio.h>
```

or, on the command line when compiling enter:

```
xlc ... -D_AIO_AIX_SOURCE ... legacy_aio_program.c
```

**Parameters**

- `count` Specifies the number of entries in the `aiocbpa` array.
- `aiocbpa` Points to the **aiocb** or **aiocb64** structures associated with the asynchronous I/O operations.

**aiocb Structure**

The **aiocb** structure is defined in the `/usr/include/aio.h` file and contains the following members:

```c
struct aiocb
{
    int aio_whence;
    off_t aio_offset;
    char *aio_buf;
    ssize_t aio_return;
    int aio_errno;
    size_t aio_nbytes;
    union {
        int reqprio;
        struct {
            int version:8;
            int priority:8;
            int cache_hint:16;
        };
    }
}
```
Execution Environment

The `aio_suspend` and `aio_suspend64` subroutines can be called from the process environment only.

Return Values

If one or more of the I/O requests completes, the `aio_suspend` subroutine returns the index into the `aiocbpa` array of one of the completed requests. The index of the first element in the `aiocbpa` array is 0. If more than one request has completed, the return value can be the index of any of the completed requests.

In the event of an error, the `aio_suspend` subroutine returns a value of -1 and sets the `errno` global variable to identify the error. Return codes can be set to the following `errno` values:

- EINTR Indicates that a signal or event interrupted the `aio_suspend` subroutine call.
- EINVAL Indicates that the `aio_whence` field does not have a valid value or that the resulting pointer is not valid.

Related Information

- "aio_cancel or aio_cancel64 Subroutine" on page 38
- "aio_error or aio_error64 Subroutine" on page 42
- "aio_nwait Subroutine" on page 47
- "aio_nwait_timeout Subroutine" on page 49
- "aio_read or aio_read64 Subroutine" on page 50
- "aio_return or aio_return64 Subroutine" on page 55
- "aio_write or aio_write64 Subroutine," and "lio_listio or lio_listio64 Subroutine" on page 713.


The Input and Output Handling Programmer's Overview in AIX 5L Version 5.3 General Programming Concepts: Writing and Debugging Programs describes the files, commands, and subroutines used for low-level, stream, terminal, and asynchronous I/O interfaces.

aio_write or aio_write64 Subroutine

The `aio_write` subroutine includes information for the POSIX AIO `aio_write` subroutine (as defined in the IEEE std 1003.1-2001), and the Legacy AIO `aio_write` subroutine.

POSIX AIO `aio_write` Subroutine

Purpose

Asynchronously writes to a file.

Library

Standard C Library (libc.a)
Syntax

```c
#include <aio.h>

int aio_write (aiocbp);
struct aiocb *aiocbp;
```

Description

The `aio_write` subroutine writes `aio_nbytes` to the file associated with `aio_fildes` from the buffer pointed to by `aio_buf`. The subroutine returns when the write request has been initiated or queued to the file or device.

The `aiocbp` parameter may be used as an argument to the `aio_error` and `aio_return` subroutines in order to determine the error status and return status, respectively, of the asynchronous operation while it is proceeding.

The `aiocbp` parameter points to an `aiocb` structure. If the buffer pointed to by `aio_buf` or the control block pointed to by `aiocbp` becomes an illegal address prior to asynchronous I/O completion, the behavior is undefined.

If O_APPEND is not set for the `aio_fildes` file descriptor, the requested operation takes place at the absolute position in the file as given by `aio_offset`. This is done as if the `lseek` subroutine were called immediately prior to the operation with an offset equal to `aio_offset` and a whence equal to SEEK_SET. If O_APPEND is set for the file descriptor, write operations append to the file in the same order as the calls were made. After a successful call to enqueue an asynchronous I/O operation, the value of the file offset for the file is unspecified.

The `aio_lio_opcode` field is ignored by the `aio_write` subroutine.

If prioritized I/O is supported for this file, the asynchronous operation is submitted at a priority equal to the scheduling priority of the process minus `aiocbp->aio_reqprio`.

Simultaneous asynchronous operations using the same `aiocbp` produce undefined results.

If synchronized I/O is enabled on the file associated with `aio_fildes`, the behavior of this subroutine is according to the definitions of synchronized I/O data integrity completion, and synchronized I/O file integrity completion.

For any system action that changes the process memory space while an asynchronous I/O is outstanding, the result of that action is undefined.

For regular files, no data transfer occurs past the offset maximum established in the open file description associated with `aio_fildes`.

If you use the `aio_write` or `aio_write64` subroutine with a file descriptor obtained from a call to the `shm_open` subroutine, it will fail with `EINVAL`.

Parameters

- `aiocbp` Points to the `aiocb` structure associated with the I/O operation.

aiocb Structure

The `aiocb` structure is defined in the `/usr/include/aio.h` file and contains the following members:

```c
int aio_fildes
off_t aio_offset
char *aio_buf
```
size_t aio_nbytes
int aio_reqprio
struct sigevent aio_sigevent
int aio_lio_opcode

Execution Environment
The aio_write and aio_write64 subroutines can be called from the process environment only.

Return Values
The aio_write subroutine returns a 0 to the calling process if the I/O operation is successfully queued. Otherwise, a -1 is returned and the errno global variable is set to indicate the error.

Error Codes
EAGAIN The requested asynchronous I/O operation was not queued due to system resource limitations.

Each of the following conditions may be detected synchronously at the time of the call to aio_write, or asynchronously. If any of the conditions below are detected synchronously, the aio_write subroutine returns a -1 and sets the errno global variable to the corresponding value. If any of the conditions below are detected asynchronously, the return status of the asynchronous operation is set to -1, and the error status of the asynchronous operation is set to the corresponding value.

EBADF The aio_fildes parameter is not a valid file descriptor open for writing.
EINVAL The file offset value implied by aio_offset is invalid, aio_reqprio is an invalid value, or aio_nbytes is an invalid value. The aio_write or aio_write64 subroutine was used with a file descriptor obtained from a call to the shm_open subroutine.

If the aio_write subroutine successfully queues the I/O operation, the return status of the asynchronous operation is one of the values normally returned by the write subroutine call. If the operation is successfully queued but is subsequently canceled or encounters an error, the error status for the asynchronous operation contains one of the values normally set by the write subroutine call, or one of the following:

EBADF The aio_fildes parameter is not a valid file descriptor open for writing.
EINVAL The file offset value implied by aio_offset would be invalid.
ECANCELED The requested I/O was canceled before the I/O completed due to an aio_cancel request.

The following condition may be detected synchronously or asynchronously:

EFBIG The file is a regular file, aio_nbytes is greater than 0, and the starting offset in aio_offset is at or beyond the offset maximum in the open file description associated with aio_fildes.

Related Information
"aio_cancel or aio-cancel64 Subroutine" on page 38, "aio_error or aio-error64 Subroutine" on page 42, "aio_nwait Subroutine" on page 47, "aio_nwait_timeout Subroutine" on page 49, "lio_listio or lio_listio64 Subroutine" on page 713, "aio_read or aio_read64 Subroutine" on page 50, "aio_suspend or aio_suspend64 Subroutine" on page 58, "aio_return or aio_return64 Subroutine" on page 55, "close Subroutine" on page 175, "exec: execl, execlp, execv, execve, execvpe, or execl Subroutine" on page 235, "exit, atexit, unatexit, exit, or Exit Subroutine" on page 242, "fork, f_fork, or vfork Subroutine" on page 287, and "lseek, llseek or lseek64 Subroutine" on page 756.

Legacy AIO aio_write Subroutine

Purpose
Writes to a file asynchronously.

Library
Standard C Library (libc.a)

Syntax
#include <aio.h>

int aio_write( FileDescriptor, aiocbp)
int FileDescriptor;
struct aiocb *aiocbp;

int aio_write64( FileDescriptor, aiocbp)
int FileDescriptor;
struct aiocb64 *aiocbp;

Description
The aio_write subroutine writes asynchronously to a file. Specifically, the aio_write subroutine writes to the file associated with the FileDescriptor parameter from a buffer. To handle this, the subroutine uses information from the aio control block (aiocb) structure, which is pointed to by the aiocbp parameter. This information includes the following fields:

- **aio_buf**: Indicates the buffer to use.
- **aio_nbytes**: Indicates the number of bytes to write.

The aio_write64 subroutine is similar to the aio_write subroutine except that it takes an aiocb64 reference parameter. This allows the aio_write64 subroutine to specify offsets in excess of OFF_MAX (2 gigabytes minus 1).

In the large file enabled programming environment, aio_read is redefined to be aio_read64.

If you use the aio_write or aio_write64 subroutine with a file descriptor obtained from a call to the shm_open subroutine, it will fail with EINVAL.

When the write request has been queued, the aio_write subroutine updates the file pointer specified by the aio_whence and aio_offset fields in the aiocb structure as if the requested I/O completed. It then returns to the calling program. The aio_whence and aio_offset fields have the same meaning as the whence and offset parameters in the lseek subroutine. The subroutine ignores them for file objects that are not capable of seeking.

If an error occurs during the call, the write request is not initiated or queued. To determine the status of a request, use the aio_error subroutine.

To have the calling process receive the SIGIO signal when the I/O operation completes, set the AIO_SIGNAL bit in the aio_flag field in the aiocb structure.

Note: The event structure in the aiocb structure is currently not in use but is included for future compatibility.
Note: The _AIO_AIX_SOURCE macro used in aio.h must be defined when using aio.h to compile an aio application with the Legacy AIO function definitions. The default compile using the aio.h file is for an application with the POSIX AIO definitions. In the source file enter:

```c
#define _AIO_AIX_SOURCE
#include <sys/aio.h>
```

or, on the command line when compiling enter:
```
xlc ... -D_AIO_AIX_SOURCE ... legacy_aio_program.c
```

Since prioritized I/O is not supported at this time, the aio_reqprio field of the structure is not presently used.

Parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>FileDescriptor</td>
<td>Identifies the object to be written as returned from a call to open.</td>
</tr>
<tr>
<td>aiocbp</td>
<td>Points to the asynchronous I/O control block structure associated with the I/O operation.</td>
</tr>
</tbody>
</table>

aiocb Structure

The aiocb structure is defined in the /usr/include/aio.h file and contains the following members:

```c
struct aiocb {
    int   aio_whence;
    off_t aio_offset;
    char  *aio_buf;
    ssize_t aio_return;
    int   aio_errno;
    size_t aio_nbytes;
    union {
        int        reqprio;
        struct {
            int    version:8;
            int    priority:8;
            int    cache_hint:16;
        } ext;
    } aio_u1;
    int    aio_flag;
    int    aio_iocpfd;
    aio_handle_t aio_handle;
}
```

# define aio_reqprio  aio_u1.reqprio
# define aio_version  aio_u1.ext.version
# define aio_priority aio_u1.ext.priority
# define aio_cache_hint aio_u1.ext.cache_hint

Execution Environment

The aio_write and aio_write64 subroutines can be called from the process environment only.

Return Values

When the write request queues successfully, the aio_write subroutine returns a value of 0. Otherwise, it returns a value of -1 and sets the errno global variable to identify the error.

Return codes can be set to the following errno values:

<table>
<thead>
<tr>
<th>Error Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EAGAIN</td>
<td>Indicates that the system resources required to queue the request are not available. Specifically, the transmit queue may be full, or the maximum number of opens may have been reached.</td>
</tr>
<tr>
<td>EBADF</td>
<td>Indicates that the FileDescriptor parameter is not valid.</td>
</tr>
<tr>
<td>EFAULT</td>
<td>Indicates that the address specified by the aiocbp parameter is not valid.</td>
</tr>
</tbody>
</table>
EINVAL Indicates that the aio_whence field does not have a valid value or that the resulting pointer is not valid. The aio_write or aio_write64 subroutine was used with a file descriptor obtained from a call to the shm_open subroutine.

When using I/O Completion Ports with AIO Requests, return codes can also be set to the following errno values:

EBADF Indicates that the aio_iocpfld field in the aiocb structure is not a valid I/O Completion Port file descriptor.
EINVAL Indicates that an I/O Completion Port service failed when attempting to start the AIO Request.
EPERM Indicates that I/O Completion Port services are not available.

Note: Other error codes defined in the /usr/include/sys/errno.h file may be returned by the aio_error subroutine if an error during the I/O operation is encountered.

Related Information
"aio_cancel or aio_cancel64 Subroutine" on page 38, "aio_error or aio_error64 Subroutine" on page 42, "aio_nwait Subroutine" on page 47, "aio_nwait_timeout Subroutine" on page 49, "aio_read or aio_read64 Subroutine" on page 50, "aio_return or aio_return64 Subroutine" on page 55, "aio_suspend or aio_suspend64 Subroutine" on page 58, "lio_listio or lio_listio64 Subroutine" on page 713.


The Input and Output Handling Programmer's Overview in AIX 5L Version 5.3 General Programming Concepts: Writing and Debugging Programs describes the files, commands, and subroutines used for low-level, stream, terminal, and asynchronous I/O interfaces.

---

 alloc, dealloc, print, read_data, read_regs, symbol_addrs, write_data, and write_regs Subroutine

Purpose

Provide access to facilities needed by the pthread debug library and supplied by the debugger or application.

Library

pthread debug library (libpthdebug.a)

Syntax

#include <sys/pthdebug.h>

int alloc (pthdb_user_t user, len, bufp);

int dealloc (user, buf);

int print (user, str);

alloc, dealloc, print, read_data, read_regs, symbol_addrs, write_data, and write_regs Subroutine

Purpose

Provide access to facilities needed by the pthread debug library and supplied by the debugger or application.

Library

pthread debug library (libpthdebug.a)

Syntax

#include <sys/pthdebug.h>

int alloc (user, len, bufp);

int dealloc (user, buf);

int print (user, str);

int read_data (user, buf, addr, size)
pthdb_user_t user;
void *buf;
pthdb_addr_t addr;
int size;

int read_regs (user, tid, flags, context)
pthdb_user_t user;
tid_t tid;
unsigned long long flags;
struct context64 *context;

int symbol_addrs (user, symbols[], count)
pthdb_user_t user;
pthdb_symbol_t symbols[];
int count;

int write_data (user, buf, addr, size)
pthdb_user_t user;
void *buf;
pthdb_addr_t addr;
int size;

int write_regs (user, tid, flags, context)
pthdb_user_t user;
tid_t tid;
unsigned long long flags;
struct context64 *context;

Description

int alloc()
Allocates len bytes of memory and returns the address. If successful, 0 is returned; otherwise, a nonzero number is returned. This call back function is always required.

int dealloc()
Takes a buffer and frees it. If successful, 0 is returned; otherwise, a nonzero number is returned. This call back function is always required.

int print()
Prints the character string to the debugger’s stdout. If successful, 0 is returned; otherwise, a nonzero number is returned. This call back function is always required.

int read_data()
Reads the requested number of bytes of data at the requested location from an active process or core file and returns the data through a buffer. If successful, 0 is returned; otherwise, a nonzero number is returned. This call back function is always required.

int read_regs()
Reads the context information of a debuggee kernel thread from an active process or from a core file. The information should be formatted in context64 form for both a 32-bit and a 64-bit process. If successful, 0 is returned; otherwise, a nonzero number is returned. This function is only required when using the pthdb_pthread_context and pthdb_pthread_setcontext subroutines.

int symbol_addrs()
Resolves the address of symbols in the debuggee. The pthread debug library calls this subroutine to get the address of known debug symbols. If the symbol has a name of NULL or "", set the address to 0LL instead of doing a lookup or returning an error. If successful, 0 is returned; otherwise, a nonzero number is returned. In introspective mode, when the PTHDB_FLAG_SUSPEND flag is set, the application can use the pthread debug library by passing NULL, or it can use one of its own.

int write_data()
Writes the requested number of bytes of data to the requested location. The libpthdebug.a library may use this to write data to the active process. If successful, 0 is returned; otherwise, a nonzero
number is returned. This call back function is required when the `PTHDB_FLAG_HOLD` flag is set and when using the `pthdb_pthread_setcontext` subroutine.

```c
int write_regs()
```

Writes requested context information to specified debuggee’s kernel thread id. If successful, 0 is returned; otherwise, a nonzero number is returned. This subroutine is only required when using the `pthdb_pthread_setcontext` subroutine.

**Note:** If the `write_data` and `write_regs` subroutines are NULL, the pthread debug library will not try to write data or regs. If the `pthdb_pthread_set_context` subroutine is called when the `write_data` and `write_regs` subroutines are NULL, `PTHDB_NOTSUP` is returned.

**Parameters**

- `user` : User handle.
- `symbols` : Array of symbols.
- `count` : Number of symbols.
- `buf` : Buffer.
- `addr` : Address to be read from or wrote to.
- `size` : Size of the buffer.
- `flags` : Session flags, must accept `PTHDB_FLAG_GPRS`, `PTHDB_FLAG_SPRS`, `PTHDB_FLAG_FPRS`, and `PTHDB_FLAG_REGS`.
- `tid` : Thread id.
- `flags` : Flags that control which registers are read or wrote.
- `context` : Context structure.
- `len` : Length of buffer to be allocated or reallocated.
- `bufp` : Pointer to buffer.
- `str` : String to be printed.

**Return Values**

If successful, these subroutines return 0; otherwise they return a nonzero value.

**Related Information**

The "malloc, free, realloc, calloc, mallopt, mallinfo, mallinfo_heap, alloca, valloc, or posix_memalign Subroutine" on page 769.

**allocclmb Subroutine**

**Purpose**

Allocates a contiguous block of contiguous real memory for exclusive use by the caller. The block of memory reserved will be the size of a system LMB.

**Syntax**

```c
#include <sys/dr.h>

int allocclmb(long long *laddr, int flags)
```

**Description**

The `allocclmb()` subroutine reserves an LMB sized block of contiguous real memory for exclusive use by the caller. It returns the partition logical address of that memory in `*laddr`. 
allocmb() is only valid in an LPAR environment, and it fails (with ENOTSUP) if called in another environment.

Only a privileged user should call allocmb().

Parameters

| laddr       | On successful return, contains the logical address of the allocated LMB. |
| flags       | Must be 0. |

Execution Environment

This allocmb() interface should only be called from the process environment.

Return Values

0 The LMB is successfully allocated.

Error Codes

<table>
<thead>
<tr>
<th>Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ENOTSUP</td>
<td>LMB allocation not supported on this system.</td>
</tr>
<tr>
<td>EINVAL</td>
<td>Invalid flags value.</td>
</tr>
<tr>
<td>EINVAL</td>
<td>Not in the process environment.</td>
</tr>
<tr>
<td>ENOMEM</td>
<td>A free LMB could not be made available.</td>
</tr>
</tbody>
</table>

Related Information

“freelmb Subroutine” on page 310

arm_end Subroutine

Purpose

The arm_end subroutine is used to mark the end of an application. This subroutine call must always be called when a program that issued an arm_init subroutine call terminates. In the PTX® implementation of ARM, application data structures may persist after arm_end is issued.

Library

ARM Library (libarm.a).

Syntax

```c
#include arm.h

arm_ret_stat_t ARM_API arm_end( arm_appl_id_t appl_id, /* application id */
                               arm_flag_t flags,    /* Reserved = 0 */
                               arm_data_t *data,   /* Reserved = NULL */
                               arm_data_sz_t data_size); /* Reserved = 0 */
```

Base Operating System (BOS) Runtime Services (A-P) 69
Description

By calling the arm_end subroutine, an application program signals to the ARM library that it has ceased issuing ARM subroutine calls for the application specified and that the library code can remove references to the application. As far as the calling program is concerned, all references to transactions defined for the named application can be removed as well.

This subroutine is part of the implementation of the ARM API in the Performance Toolbox for AIX licensed product.

Note that, in the PTX implementation of ARM, multiple processes can issue arm_init subroutine calls for a given application with the effect that multiple simultaneous definitions of the application are effective. The ARM library code points all these definitions to a single application structure in the ARM private shared memory area. A use-count keeps track of the number of simultaneous definitions. Each time arm_init is issued for the application name, the counter is incremented and each time the arm_end subroutine call is issued for the associated appl_id, the counter is decremented. No call to arm_end is permitted to decrement the counter less than zero.

Only when the counter reaches zero is the application structure inactivated. As long as the counter is non-zero, transactions defined for the application remain active and new transactions can be defined for the application. It does not matter which process created the definition of the application.

This implementation was chosen because it makes perfect sense in a PTX environment. Any more restrictive implementation would have increased memory use significantly and would be useless for PTX monitoring purposes.

Parameters

appl_id

The identifier is returned by an earlier call to arm_init, The PTX implementation does not require that the arm_init subroutine call was issued by the same program or process now issuing the arm_end subroutine call. However, each time the arm_end subroutine call is issued against an appl_id, the use-count of the transaction structure is decremented. When the count reaches zero, the application structure and all associated transaction structures are marked as inactive. Subsequent arm_init calls can reactivate the application structure but transaction structures formerly associated with the application are not automatically activated. Each transaction must be reactivated through the arm_getid subroutine call.

The appl_id is used to look for an application structure. If none is found, no action is taken and the function returns -1. If one is found, the use-count of the application structure is decremented. If that makes the counter zero, the use-counts of all associated transaction structures are set to zero. The total number of application structures that have been initialized for the calling process but not ended is decremented. If this count reaches zero, access to the shared memory from the process is released and the count of users of the shared memory area is decremented. If the count of users of the shared memory segment reaches zero, the shared memory segment is deleted.

flags, data, data_size

In the current API definition, the last three arguments are for future use and they are ignored in the implementation.

Return Values

If successful, the subroutine returns zero. If the subroutine fails, a value less than zero is returned.
Error Codes
No error codes are defined by the PTX implementation of the ARM API.

Files
/usr/include/arm.h Declares the subroutines, data structures, handles, and macros that an application program can use to access the ARM library.

Related Information
arm_init subroutine, arm_getid subroutine.

arm_end Dual Call Subroutine

Purpose
The arm_end subroutine is used to mark the end of an application. This subroutine call must always be called when a program that issued an arm_init subroutine call terminates. In the PTX implementation of ARM, application data structures may persist after arm_end is issued. This may not be the case for the lower library implementation.

Library
ARM Library (libarm.a).

Syntax
#include arm.h

arm_ret_stat_t ARM_API arm_end( arm_appl_id_t appl_id, /* application id */
     arm_flag_t flags,  /* Reserved = 0 */
     arm_data_t *data, /* Reserved = NULL */
     arm_data_sz_t data_size); /* Reserved = 0 */

Description
By calling the arm_end subroutine, an application program signals to the ARM library that it has ceased issuing ARM subroutine calls for the application specified and that the library code can remove references to the application. As far as the calling program is concerned, all references to transactions defined for the named application can be removed as well.

Before the PTX implementation code is executed, the lower library is called. If this call returns a value of zero, that return value is passed to the caller. If the value returned by the lower library is non-zero, the return value is the one generated by the PTX library code.

This subroutine is part of the implementation of the ARM API in the Performance Toolbox for AIX licensed product.

Note that, in the PTX implementation of ARM, multiple processes can issue arm_init subroutine calls for a given application with the effect that multiple simultaneous definitions of the application are effective. The ARM library code points all these definitions to a single application structure in the ARM private shared memory area. A use-count keeps track of the number of simultaneous definitions. Each time arm_init is issued for the application name, the counter is incremented and each time the arm_end subroutine call is issued for the associated appl_id, the counter is decremented. No call to arm_end is permitted to decrement the counter less than zero.
Only when the counter reaches zero is the application structure inactivated. As long as the counter is non-zero, transactions defined for the application remain active and new transactions can be defined for the application. It does not matter which process created the definition of the application.

This implementation was chosen because it makes perfect sense in a PTX environment. Any more restrictive implementation would have increased memory use significantly and would be useless for PTX monitoring purposes.

For the implementation of `arm_end` in the lower library, other restrictions may exist.

**Parameters**

`appl_id`

The identifier returned by an earlier call to `arm_init` ("arm_init Dual Call Subroutine" on page 79). The identifier is passed to the `arm_end` function of the lower library. If the lower library returns a zero, a zero is returned to the caller. After the invocation of the lower library, the PTX implementation attempts to translate the `appl_id` argument to its own identifier from the cross-reference table created by `arm_init` ("arm_init Dual Call Subroutine" on page 79). If one can be found, it is used for the PTX implementation; if no cross reference is found, the `appl_id` is used as passed in. The PTX implementation does not require that the `arm_init` subroutine call was issued by the same program or process now issuing the `arm_end` subroutine call. However, each time the `arm_end` subroutine call is issued against an `appl_id`, the use-count of the transaction structure is decremented. When the count reaches zero, the application structure and all associated transaction structures are marked as inactive. Subsequent `arm_init` calls can reactivate the application structure but transaction structures formerly associated with the application are not automatically activated. Each transaction must be reactivated through the `arm_getid` ("arm_getid Dual Call Subroutine" on page 75) subroutine call.

In the PTX implementation, the `appl_id` (as retrieved from the cross-reference table) is used to look for an application structure. If none is found, no action is taken and the PTX function is considered to have failed. If one is found, the use-count of the application structure is decremented. If that makes the counter zero, the use-counts of all associated transaction structures are set to zero. The total number of application structures that have been initialized for the calling process but not ended is decremented. If this count reaches zero, access to the shared memory from the process is released and the count of users of the shared memory area is decremented. If the count of users of the shared memory segment reaches zero, the shared memory segment is deleted.

`flags`, `data`, `data_size`

In the current API definition, the last three arguments are for future use and they are ignored in the implementation.

**Return Values**

If successful, the subroutine returns zero. If the subroutine fails, a value less than zero is returned. If the call to the lower library was successful, a zero is returned. If the subroutine call to the lower library failed but the PTX implementation didn't fail, a zero is returned. If both implementations failed, a value less than zero is returned.

**Error Codes**

No error codes are defined by the PTX implementation of the ARM API.

**Files**

`/usr/include/arm.h` Declares the subroutines, data structures, handles, and macros that an application program can use to access the ARM library.
Related Information
- “arm_init Dual Call Subroutine” on page 77
- “arm_getid Dual Call Subroutine” on page 75

arm_getid Subroutine

Purpose
The arm_getid subroutine is used to register a transaction as belonging to an application and assign a unique identifier to the application/transaction pair. In the PTX implementation of ARM, multiple instances of a transaction within one application can’t be defined. A transaction must be registered before any ARM measurements can begin.

Library
ARM Library (libarm.a).

Syntax
```c
#include arm.h

arm_tran_id_t arm_getid( arm_appl_id_t appl_id, /* application handle */
                        arm_ptr_t *tran_name,   /* transaction name */
                        arm_ptr_t *tran_detail, /* transaction additional info */
                        arm_flag_t flags,      /* Reserved = 0 */
                        arm_data_t *data,      /* Reserved = NULL */
                        arm_data_sz_t data_size);
```

Description
Each transaction needs to be defined by a unique name within an application. Transactions can be defined so they best fit the application environment. For example, if a given environment has thousands of unique transactions, it may be feasible to define groups of similar transactions to prevent data overload. In other situations, you may want to use generated transaction names that reflect what data a transaction carries along with the transaction type. For example, the type of SQL query could be analyzed to group customer query transactions according to complexity, such as customer_simple, customer, customer_complex. Whichever method is used to name transactions, in the PTX implementation of the ARM API, measurements are always collected for each unique combination of:
1. Hostname of the machine where the instrumented application executes.
2. Unique application name.
3. Unique transaction name.

This subroutine is part of the implementation of the ARM API in the Performance Toolbox for AIX licensed product.

Note that the use-count for a transaction structure is either one or zero. This ensures that as long as the application structure is active, so are all transactions for which an arm_getid call was issued after the application was activated by an arm_init call. The transaction use-count is reset to zero by the arm_end call if this call causes the application use-count to go to zero.

Note that the implementation of arm_getid doesn’t allow unique instances of a transaction to be defined. The tran_id associated with a transaction is stored in the ARM shared memory area and will remain constant throughout the life of the shared memory area. Consequently, subsequent executions of a
program that defines one or more transactions under a given application will usually have the same ID returned for the transactions each time. The same is true when different programs define the same transaction within an application: As long as the shared memory area exists, they will all have the same ID returned. This is done to minimize the use of memory for transaction definitions and because it makes no difference from a PTX point of view.

If this is not acceptable from an application point of view, programs can dynamically generate transaction names to pass on the `arm_getid` subroutine call.

**Parameters**

**appl_id**

The identifier returned by an earlier call to `arm_init` [“arm_init Subroutine” on page 77](#). The PTX implementation does not require that the `arm_init` subroutine call was issued by the same program or process now issuing the `arm_getid` subroutine call. However, the number of issued `arm_init` subroutine calls for the application name must exceed the number of issued `arm_end` subroutine calls for this `appl_id`.

The `appl_id` is used to look for an application structure. If one is not found or if the use-count of the one found is zero, no action is taken and the function returns -1.

**tran_name**

A unique transaction name. The name only needs to be unique within the `appl_id`. The maximum length is 128 characters including the terminating zero. The argument is converted to a key by removing all blanks and truncating the string to 32 characters, including a terminating zero. This key is used to look for a transaction structure (that belongs to the application identified in the first argument) in the library’s private shared memory area. If a transaction structure is found, its use-count is set to one and the transaction ID stored in the structure is returned to the caller. If the structure is not found, one is created and assigned the next free transaction ID, given a use-count of one and added to the application’s linked list of transactions. The new assigned transaction ID is returned to the caller.

Up-to 64 bytes, including the terminating zero, of the `tran_name` parameter is saved as the description of the SpmiCx context that represents the transaction in the Spmi hierarchy. The key is used as the short name of the context.

**tran_detail**

Can be passed in as NULL or some means of specifying a unique instance of the transaction. In the PTX implementation of the ARM API, this parameter is ignored. Consequently, it is not possible to define unique instances of a transaction. If specified as non-NULL, this parameter must be a string not exceeding 128 bytes in length, including the terminating zero.

For the implementation to take this argument in use, another context level would have to be defined between the application context and the transaction context. This was deemed excessive.

**flags, data, data_size**

In the current API definition, the last three arguments are for future use and they are ignored in the implementation.

**Return Values**

If successful, the subroutine returns an `tran_id` application identifier. If the subroutine fails, a value less than zero is returned. In compliance with the ARM API specifications, the error return value can be passed to the `arm_start` [“arm_start Subroutine” on page 81](#) subroutine, which will cause `arm_start` to function as a no-operation.
Error Codes
No error codes are defined by the PTX implementation of the ARM API.

Files
/usr/include/arm.h Declares the subroutines, data structures, handles, and macros that an application program can use to access the ARM library.

Related Information
arm_init ("arm_init Subroutine" on page 77) subroutine, arm_end ("arm_end Subroutine" on page 69) subroutine.

arm_getid Dual Call Subroutine

Purpose
The arm_getid subroutine is used to register a transaction as belonging to an application and assign a unique identifier to the application/transaction pair. In the PTX implementation of ARM, multiple instances of a transaction within one application can't be defined. The lower library implementation of this subroutine may provide support for instances of transactions. A transaction must be registered before any ARM measurements can begin.

Library
ARM Library (libarm.a).

Syntax
#include arm.h

arm_tran_id_t arm_getid( arm_appl_id_t appl_id, /* application handle */
    arm_ptr_t *tran_name, /* transaction name */
    arm_ptr_t *tran_detail, /* transaction additional info */
    arm_flag_t flags, /* Reserved = 0 */
    arm_data_t *data, /* Reserved = NULL */
    arm_data_sz_t data_size); /* Reserved = 0 */

Description
Each transaction needs to be defined by a unique name within an application. Transactions can be defined so they best fit the application environment. For example, if a given environment has thousands of unique transactions, it may be feasible to define groups of similar transactions to prevent data overload. In other situations, you may want to use generated transaction names that reflect what data a transaction carries along with the transaction type. For example, the type of SQL query could be analyzed to group customer query transactions according to complexity, such as customer_simple, customer, customer_complex. Whichever method is used to name transactions, in the PTX implementation of the ARM API, measurements are always collected for each unique combination of:
1. Hostname of the machine where the instrumented application executes.
2. Unique application name.
3. Unique transaction name.

Before the PTX implementation code is executed, the lower library is called. If this call returns a value greater than zero, that return value is passed to the caller as the transaction ID. If the returned value from the lower library is zero or negative, the return value is the one generated by the PTX library code.
This subroutine is part of the implementation of the ARM API in the Performance Toolbox for AIX licensed product.

Note that the use-count for a transaction structure is either one or zero. This ensures that as long as the application structure is active, so are all transactions for which an `arm_getid` call was issued after the application was activated by an `arm_init` call. The transaction use-count is reset to zero by the `arm_end` call if this call causes the application use-count to go to zero.

Note that the implementation of `arm_getid` doesn't allow unique instances of a transaction to be defined. The `tran_id` associated with a transaction is stored in the ARM shared memory area and will remain constant throughout the life of the shared memory area. Consequently, subsequent executions of a program that defines one or more transactions under a given application will usually have the same ID returned for the transactions each time. The same is true when different programs define the same transaction within an application: As long as the shared memory area exists, they will all have the same ID returned. This is done to minimize the use of memory for transaction definitions and because it makes no difference from a PTX point of view.

If this is not acceptable from an application point of view, programs can dynamically generate transaction names to pass on the `arm_getid` subroutine call.

Regardless of the implementation restrictions of the PTX library, the `lower library` may or may not have its own implementation restrictions.

**Parameters**

**appl_id**

The identifier returned by an earlier call to `arm_init`.

The identifier is passed to the `arm_getid` function of the `lower library`. If the `lower library` returns an identifier greater than zero, that identifier is the one that'll eventually be returned to the caller.

After the invocation of the `lower library`, the PTX implementation attempts to translate the `appl_id` argument to its own identifier by consulting the cross-reference table created by `arm_init`. If one can be found, it is used for the PTX implementation; if no cross reference is found, the `appl_id` is used as passed in. The PTX implementation does not require that the `arm_init` subroutine call was issued by the same program or process now issuing the `arm_getid` subroutine call. However, the number of issued `arm_init` subroutine calls for the application name must exceed the number of issued `arm_end` subroutine calls for this `appl_id`.

In the PTX implementation, the `appl_id` (as retrieved from the cross-reference table) is used to look for an application structure. If one is not found or if the use-count of the one found is zero, the PTX implementation is considered to have failed and no action is taken by the PTX library.

**tran_name**

A unique transaction name. The name only needs to be unique within the `appl_id`. The maximum length is 128 characters including the terminating zero. In the PTX implementation, the argument is converted to a key by removing all blanks and truncating the string to 32 characters, including a terminating zero. This key is used to look for a transaction structure (that belongs to the application identified in the first argument) in the library's private shared memory area. If a transaction structure is found, its use-count is set to one and the transaction ID stored in the structure is saved. If the structure is not found, one is created and assigned the next free transaction ID, given a use-count of one and added to the application's linked list of transactions. The new assigned transaction ID is saved. If the call to the `lower library` was successful, a cross-reference is created from the `lower library's` transaction ID to the PTX library's transaction ID for use by `arm_start`.

---

76 Technical Reference, Volume 1: Base Operating System and Extensions
Up-to 64 bytes, including the terminating zero, of the tran_name parameter is saved as the description of the SpmiCx context that represents the transaction in the Spmi hierarchy. The key is used as the short name of the context.

**tran_detail**

Can be passed in as NULL or some means of specifying a unique instance of the transaction. In the PTX implementation of the ARM API, this parameter is ignored. Consequently, it is not possible to define unique instances of a transaction. If specified as non-NULL, this parameter must be a string not exceeding 128 bytes in length, including the terminating zero.

For the implementation to take this argument in use, another context level would have to be defined between the application context and the transaction context. This was deemed excessive.

For the *lower library* implementation of this subroutine call, the tran_detail argument may have significance. If so, it's transparent to the PTX implementation.

**flags, data, data_size**

In the current API definition, the last three arguments are for future use and they are ignored in the implementation. In the current API definition, the last three arguments are for future use and they are ignored in the implementation.

**Return Values**

If successful, the subroutine returns an tran_id application identifier. If the subroutine fails, a value less than zero is returned. In compliance with the ARM API specifications, the error return value can be passed to the arm_start subroutine, which will cause arm_start to function as a no-operation.

If the call to the *lower library* was successful, the tran_id transaction identifier returned is the one assigned by the *lower library*. If the subroutine call to the *lower library* failed but the PTX implementation didn't fail, the tran_id returned is the one assigned by the PTX library. If both implementations fail, a value less than zero is returned. In compliance with the ARM API specification, an error return value can be passed to the arm_start subroutine, which will cause arm_start to function as a no-operation.

**Error Codes**

No error codes are defined by the PTX implementation of the ARM API.

**Files**

/usr/include/arm.h Declares the subroutines, data structures, handles, and macros that an application program can use to access the ARM library.

**Related Information**

arm_init subroutine, arm_end subroutine.

**arm_init Subroutine**

**Purpose**

The arm_init subroutine is used to define an application or a unique instance of an application to the ARM library. In the PTX implementation of ARM, instances of applications can't be defined. An application must be defined before any other ARM subroutine is issued.
Library
ARM Library (libarm.a).

Syntax
#include arm.h

arm_appl_id_t arm_init( arm_ptr_t *appname, /* application name */
arm_ptr_t *appl_user_id, /* Name of the application user */
arm_flag_t flags, /* Reserved = 0 */
arm_data_t *data, /* Reserved = NULL */
arm_data_sz_t data_size); /* Reserved = 0 */

Description
Each application needs to be defined by a unique name. An application can be defined as loosely or as rigidly as required. It may be defined as a single execution of one program, multiple (possibly simultaneous) executions of one program, or multiple executions of multiple programs that together constitute an application. Any one user of ARM may define the application so it best fits the measurement granularity desired. Measurements are always collected for each unique combination of:
1. Hostname of the machine where the instrumented application executes.
2. Unique application name.
3. Unique transaction name.

This subroutine is part of the implementation of the ARM API in the Performance Toolbox for AIX licensed product.

Note that the implementation of arm_init doesn’t allow unique instances of an application to be defined. The appl_id associated with an application is stored in the ARM shared memory area and will remain constant throughout the life of the shared memory area. Consequently, subsequent executions of a program that defines one or more applications will usually have the same ID returned for the application each time. The same is true when different programs define the same application: As long as the shared memory area exists, they will all have the same ID returned. This is done to minimize the use of memory for application definitions and because it makes no difference from a PTX point of view.

If this is not acceptable from an application point of view, programs can dynamically generate application names to pass on the arm_init subroutine call.

Parameters
appname

A unique application name. The maximum length is 128 characters including the terminating zero. The argument is converted to a key by removing all blanks and truncating the string to 32 characters, including a terminating zero. This key is used to look for an application structure in the library’s private shared memory area. If a structure is found, its use-count is incremented and the application ID stored in the structure is returned to the caller. If the structure is not found, one is created, assigned the next free application ID and given a use-count of one. The new assigned application ID is returned to the caller.

Up-to 64 bytes, including the terminating zero, of the appname parameter is saved as the description of the SpmiCx context that represents the application in the Spmi hierarchy. The key is used as the short name of the context.

appl_user_id

Can be passed in as NULL or some means of specifying a user ID for the application. This allows the calling program to define unique instances of an application. In the PTX implementation of the
ARM API, this parameter is ignored. Consequently, it is not possible to define unique instances of
an application. If specified as non-NULL, this parameter must be a string not exceeding 128 bytes
in length, including the terminating zero.

For the implementation to take this argument in use, another context level would have to be
defined between the application context and the transaction context. This was deemed excessive.

flags, data, data_size
In the current API definition, the last three arguments are for future use and they are ignored in the
implementation.

Return Values
If successful, the subroutine returns an appl_id application identifier. If the subroutine fails, a value less
than zero is returned.

Error Codes
No error codes are defined by the PTX implementation of the ARM API.

Files
/usr/include/arm.h Declares the subroutines, data structures, handles, and macros that an
application program can use to access the ARM library.

arm_init Dual Call Subroutine

Purpose
The arm_init subroutine is used to define an application or a unique instance of an application to the ARM
library. While, in the PTX implementation of ARM, instances of applications can’t be defined, the ARM
implementation in the lower library may permit this. An application must be defined before any other ARM
subroutine is issued.

Library
ARM Library (libarm.a).

Syntax
#include arm.h

arm_appl_id_t arm_init( arm_ptr_t *appname, /* application name
*/
               arm_ptr_t *appl_user_id, /* Name of the application user */
               arm_flag_t flags, /* Reserved = 0 */
               arm_data_t *data, /* Reserved = NULL */
               arm_data_sz_t data_size); /* Reserved = 0 */

Description
Each application needs to be defined by a unique name. An application can be defined as loosely or as
rigidly as required. It may be defined as a single execution of one program, multiple (possibly
simultaneous) executions of one program, or multiple executions of multiple programs that together
constitute an application. Any one user of ARM may define the application so it best fits the measurement
granularity desired. For the PTX implementation, measurements are always collected for each unique
combination of:
1. Hostname of the machine where the instrumented application executes.
2. Unique application name.
3. Unique transaction name.

Before the PTX implementation code is executed, the lower library is called. If this call returns a value greater than zero, that return value is passed to the caller as the application ID. If the returned value from the lower library is zero or negative, the return value is the one generated by the PTX library code.

This subroutine is part of the implementation of the ARM API in the Performance Toolbox for AIX licensed product.

Note that the implementation of arm_init doesn't allow unique instances of an application to be defined. The appl_id associated with an application is stored in the ARM shared memory area and will remain constant throughout the life of the shared memory area. Consequently, subsequent executions of a program that defines one or more applications will usually have the same ID returned for the application each time. The same is true when different programs define the same application: As long as the shared memory area exists, they will all have the same ID returned. This is done to minimize the use of memory for application definitions and because it makes no difference from a PTX point of view.

If this is not acceptable from an application point of view, programs can dynamically generate application names to pass on the arm_init subroutine call.

Regardless of the implementation restrictions of the PTX library, the lower library may or may not have its own implementation restrictions.

**Parameters**

**appname**

A unique application name. The maximum length is 128 characters including the terminating zero. The PTX library code converts this value to a key by removing all blanks and truncating the string to 32 characters, including a terminating zero. This key is used to look for an application structure in the library’s private shared memory area. If a structure is found, its use-count is incremented and the application ID stored in the structure is saved. If the structure is not found, one is created, assigned the next free application ID and given a use-count of one. The new assigned application ID is saved. If the call to the lower library was successful, a cross-reference is created from the lower library’s application ID to the PTX library’s application ID for use by arm_getid ("arm_getid Dual Call Subroutine" on page 75) and arm_end ("arm_end Dual Call Subroutine" on page 71).

Up-to 64 bytes, including the terminating zero, of the appname parameter is saved as the description of the SpmiCx context that represents the application in the Spmi hierarchy. The key is used as the short name of the context.

**appl_user_id**

Can be passed in as NULL or some means of specifying a user ID for the application. This allows the calling program to define unique instances of an application. In the PTX implementation of the ARM API, this parameter is ignored. Consequently, it is not possible to define unique instances of an application. If specified as non-NULL, this parameter must be a string not exceeding 128 bytes in length, including the terminating zero.

For the PTX implementation to take this argument in use, another context level would have to be defined between the application context and the transaction context. This was deemed excessive.

For the lower library implementation of this subroutine call, the appl_user_id argument may have significance. If so, it’s transparent to the PTX implementation.

**flags, data, data_size**

In the current API definition, the last three arguments are for future use and they are ignored in the implementation.
Return Values
If the call to the lower library was successful, the subroutine returns an appl_id application identifier as returned from the lower library. If the subroutine call to the lower library fails but the PTX implementation doesn’t fail, the appl_id returned is the one assigned by the PTX library. If both implementations fail, a value less than zero is returned.

Error Codes
No error codes are defined by the PTX implementation of the ARM API.

Files
/usr/include/arm.h Declares the subroutines, data structures, handles, and macros that an application program can use to access the ARM library.

arm_start Subroutine

Purpose
The arm_start subroutine is used to mark the beginning of the execution of a transaction. Measurement of the transaction response time starts at the execution of this subroutine.

Library
ARM Library (libarm.a).

Syntax
#include arm.h

arm_start_handle_t arm_start( arm_tran_id_t tran_id, /* transaction name identifier */
    arm_flag_t flags, /* Reserved = 0 */
    arm_data_t *data, /* Reserved = NULL */
    arm_data_sz_t data_size); /* Reserved = 0 */

Description
Each arm_start subroutine call marks the beginning of another instance of a transaction within an application. Multiple instances (simultaneous executions of the transaction) may exist. Control information for the transaction instance is held until the execution of a matching arm_stop ("arm_stop Subroutine" on page 84) subroutine call, at which time the elapsed time is calculated and used to update transaction measurement metrics for the transaction. Metrics are accumulated for each unique combination of the following three components:
1. Hostname of the machine where the instrumented application executes.
2. Unique application name.
3. Unique transaction name.

This subroutine is part of the implementation of the ARM API in the Performance Toolbox for AIX licensed product.

Parameters
tran_id
The identifier is returned by an earlier call to arm_getid, "arm_getid Subroutine” on page 73. The PTX implementation does not require that the arm_getid subroutine call was issued by the same
program or process now issuing the `arm_start` subroutine call. However, the transaction's application structure must be active, which means that the number of issued `arm_init` subroutine calls for the application name must exceed the number of issued `arm_end` subroutine calls for the application's `appl_id`. If an application was inactivated by issuing a sufficient number of `arm_end` calls, all transactions defined for that application will have their use_count set to zero. The count remains zero (and the transaction inactive) until a new `arm_getid` subroutine is issued for the transaction.

The `tran_id` argument is used to look for a transaction structure. If one is not found or if the use-count of the one found is zero, no action is taken and the function returns -1. If one is found, a transaction instance structure (called a slot structure) is allocated, assigned the next free instance ID, and updated with the start time of the transaction instance. The assigned instance ID is returned to the caller.

In compliance with the ARM API specifications, if the `tran_id` passed is one returned from a previous `arm_getid` subroutine call that failed, the `arm_start` subroutine call functions as a no-operation function. It will return a NULL `start_handle`, which can be passed to subsequent `arm_update` ("arm_update Subroutine" on page 88) and `arm_stop` ("arm_stop Subroutine" on page 84) subroutine calls with the effect that those calls are no-operation functions.

flags, data, data_size

In the current API definition, the last three arguments are for future use and they are ignored in the implementation.

Return Values

If successful, the subroutine returns a `start_handle`, which uniquely defines this transaction execution instance. If the subroutine fails, a value less than zero is returned. In compliance with the ARM API specifications, the error return value can be passed to the `arm_update` ("arm_update Subroutine" on page 88) and `arm_stop` ("arm_stop Subroutine" on page 84) subroutines, which will cause those subroutines to operate as no-operation functions.

Error Codes

No error codes are defined by the PTX implementation of the ARM API.

Files

/usr/include/arm.h
Declares the subroutines, data structures, handles, and macros that an application program can use to access the ARM library.

Related Information

`arm_init` ("arm_init Subroutine" on page 77) subroutine, `arm_getid` ("arm_getid Subroutine" on page 73) subroutine, `arm_end` ("arm_end Subroutine" on page 69) subroutine.

---

`arm_start` Dual Call Subroutine

Purpose

The `arm_start` subroutine is used to mark the beginning of the execution of a transaction. Measurement of the transaction response time starts at the execution of this subroutine.

Library

ARM Library (libarm.a).
Syntax

```c
#include arm.h

arm_start_handle_t arm_start( arm_tran_id_t tran_id, /* transaction name identifier */
    arm_flag_t flags, /* Reserved = 0 */
    arm_data_t *data, /* Reserved = NULL */
    arm_data_sz_t data_size); /* Reserved = 0 */
```

Description

Each `arm_start` subroutine call marks the beginning of another instance of a transaction within an application. Multiple instances (simultaneous executions of the transaction) may exist. Control information for the transaction instance is held until the execution of a matching `arm_stop` subroutine call, at which time the elapsed time is calculated and used to update transaction measurement metrics for the transaction. Metrics are accumulated for each unique combination of the following three components:

1. Hostname of the machine where the instrumented application executes.
2. Unique application name.
3. Unique transaction name.

Before the PTX implementation code is executed, the lower library is called. If this call returns a value greater than zero, that return value is passed to the caller as the start handle. If the value returned by the lower library is zero or negative, the return value is the one generated by the PTX library code.

This subroutine is part of the implementation of the ARM API in the Performance Toolbox for AIX licensed product.

Parameters

**tran_id**

The identifier is returned by an earlier call to `arm_getid`, function of the lower library. If the lower library returns an identifier greater than zero, that identifier is the one that will eventually be returned to the caller. After the invocation of the lower library, the PTX implementation attempts to translate the `tran_id` argument to its own identifier from the cross-reference table created by `arm_getid`. If one can be found, it is used for the PTX implementation; if no cross reference is found, the `tran_id` is used as passed in. The PTX implementation does not require that the `arm_getid` subroutine call was issued by the same program or process now issuing the `arm_start` subroutine call. However, the transaction's application structure must be active, which means that the number of issued `arm_init` subroutine calls for the application name must exceed the number of issued `arm_end` subroutine calls for the application's `appl_id`. If an application was inactivated by issuing a sufficient number of `arm_end` calls, all transactions defined for that application will have their use_count set to zero. The count remains zero (and the transaction inactive) until a new `arm_getid` subroutine is issued for the transaction.

In the PTX implementation, the `tran_id` (as retrieved from the cross-reference table) is used to look for a transaction structure. If one is not found or if the use-count of the one found is zero, the PTX implementation is considered to have failed and no action is taken by the PTX library. If one is found, a transaction instance structure (called a slot structure) is allocated, assigned the next free instance ID, and updated with the start time of the transaction instance. The assigned instance ID is saved as the `start_handle`. If the call to the lower library was successful, a cross-reference is created from the lower library's `start_handle` to the PTX library's `start_handle` for use by `arm_update` and `arm_stop` subroutine calls for the transaction.
In compliance with the ARM API specifications, if the `tran_id` passed is one returned from a previous `arm_getid` subroutine call that failed, the `arm_start` subroutine call functions as a no-operation function. It will return a NULL `start_handle`, which can be passed to subsequent `arm_update` ("arm_update Dual Call Subroutine" on page 89) and `arm_stop` ("arm_stop Dual Call Subroutine" on page 86) subroutine calls with the effect that those calls are no-operation functions.

flags, data, data_size

In the current API definition, the last three arguments are for future use and they are ignored in the implementation.

Return Values
If successful, the subroutine returns a `start_handle`, which uniquely defines this transaction execution instance. If the subroutine fails, a value less than zero is returned. In compliance with the ARM API specifications, the error return value can be passed to the `arm_update` ("arm_update Dual Call Subroutine" on page 89) and `arm_stop` ("arm_stop Dual Call Subroutine" on page 86) subroutines, which will cause those subroutines to operate as no-operation functions.

If the call to the `lower library` was successful, the `start_handle` instance ID returned is the one assigned by the `lower library`. If the subroutine call to the `lower library` failed but the PTX implementation didn’t fail, the `start_handle` returned is the one assigned by the PTX library. If both implementations fail, a value less than zero is returned.

Error Codes
No error codes are defined by the PTX implementation of the ARM API.

Files
/usr/include/arm.h
Declares the subroutines, data structures, handles, and macros that an application program can use to access the ARM library.

Related Information
`arm_init` ("arm_init Dual Call Subroutine" on page 79) subroutine, `arm_getid` ("arm_getid Dual Call Subroutine" on page 75) subroutine, `arm_end` ("arm_end Dual Call Subroutine" on page 71) subroutine.

arm_stop Subroutine

Purpose
The `arm_stop` subroutine is used to mark the end of the execution of a transaction. Measurement of the transaction response time completes at the execution of this subroutine.

Library
ARM Library (libarm.a).

Syntax
```c
#include arm.h

arm_ret_stat_t arm_stop( arm_start_handle_t arm_handle,
                          const arm_status_t comp_status,
                          arm_flag_t flags,
                          arm_data_t * data,
                          arm_data_sz_t data_size);
```


Description
Each **arm_stop** subroutine call marks the end of an instance of a transaction within an application. Multiple instances (simultaneous executions of the transaction) may exist. Control information for the transaction instance is held from the execution of the **arm_start** subroutine call and until the execution of a matching **arm_stop** subroutine call, at which time the elapsed time is calculated and used to update transaction measurement metrics for the transaction. Metrics are accumulated for each unique combination of the following three components:
1. Hostname of the machine where the instrumented application executes.
2. Unique application name.
3. Unique transaction name.

This subroutine is part of the implementation of the ARM API in the Performance Toolbox for AIX licensed product.

Parameters

**arm_handle**

The identifier is returned by an earlier call to **arm_start**, The **arm_handle** argument is used to look for a slot structure created by the **arm_start** call, which returned this **arm_handle**. If one is not found, no action is taken and the function returns -1. If one is found, a post structure is allocated and added to the linked list of post structures used to pass data to the SpmiArmd daemon. The post structure is updated with the start time from the slot structure, the path to the transaction context, and the stop time of the transaction instance.

In compliance with the ARM API specifications, if the **start_handle** passed is one returned from a previous **arm_start** subroutine call that failed, or from an **arm_start** subroutine operating as a no-operation function, the **arm_stop** subroutine call executes as a no-operation function. It will return a zero to indicate successful completion.

**comp_status**

User supplied transaction completion code. The following codes are defined:

- **ARM_GOOD** - successful completion. Response time is calculated. The response time is calculated as a fixed point value in milliseconds and saved in the metric **resptime**. In addition, the weighted average response time is calculated as a floating point value using a variable **weight** that defaults to 75%. The average response time is calculated as **weight** percent of the previous value of the average plus (100 - **weight**) percent of the latest response time observation. The value of **weight** can be changed from the SpmiArmd daemon’s configuration file /etc/perf/SpmiArmd.cf. In addition, the maximum and minimum response time for this transaction is updated, if required. Finally the **count** of successful transaction executions is incremented.

- **ARM_ABORT** - transaction aborted. The **aborted** counter is incremented. No other updates occur.

- **ARM_FAILED** - transaction failed. The **failed** counter is incremented. No other updates occur.

**flags, data, data_size**

In the current API definition, the last three arguments are for future use and they are ignored in the implementation.

Return Values

If successful, the subroutine returns zero. If the subroutine fails, a value less than zero is returned.
Error Codes
No error codes are defined by the PTX implementation of the ARM API.

Files
/usr/include/arm.h Declares the subroutines, data structures, handles, and macros that an application program can use to access the ARM library.

Related Information
arm_init (arm_init Subroutine” on page 77) subroutine, arm_getid (“arm_getid Subroutine” on page 73) subroutine, arm_start (“arm_start Subroutine” on page 81) subroutine, arm_end (“arm_end Subroutine” on page 69) subroutine.

arm_stop Dual Call Subroutine

Purpose
The arm_stop subroutine is used to mark the end of the execution of a transaction. Measurement of the transaction response time completes at the execution of this subroutine.

Library
ARM Library (libarm.a).

Syntax
#include arm.h

arm_ret_stat_t arm_stop( arm_start_handle_t arm_handle, /* unique transaction handle */
    const arm_status_t comp_status, /* Good=0, Abort=1, Failed=2 */
    arm_flag_t flags, /* Reserved = 0 */
    arm_data_t *data, /* Reserved = NULL */
    arm_data_sz_t data_size); /* Reserved = 0 */

Description
Each arm_stop subroutine call marks the end of an instance of a transaction within an application. Multiple instances (simultaneous executions of the transaction) may exist. Control information for the transaction instance is held from the execution of the arm_start (“arm_start Dual Call Subroutine” on page 82) subroutine call and until the execution of a matching arm_stop subroutine call, at which time the elapsed time is calculated and used to update transaction measurement metrics for the transaction. Metrics are accumulated for each unique combination of the following three components:

1. Hostname of the machine where the instrumented application executes.
2. Unique application name.
3. Unique transaction name.

Before the PTX implementation code is executed, the lower library is called. If this call returns a value of zero, that return value is passed to the caller. If the value returned by the lower library is non-zero, the return value is the one generated by the PTX library code.

This subroutine is part of the implementation of the ARM API in the Performance Toolbox for AIX licensed product.
Parameters

arm_handle

The identifier is returned by an earlier call to arm_start. The identifier is passed to the arm_stop function of the lower library. If the lower library returns a zero return code, that return code is returned to the caller. After the invocation of the lower library, the PTX implementation attempts to translate the arm_handle argument to its own identifier from the cross-reference table created by arm_start. If one can be found, it is used for the PTX implementation; if no cross reference is found, the arm_handle is used as passed in. The PTX implementation uses the start_handle argument to look for the slot structure created by the arm_start subroutine call. If one is found, a post structure is allocated and added to the linked list of post structures used to pass data to the SpmiArmd daemon. The post structure is updated with the start time from the slot structure, the path to the transaction context, and the stop time of the transaction instance. If no slot structure was found, the PTX implementation is considered to have failed.

In compliance with the ARM API specifications, if the start_handle passed is one returned from a previous arm_start subroutine call that failed, or from an arm_start subroutine operating as a no-operation function, the arm_stop subroutine call executes as a no-operation function. It will return a zero to indicate successful completion.

comp_status

User supplied transaction completion code. The following codes are defined:

• ARM_GOOD - successful completion. Response time is calculated. The response time is calculated as a fixed point value in milliseconds and saved in the metric resptime. In addition, the weighted average response time (in respavg) is calculated as a floating point value using a variable weight, that defaults to 75%. The average response time is calculated as weight percent of the previous value of the average plus (100 - weight) percent of the latest response time observation. The value of weight can be changed from the SpmiArmd daemon’s configuration file /etc/perf/SpmiArmd.cf. In addition, the maximum and minimum response time for this transaction is updated, if required. Finally the count of successful transaction executions is incremented.

• ARM_ABORT - transaction aborted. The aborted counter is incremented. No other updates occur.

• ARM_FAILED - transaction failed. The failed counter is incremented. No other updates occur.

flags, data, data_size

In the current API definition, the last three arguments are for future use and they are ignored in the implementation. In the current API definition, the last three arguments are for future use and they are ignored in the implementation.

Return Values

If successful, the subroutine returns zero. If the subroutine fails, a value less than zero is returned. If the call to the lower library was successful, a zero is returned. If the subroutine call to the lower library failed but the PTX implementation didn’t fail, a zero is returned. If both implementations failed, a value less than zero is returned.

Error Codes

No error codes are defined by the PTX implementation of the ARM API.

Files

/usr/include/arm.h Declares the subroutines, data structures, handles, and macros that an application program can use to access the ARM library.
Related Information

arm_init subroutine, arm_getid subroutine, arm_start subroutine, arm_end subroutine.

arm_update Subroutine

Purpose
The arm_update subroutine is used to collect information about a transaction’s progress. It is a no-operation subroutine in the PTX implementation.

Library
ARM Library (libarm.a).

Syntax

```
#include arm.h

arm_ret_stat_t arm_update( arm_start_handle_t arm_handle, /* unique transaction handle */
               arm_flag_t flags, /* Reserved = 0 */
               arm_data_t *data, /* Reserved = NULL */
               arm_data_sz_t data_size); /* Reserved = 0 */
```

Description
The arm_update subroutine is implemented as a no-operation in the PTX version of the ARM API. It is intended to be used for providing status information for a long-running transaction. Because there’s no feasible way to display such information in current PTX monitors, the subroutine is a NULL function.

This subroutine is part of the implementation of the ARM API in the Performance Toolbox for AIX licensed product. It is implemented as a NULL subroutine call.

Parameters

start_handle

The identifier is returned by an earlier call to arm_start subroutine. The start_handle argument is used to look for the slot structure created by the arm_start subroutine call. If one is not found, no action is taken and the function returns -1. Otherwise a zero is returned.

In compliance with the ARM API specifications, if the start_handle passed is one returned from a previous arm_start subroutine call that failed, or from an arm_start subroutine operating as a no-operation function, the arm_update subroutine call executes as a no-operation function. It will return a zero to indicate successful completion.

flags, data, data_size

In the current API definition, the last three arguments are for future use and they are ignored in the implementation.

Return Values

If successful, the subroutine returns zero. If the subroutine fails, a value less than zero is returned.

Error Codes

No error codes are defined by the PTX implementation of the ARM API.
Files

/usr/include/arm.h Declares the subroutines, data structures, handles, and macros that an application program can use to access the ARM library.

Related Information

arm_init ("arm_init Subroutine" on page 77) subroutine, arm_getid ("arm_getid Subroutine" on page 73) subroutine, arm_start ("arm_start Subroutine" on page 81) subroutine, arm_stop ("arm_stop Subroutine" on page 84) subroutine, arm_end ("arm_end Subroutine" on page 69) subroutine.

arm_update Dual Call Subroutine

Purpose

The arm_update subroutine is used to collect information about a transaction’s progress. It is a no-operation subroutine in the PTX implementation but may be fully implemented by the lower library.

Library

ARM Library (libarm.a).

Syntax

```c
#include arm.h

arm_ret_stat_t arm_update( arm_start_handle_t arm_handle, /* unique transaction handle */
                         arm_flag_t flags, /* Reserved = 0 */
                         arm_data_t *data, /* Reserved = NULL */
                         arm_data_sz_t data_size); /* Reserved = 0 */
```

Description

The arm_update subroutine is implemented as a no-operation in the PTX version of the ARM API. It is intended to be used for providing status information for a long-running transaction. Because there’s no feasible way to display such information in current PTX monitors, the subroutine is a NULL function.

The lower library implementation of the arm_update subroutine is always invoked.

This subroutine is part of the implementation of the ARM API in the Performance Toolbox for AIX licensed product. It is implemented as a NULL subroutine call.

Parameters

start_handle

The identifier is returned by an earlier call to arm_start ("arm_start Dual Call Subroutine" on page 82). The identifier is passed to the arm_update function of the lower library. If the lower library returns a zero return code, that return code is returned to the caller. After the invocation of the lower library, the PTX implementation attempts to translate the arm_handle argument to its own identifier from the cross-reference table created by arm_start. If one can be found, it is used for the PTX implementation; if no cross reference is found, the arm_handle is used as passed in. The PTX implementation uses the start_handle argument to look for the slot structure created by the arm_start subroutine call. If one is found the PTX implementation is considered to have succeeded, otherwise it is considered to have failed.
In compliance with the ARM API specifications, if the `start_handle` passed is one returned from a previous `arm_start` subroutine call that failed, or from an `arm_start` subroutine operating as a no-operation function, the `arm_update` subroutine call executes as a no-operation function. It will return a zero to indicate successful completion.

flags, data, data_size

In the current API definition, the last three arguments are for future use and they are ignored in the implementation.

Return Values

If successful, the subroutine returns zero. If the subroutine fails, a value less than zero is returned. If the call to the lower library was successful, a zero is returned. If the subroutine call to the lower library failed but the PTX implementation didn't fail, a zero is returned. If both implementations failed, a value less than zero is returned.

Error Codes

No error codes are defined by the PTX implementation of the ARM API.

Files

`/usr/include/arm.h` Declares the subroutines, data structures, handles, and macros that an application program can use to access the ARM library.

Related Information

arm_init subroutine, arm_getid subroutine, arm_start subroutine, arm_stop subroutine, arm_end subroutine.

asinh, asinhf, or asinhl Subroutine

Purpose

Computes the inverse hyperbolic sine.

Syntax

```c
#include <math.h>

float asinhf (x)
float x;

long double asinhl (x)
long double x;

double asinh (x)
double x;
```

Description

The `asinh`, `asinhl`, and `asinhf` subroutines compute the inverse hyperbolic sine of the `x` parameter.
An application wishing to check for error situations should set `errno` to zero and call `fetestexcept(FE_ALL_EXCEPT)` before calling these subroutines. Upon return, if the `errno` global variable is nonzero or `fetestexcept(FE_INVALID | FE_DIVBYZERO | FE_OVERFLOW | FE_UNDERFLOW)` is nonzero, an error has occurred.

**Parameters**

- **x** Specifies the value to be computed.

**Return Values**

Upon successful completion, the `asinhf`, `asinhl`, and `asinh` subroutines return the inverse hyperbolic sine of the given argument.

- If `x` is NaN, a NaN is returned.
- If `x` is 0, or ±Inf, `x` is returned.
- If `x` is subnormal, a range error may occur and `x` will be returned.

**Related Information**

`math.h` in AIX 5L Version 5.3 Files Reference.

**asinf, asinl, or asin Subroutine**

**Purpose**

Computes the arc sine.

**Syntax**

```c
#include <math.h>

float asinf (x)
float x;

long double asinl (x)
long double x;

double asin (x)
double x;
```

**Description**

The `asinf`, `asinl`, and `asin` subroutines compute the principal value of the arc sine of the `x` parameter. The value of `x` should be in the range [-1,1].

An application wishing to check for error situations should set the `errno` global variable to zero and call `feclearexcept(FE_ALL_EXCEPT)` before calling these subroutines. On return, if `errno` is nonzero or `fetestexcept(FE_INVALID | FE_DIVBYZERO | FE_OVERFLOW | FE_UNDERFLOW)` is nonzero, an error has occurred.

**Parameters**

- **x** Specifies the value to be computed.
Return Values
Upon successful completion, the \texttt{asinf}, \texttt{asinl}, and \texttt{asin} subroutines return the arc sine of \( x \), in the range \([-\pi/2, \pi/2]\) radians.

For finite values of \( x \) not in the range \([-1,1]\), a domain error occurs, and a NaN is returned.

If \( x \) is NaN, a NaN is returned.

If \( x \) is 0, \( x \) is returned.

If \( x \) is ±Inf, a domain error occurs, and a NaN is returned.

If \( x \) is subnormal, a range error may occur and \( x \) is returned.

Related Information
The "asinh, asinhf, or asinhl Subroutine" on page 90.

\texttt{math.h} in AIX 5L Version 5.3 Files Reference.

Subroutines Overview in AIX 5L Version 5.3 General Programming Concepts: Writing and Debugging Programs.

---

assert Macro

Purpose
Verifies a program assertion.

Library
Standard C Library (\texttt{libc.a})

Syntax
\begin{verbatim}
#include <assert.h>

void assert ( \texttt{Expression} )
int \texttt{Expression};
\end{verbatim}

Description
The \texttt{assert} macro puts error messages into a program. If the specified expression is false, the \texttt{assert} macro writes the following message to standard error and stops the program:

\begin{verbatim}
Assertion failed: Expression, file FileName, line LineNumber
\end{verbatim}

In the error message, the \texttt{FileName} value is the name of the source file and the \texttt{LineNumber} value is the source line number of the \texttt{assert} statement.

Parameters
\begin{itemize}
  \item \texttt{Expression} Specifies an expression that can be evaluated as true or false. This expression is evaluated in the same manner as the C language IF statement.
\end{itemize}
Related Information
The abort subroutine.

The cpp command.

Subroutines Overview in AIX 5L Version 5.3 General Programming Concepts: Writing and Debugging Programs.

atan2f, atan2l, or atan2 Subroutine

Purpose
Computes the arc tangent.

Syntax
#include <math.h>

float atan2f (y, x)
float y, float x;

long double atan2l (y, x)
long double y, long double x;

double atan2 (y, x)
double y, x;

Description
The atan2f, atan2l, and atan2 subroutines compute the principal value of the arc tangent of y/x, using the signs of both parameters to determine the quadrant of the return value.

An application wishing to check for error situations should set the errno global variable to zero and call feclearexcept(FE_ALL_EXCEPT) before calling these functions. On return, if errno is nonzero or fetestexcept(FE_INVALID | FE_DIVBYZERO | FE_OVERFLOW | FE_UNDERFLOW) is nonzero, an error has occurred.

Parameters

y Specifies the value to compute.
x Specifies the value to compute.

Return Values
Upon successful completion, the atan2f, atan2l, and atan2 subroutines return the arc tangent of y/x in the range [-pi, pi] radians.

If y is 0 and x is < 0, ±pi is returned.
If y is 0 and x is > 0, 0 is returned.
If y is < 0 and x is 0, -pi/2 is returned.
If y is > 0 and x is 0, pi/2 is returned.
If x is 0, a pole error does not occur.
If either $x$ or $y$ is NaN, a NaN is returned.

If the result underflows, a range error may occur and $y/x$ is returned.

If $y$ is 0 and $x$ is $-0$, $\pm x$ is returned.

If $y$ is 0 and $x$ is $+0$, 0 is returned.

For finite values of $\pm y > 0$, if $x$ is $-\infty$, $\pm x$ is returned.

For finite values of $\pm y > 0$, if $x$ is $+\infty$, 0 is returned.

For finite values of $x$, if $y$ is $\pm\infty$, $\pm x/2$ is returned.

If $y$ is $\pm\infty$ and $x$ is $-\infty$, $\pm 3\pi/4$ is returned.

If $y$ is $\pm\infty$ and $x$ is $+\infty$, $\pm \pi/4$ is returned.

If both arguments are 0, a domain error does not occur.

**Related Information**

`math.h` in AIX 5L Version 5.3 Files Reference.

---

**atan, atanf, or atanl Subroutine**

**Purpose**

Computes the arc tangent.

**Syntax**

```c
#include <math.h>

float atanf (x)
    float x;

long double atanl (x)
    long double x;

double atan (x)
    double x;
```

**Description**

The `atanf`, `atanl`, and `atan` subroutines compute the principal value of the arc tangent of the $x$ parameter.

An application wishing to check for error situations should set the `errno` global variable to zero and call `feclearexcept(FE_ALL_EXCEPT)` before calling these functions. On return, if `errno` is nonzero or `fetestexcept(FE_INVALID | FE_DIVBYZERO | FE_OVERFLOW | FE_UNDERFLOW)` is nonzero, an error has occurred.

**Parameters**

$x$ Specifies the value to be computed.
Return Values
Upon successful completion, the atanf, atanl, and atan subroutines return the arc tangent of \(x\) in the range \([-\pi/2, \pi/2]\) radians.

If \(x\) is NaN, a NaN is returned.

If \(x\) is 0, \(x\) is returned.

If \(x\) is ±\(\infty\), ±\(x/2\) is returned.

If \(x\) is subnormal, a range error may occur and \(x\) is returned.

Related Information
The “atan2f, atan2l, or atan2 Subroutine” on page 93 and “atanh, atanhf, or atanhl Subroutine.”

math.h in AIX 5L Version 5.3 Files Reference.

atanh, atanhf, or atanhl Subroutine

Purpose
Computes the inverse hyperbolic tangent.

Syntax

```c
#include <math.h>

float atanh (x);
long double atanhl (x);
double atanh (x);
```

Description
The atanhf, atanhl, and atanh subroutines compute the inverse hyperbolic tangent of the \(x\) parameter.

An application wishing to check for error situations should set the errno global variable to zero and call feclearexcept(FE_ALL_EXCEPT) before calling these functions. On return, if errno is nonzero or fetestexcept(FE_INVALID | FE_DIVBYZERO | FE_OVERFLOW | FE_UNDERFLOW) is nonzero, an error has occurred.

Parameters

\(x\) Specifies the value to be computed.

Return Values
Upon successful completion, the atanhf, atanhl, and atanh subroutines return the inverse hyperbolic tangent of the given argument.

If \(x\) is ±1, a pole error occurs, and atanhf, atanhl, and atanh return the value of the macro HUGE_VALF, HUGE_VALL, and HUGE_VAL respectively, with the same sign as the correct value of the function.
For finite \(|x|>1\), a domain error occurs, and a NaN is returned.

If \(x\) is NaN, a NaN is returned.

If \(x\) is 0, \(x\) is returned.

If \(x\) is ±Inf, a domain error shall occur, and a NaN is returned.

If \(x\) is subnormal, a range error may occur and \(x\) is returned.

**Error Codes**
The **atanhf**, **atanhl**, and **atanh** subroutines return **NaNQ** and set **errno** to **EDOM** if the absolute value of \(x\) is greater than 1.

**Related Information**
- “exp, expf, or expl Subroutine” on page 244
- math.h in AIX 5L Version 5.3 Files Reference.
- Subroutines Overview in AIX 5L Version 5.3 General Programming Concepts: Writing and Debugging Programs.

---

**atof atoff Subroutine**

**Purpose**
Converts an ASCII string to a floating-point or double floating-point number.

**Libraries**
Standard C Library (**libc.a**)  

**Syntax**
```c
#include <stdlib.h>
double atof (NumberPointer)
const char *NumberPointer;
float atoff (NumberPointer)
char *NumberPointer;
```

**Description**
The **atof** subroutine converts a character string, pointed to by the **NumberPointer** parameter, to a double-precision floating-point number. The **atoff** subroutine converts a character string, pointed to by the **NumberPointer** parameter, to a single-precision floating-point number. The first unrecognized character ends the conversion.

Except for behavior on error, the **atof** subroutine is equivalent to the **strtod** subroutine call, with the **EndPointer** parameter set to (**char**) \(NULL\).

Except for behavior on error, the **atoff** subroutine is equivalent to the **strtof** subroutine call, with the **EndPointer** parameter set to (**char**) \(NULL\).

These subroutines recognize a character string when the characters are in one of two formats: numbers or numeric symbols.
- For a string to be recognized as a number, it should contain the following pieces in the following order:
1. An optional string of white-space characters
2. An optional sign
3. A nonempty string of digits optionally containing a radix character
4. An optional exponent in E-format or e-format followed by an optionally signed integer.

• For a string to be recognized as a numeric symbol, it should contain the following pieces in the following order:
  1. An optional string of white-space characters
  2. An optional sign
  3. One of the strings: INF, infinity, NaNQ, NaNs, or NaN (case insensitive)

The atoff subroutine is not part of the ANSI C Library. These subroutines are at least as accurate as required by the IEEE Standard for Binary Floating-Point Arithmetic. The atof subroutine accepts at least 17 significant decimal digits. The atoff and subroutine accepts at least 9 leading 0’s. Leading 0’s are not counted as significant digits.

Parameters

NumberPointer Specifies a character string to convert.
EndPoint Specifies a pointer to the character that ended the scan or a null value.

Return Values

Upon successful completion, the atof, and atoff subroutines return the converted value. If no conversion could be performed, a value of 0 is returned and the errno global variable is set to indicate the error.

Error Codes

If the conversion cannot be performed, a value of 0 is returned, and the errno global variable is set to indicate the error.

If the conversion causes an overflow (that is, the value is outside the range of representable values), +/-HUGE_VAL is returned with the sign indicating the direction of the overflow, and the errno global variable is set to ERANGE.

If the conversion would cause an underflow, a properly signed value of 0 is returned and the errno global variable is set to ERANGE.

The atoff subroutine has only one rounding error. (If the atof subroutine is used to create a double-precision floating-point number and then that double-precision number is converted to a floating-point number, two rounding errors could occur.)

Related Information

The scanf subroutine, atol, or atol subroutine, wstrtol,watol, or watoi subroutine.

Subroutines Overview in AIX 5L Version 5.3 General Programming Concepts: Writing and Debugging Programs.

128-Bit long double Floating-Point Format in AIX 5L Version 5.3 General Programming Concepts: Writing and Debugging Programs.
atol or atoll Subroutine

Purpose
Converts a string to a long integer.

Syntax
#include <stdlib.h>

long long atoll (nptr)
const char *nptr;

long atol (nptr)
const char *nptr;

Description
The atol and atol subroutines (str) are equivalent to strtoll(nptr, (char **)NULL, 10) and
strtol(nptr, (char **)NULL, 10), respectively. If the value cannot be represented, the behavior is
undefined.

Parameters

nptr Points to the string to be converted into a long integer.

Return Values
The atol and atol subroutines return the converted value if the value can be represented.

Related Information
strtol, strtoul, strtoll, strtoull, or atoi Subroutine in AIX 5L Version 5.3 Technical Reference: Base Operating
System and Extensions Volume 2.

audit Subroutine

Purpose
Enables and disables system auditing.

Library
Standard C Library (libc.a)

Syntax
#include <sys/audit.h>

int audit (Command, Argument)
int Command;
int Argument;

Description
The audit subroutine enables or disables system auditing.
When auditing is enabled, audit records are created for security-relevant events. These records can be collected through the `auditbin` subroutine, or through the `/dev/audit` special file interface.

### Parameters

**Command**

Defined in the `sys/audit.h` file, can be one of the following values:

- **AUDIT_QUERY**
  Returns a mask indicating the state of the auditing subsystem. The mask is a logical ORing of the AUDIT_ON, AUDIT_OFF, and AUDIT_PANIC flags. The Argument parameter is ignored.

- **AUDIT_ON**
  Enables auditing. If auditing is already enabled, only the failure-mode behavior changes. The Argument parameter specifies recovery behavior in the event of failure and may be either 0 or the value AUDIT_PANIC.
  **Note:** If AUDIT_PANIC is specified, bin-mode auditing must be enabled before the audit subroutine call.

- **AUDIT_OFF**
  Disables the auditing system if auditing is enabled. If the auditing system is disabled, the audit subroutine does nothing. The Argument parameter is ignored.

- **AUDIT_RESET**
  Disables the auditing system (as does AUDIT_OFF) and resets the auditing system. If auditing is already disabled, only the system configuration is reset. Resetting the audit configuration involves clearing the audit events and audited objects table, and terminating bin and stream auditing. The Argument parameter is ignored.

- **AUDIT_EVENT_THRESHOLD**
  Audit event records will be buffered until a total of Argument records have been saved, at which time the audit event records will be flushed to disk. An Argument value of zero disables this functionality. This parameter only applies to AIX 4.1.4 and later.

- **AUDIT_BYTE_THRESHOLD**
  Audit event data will be buffered until a total of Argument bytes of data have been saved, at which time the audit event data will be flushed to disk. An Argument value of zero disables this functionality. This parameter only applies to AIX 4.1.4 and later.

**Argument**

Specifies the behavior when a bin write fails (for AUDIT_ON) or specifies the size of the audit event buffer (for AUDIT_EVENT_THRESHOLD and AUDIT_BYTE_THRESHOLD). For all other commands, the value of Argument is ignored. The valid values are:

- **AUDIT_PANIC**
  The operating system halts abruptly if an audit record cannot be written to a bin.
  **Note:** If AUDIT_PANIC is specified, bin-mode auditing must be enabled before the audit subroutine call.

**BufferSize**

The number of bytes or audit event records which will be buffered. This parameter is valid only with the command AUDIT_BYTE_THRESHOLD and AUDIT_EVENT_THRESHOLD. A value of zero will disable either byte (for AUDIT_BYTE_THRESHOLD) or event (for AUDIT_EVENT_THRESHOLD) buffering.

### Return Values

For a Command value of AUDIT_QUERY, the audit subroutine returns, upon successful completion, a mask indicating the state of the auditing subsystem. The mask is a logical ORing of the AUDIT_ON, AUDIT_OFF, AUDIT_PANIC, and AUDIT_NO_PANIC flags. For any other Command value, the audit subroutine returns 0 on successful completion.
If the `audit` subroutine fails, a value of -1 is returned and the `errno` global variable is set to indicate the error.

**Error Codes**
The `audit` subroutine fails if either of the following is true:

- **EINVAL** The `Command` parameter is not one of `AUDIT_ON`, `AUDIT_OFF`, `AUDIT_RESET`, or `AUDIT_QUERY`.
- **EINVAL** The `Command` parameter is `AUDIT_ON` and the `Argument` parameter specifies values other than `AUDIT_PANIC`.
- **EPERM** The calling process does not have root user authority.

**Files**

- `dev/audit` Specifies the audit pseudo-device from which the audit records are read.

**Related Information**
The `auditbin` subroutine, `auditevents` subroutine, `auditlog` subroutine, `auditobj` subroutine, `auditproc` subroutine.

The `audit` command.

[List of Security and Auditing Subroutines](#) and [Subroutines Overview](#) in AIX 5L Version 5.3 General Programming Concepts: Writing and Debugging Programs.

---

### auditbin Subroutine

**Purpose**
Defines files to contain audit records.

**Library**
Standard C Library (`libc.a`)

**Syntax**
```
#include <sys/audit.h>

int auditbin (Command, Current, Next, Threshold)
int Command;
int Current;
int Next;
int Threshold;
```

**Description**
The `auditbin` subroutine establishes an audit bin file into which the kernel writes audit records. Optionally, this subroutine can be used to establish an overflow bin into which records are written when the current bin reaches the size specified by the `Threshold` parameter.
Parameters

Command
If nonzero, this parameter is a logical ORing of the following values, which are defined in the sys/audit.h file:

AUDIT_EXCL
Requests exclusive rights to the audit bin files. If the file specified by the Current parameter is not the kernel's current bin file, the auditbin subroutine fails immediately with the_errno variable set to EBUSY.

AUDIT_WAIT
The auditbin subroutine should not return until:

bin full
The kernel writes the number of bytes specified by the Threshold parameter to the file descriptor specified by the Current parameter. Upon successful completion, the auditbin subroutine returns a 0. The kernel writes subsequent audit records to the file descriptor specified by the Next parameter.

bin failure
An attempt to write an audit record to the file specified by the Current parameter fails. If this occurs, the auditbin subroutine fails with the_errno variable set to the return code from the auditwrite subroutine.

bin contention
Another process has already issued a successful call to the auditbin subroutine. If this occurs, the auditbin subroutine fails with the_errno variable set to EBUSY.

system shutdown
The auditing system was shut down. If this occurs, the auditbin subroutine fails with the_errno variable set to EINTR.

Current
A file descriptor for a file to which the kernel should immediately write audit records.

Next
Specifies the file descriptor that will be used as the current audit bin if the value of the Threshold parameter is exceeded or if a write to the current bin fails. If this value is -1, no switch occurs.

Threshold
Specifies the maximum size of the current bin. If 0, the auditing subsystem will not switch bins. If it is nonzero, the kernel begins writing records to the file specified by the Next parameter, if writing a record to the file specified by the Cur parameter would cause the size of this file to exceed the number of bytes specified by the Threshold parameter. If no next bin is defined and AUDIT_PANIC was specified when the auditing subsystem was enabled, the system is shut down. If the size of the Threshold parameter is too small to contain a bin header and a bin tail, the auditbin subroutine fails and the_errno variable is set toEINVAL.

Return Values
If the auditbin subroutine is successful, a value of 0 returns.

If the auditbin subroutine fails, a value of -1 returns and the_errno global variable is set to indicate the error. If this occurs, the result of the call does not indicate whether any records were written to the bin.

Error Codes
The auditbin subroutine fails if any of the following is true:

EBADF
The Current parameter is not a file descriptor for a regular file open for writing, or the Next parameter is neither -1 nor a file descriptor for a regular file open for writing.

EBUSY
The Command parameter specifies AUDIT_EXCL and the kernel is not writing audit records to the file specified by the Current parameter.

EBUSY
The Command parameter specifies AUDIT_WAIT and another process has already registered a bin.

EINTR
The auditing subsystem is shut down.
EINVAL The Command parameter specifies a nonzero value other than AUDIT_EXCL or AUDIT_WAIT.
EINVAL The Threshold parameter value is less than the size of a bin header and trailer.
EPERM The caller does not have root user authority.

Related Information
The audit subroutine, auditevents subroutine, auditlog subroutine, auditobj subroutine, auditproc subroutine.
The audit command.
The audit file format.
List of Security and Auditing Subroutines and Subroutines Overview in AIX 5L Version 5.3 General Programming Concepts: Writing and Debugging Programs.

auditevents Subroutine

Purpose
Gets or sets the status of system event auditing.

Library
Standard C Library (libc.a)

Syntax
#include <sys/audit.h>

int auditevents (Command, Classes, NClasses)
int Command;
struct audit_class *Classes;
int NClasses;

Description
The auditevents subroutine queries or sets the audit class definitions that control event auditing. Each audit class is a set of one or more audit events.

System auditing need not be enabled before calling the auditevents subroutine. The audit subroutine can be directed with the AUDIT_RESET command to clear all event lists.
Parameters

**Command**

Specifies whether the event lists are to be queried or set. The values, defined in the `sys/audit.h` file, for the *Command* parameter are:

**AUDIT_SET**

Sets the lists of audited events after first clearing all previous definitions.

**AUDIT_GET**

Queries the lists of audited events.

**AUDIT_LOCK**

Queries the lists of audited events. This value also blocks any other process attempting to set or lock the list of audit events. The lock is released when the process holding the lock dies or calls the `auditevents` subroutine with the *Command* parameter set to *AUDIT_SET*.

**Classes**

Specifies the array of `a_event` structures for the *AUDIT_SET* operation, or after an *AUDIT_GET* or *AUDIT_LOCK* operation. The `audit_class` structure is defined in the `sys/audit.h` file and contains the following members:

- **ae_name**
  
  A pointer to the name of the audit class.

- **ae_list**
  
  A pointer to a list of null-terminated audit event names for this audit class. The list is ended by a null name (a leading null byte or two consecutive null bytes).
  
  **Note:** Event and class names are limited to 15 significant characters.

- **ae_len**
  
  The length of the event list in the *ae_list* member. This length includes the terminating null bytes. On an *AUDIT_SET* operation, the caller must set this member to indicate the actual length of the list (in bytes) pointed to by *ae_list*. On an *AUDIT_GET* or *AUDIT_LOCK* operation, the `auditevents` subroutine sets this member to indicate the actual size of the list.

**NClasses**

Serves a dual purpose. For *AUDIT_SET*, the *NClasses* parameter specifies the number of elements in the events array. For *AUDIT_GET* and *AUDIT_LOCK*, the *NClasses* parameter specifies the size of the buffer pointed to by the *Classes* parameter.

**Attention:** Only 32 audit classes are supported. One class is implicitly defined by the system to include all audit events (ALL). The administrator of your system should not attempt to define more than 31 audit classes.

Security

The calling process must have root user authority in order to use the `auditevents` subroutine.

Return Codes

If the `auditevents` subroutine completes successfully, the number of audit classes is returned if the *Command* parameter is *AUDIT_GET* or *AUDIT_LOCK*. A value of 0 is returned if the *Command* parameter is *AUDIT_SET*. If this call fails, a value of -1 is returned and the *errno* global variable is set to indicate the error.

Error Codes

The `auditevents` subroutine fails if one or more of the following are true:

- **EPERM**
  
  The calling process does not have root user authority.

- **EINVAL**
  
  The value of *Command* is not *AUDIT_SET*, *AUDIT_GET*, or *AUDIT_LOCK*.

- **EINVAL**
  
  The *Command* parameter is *AUDIT_SET*, and the value of the *NClasses* parameter is greater than or equal to 32.

- **EINVAL**
  
  A class name or event name is longer than 15 significant characters.
ENOSPC The value of Command is AUDIT_GET or AUDIT_LOCK and the size of the buffer specified by the NClasses parameter is not large enough to hold the list of event structures and names. If this occurs, the first word of the buffer is set to the required buffer size.

EFAULT The Classes parameter points outside of the process’ address space.

EFAULT The ae_list member of one or more audit_class structures passed for an AUDIT_SET operation points outside of the process’ address space.

EFAULT The Command value is AUDIT_GET or AUDIT_LOCK and the size of the Classes buffer is not large enough to hold an integer.

EBUSY Another process has already called the auditevents subroutine with AUDIT_LOCK.

ENOMEM Memory allocation failed.

Related Information
The audit subroutine, auditbin subroutine, auditlog subroutine, auditobj subroutine, auditproc subroutine, auditread, auditread_r Subroutines, auditwrite subroutine.

The audit command.

List of Security and Auditing Subroutines and Subroutines Overview in AIX 5L Version 5.3 General Programming Concepts: Writing and Debugging Programs.

auditlog Subroutine

Purpose
Appends an audit record to the audit trail file.

Library
Standard C Library (libc.a)

Syntax
#include <sys/audit.h>

int auditlog (Event, Result, Buffer, BufferSize)
char *Event;
int Result;
char *Buffer;
int BufferSize;

Description
The auditlog subroutine generates an audit record. The kernel audit-logging component appends a record for the specified Event if system auditing is enabled, process auditing is not suspended, and the Event parameter is in one or more of the audit classes for the current process.

The audit logger generates the audit record by adding the Event and Result parameters to the audit header and including the resulting information in the Buffer parameter as the audit tail.

Parameters
Event The name of the audit event to be generated. This parameter should be the name of an audit event. Audit event names are truncated to 15 characters plus null.
Result

Describes the result of this event. Valid values are defined in the `sys/audit.h` file and include the following:

- **AUDIT_OK**
  - The event was successful.

- **AUDIT_FAIL**
  - The event failed.

- **AUDIT_FAIL_ACCESS**
  - The event failed because of any access control denial.

- **AUDIT_FAIL_DAC**
  - The event failed because of a discretionary access control denial.

- **AUDIT_FAIL_PRIV**
  - The event failed because of a privilege control denial.

- **AUDIT_FAIL_AUTH**
  - The event failed because of an authentication denial.

Other nonzero values of the `Result` parameter are converted into the **AUDIT_FAIL** value.

Buffer

Points to a buffer containing the tail of the audit record. The format of the information in this buffer depends on the event name.

BufferSize

Specifies the size of the `Buffer` parameter, including the terminating null.

Return Values

Upon successful completion, the `auditlog` subroutine returns a value of 0. If `auditlog` fails, a value of -1 is returned and the `errno` global variable is set to indicate the error.

The `auditlog` subroutine does not return any indication of failure to write the record where this is due to inappropriate tailoring of auditing subsystem configuration files or user-written code. Accidental omissions and typographical errors in the configuration are potential causes of such a failure.

Error Codes

The `auditlog` subroutine fails if any of the following are true:

- **EFAULT**
  - The `Event` or `Buffer` parameter points outside of the process’ address space.

- **EINVAL**
  - The auditing system is either interrupted or not initialized.

- **EINVAL**
  - The length of the audit record is greater than 32 kilobytes.

- **EPERM**
  - The process does not have root user authority.

- **ENOMEM**
  - Memory allocation failed.

Related Information

The [audit Subroutine](#) subroutine, [auditbin Subroutine](#) subroutine, [auditevents Subroutine](#) subroutine, [auditobj Subroutine](#) subroutine, [auditproc Subroutine](#) subroutine, [auditwrite Subroutine](#) subroutine.

List of Security and Auditing Subroutines and Subroutines Overview in AIX 5L Version 5.3 General Programming Concepts: Writing and Debugging Programs.

**auditobj Subroutine**

**Purpose**

Gets or sets the auditing mode of a system data object.
Library
Standard C Library (libc.a)

Syntax
#include <sys/audit.h>

int auditobj (Command, Obj_Events, ObjSize);
int Command;
struct o_event *Obj_Events;
int ObjSize;

Description
The auditobj subroutine queries or sets the audit events to be generated by accessing selected objects. For each object in the file system name space, it is possible to specify the event generated for each access mode. Using the auditobj subroutine, an administrator can define new audit events in the system that correspond to accesses to specified objects. These events are treated the same as system-defined events.

System auditing need not be enabled to set or query the object audit events. The audit subroutine can be directed with the AUDIT_RESET command to clear the definitions of object audit events.

Parameters

Command

Specifies whether the object audit event lists are to be read or written. The valid values, defined in the sys/audit.h file, for the Command parameter are:

AUDIT_SET
Sets the list of object audit events, after first clearing all previous definitions.

AUDIT_GET
Queries the list of object audit events.

AUDIT_LOCK
Queries the list of object audit events and also blocks any other process attempting to set or lock the list of audit events. The lock is released when the process holding the lock dies or calls the auditobj subroutine with the Command parameter set to AUDIT_SET.
**Obj_Events**

Specifies the array of *o_event* structures for the **AUDIT_SET** operation or for after the **AUDIT_GET** or **AUDIT_LOCK** operation. The **o_event** structure is defined in the **sys/audit.h** file and contains the following members:

- **o_type**: Specifies the type of the object, in terms of naming space. Currently, only one object-naming space is supported:
  
  **AUDIT_FILE**
  
  Denotes the file system naming space.

- **o_name**: Specifies the name of the object.

- **o_event**: Specifies any array of event names to be generated when the object is accessed. Note that event names are currently limited to 16 bytes, including the trailing null. The index of an event name in this array corresponds to an access mode. Valid indexes are defined in the **audit.h** file and include the following:
  
  - **AUDIT_READ**
  - **AUDIT_WRITE**
  - **AUDIT_EXEC**

**Note**: The C++ compiler will generate a warning indicating that **o_event** is defined both as a structure and a field within that structure. Although the **o_event** field can be used within C++, the warning can by bypassed by defining **O_EVENT_RENAME**. This will replace the **o_event** field with **o_event_array**. **o_event** is the default field.

**ObjSize**

For an **AUDIT_SET** operation, the **ObjSize** parameter specifies the number of object audit event definitions in the array pointed to by the **Obj_Events** parameter. For an **AUDIT_GET** or **AUDIT_LOCK** operation, the **ObjSize** parameter specifies the size of the buffer pointed to by the **Obj_Events** parameter.

**Return Values**

If the **auditobj** subroutine completes successfully, the number of object audit event definitions is returned if the **Command** parameter is **AUDIT_GET** or **AUDIT_LOCK**. A value of 0 is returned if the **Command** parameter is **AUDIT_SET**. If this call fails, a value of -1 is returned and the **errno** global variable is set to indicate the error.

**Error Codes**

The **auditobj** subroutine fails if any of the following are true:

- **EFAULT**: The **Obj_Events** parameter points outside the address space of the process.
- **EFAULT**: The **Command** parameter is **AUDIT_SET**, and one or more of the **o_name** members points outside the address space of the process.
- **EFAULT**: The **Command** parameter is **AUDIT_GET** or **AUDIT_LOCK**, and the buffer size of the **Obj_Events** parameter is not large enough to hold the integer.
- **EINVAL**: The value of the **Command** parameter is not **AUDIT_SET**, **AUDIT_GET** or **AUDIT LOCK**.
- **EINVAL**: The **Command** parameter is **AUDIT_SET**, and the value of one or more of the **o_type** members is not **AUDIT_FILE**.
- **EINVAL**: An event name was longer than 15 significant characters.
- **ENOENT**: The **Command** parameter is **AUDIT_SET**, and the parent directory of one of the file-system objects does not exist.
- **ENOSPC**: The value of the **Command** parameter is **AUDIT_GET** or **AUDIT LOCK**, and the size of the buffer as specified by the **ObjSize** parameter is not large enough to hold the list of event structures and names. If this occurs, the first word of the buffer is set to the required buffer size.
- **ENOMEM**: Memory allocation failed.
- **EBUSY**: Another process has called the **auditobj** subroutine with **AUDIT LOCK**.
Related Information

The audit subroutine, auditbin subroutine, auditevents subroutine, auditlog subroutine, auditevents subroutine, auditproc subroutine.

The audit command.

The audit.h file.

List of Security and Auditing Subroutines and Subroutines Overview in AIX 5L Version 5.3 General Programming Concepts: Writing and Debugging Programs.

auditpack Subroutine

Purpose
Compresses and uncompresses audit bins.

Library
Security Library (libc.a)

Syntax

```
#include <sys/audit.h>
#include <stdio.h>

char *auditpack (Expand, Buffer)
int Expand;
char *Buffer;
```

Description
The auditpack subroutine can be used to compress or uncompress bins of audit records.

Parameters

*Expand* Specifies the operation. Valid values, as defined in the sys/audit.h header file, are one of the following:

  AUDIT_PACK
  Performs standard compression on the audit bin.

  AUDIT_UNPACK
  Unpacks the compressed audit bin.

*Buffer* Specifies the buffer containing the bin to be compressed or uncompressed. This buffer must contain a standard bin as described in the audit.h file.

Return Values

If the auditpack subroutine is successful, a pointer to a buffer containing the processed audit bin is returned. If unsuccessful, a null pointer is returned and the errno global variable is set to indicate the error.
Error Codes

The auditpack subroutine fails if one or more of the following values is true:

EINVAL
The Expand parameter is not one of the valid values (AUDIT_PACK or AUDIT_UNPACK).
EINVAL
The Expand parameter is AUDIT_UNPACK and the packed data in Buffer does not unpack to its original size.
EINVAL
The Expand parameter is AUDIT_PACK and the bin in the Buffer parameter is already compressed, or the Expand parameter is AUDIT_UNPACK and the bin in the Buffer parameter is already unpacked.
ENOSPC
The auditpack subroutine is unable to allocate space for a new buffer.

Related Information

The auditread subroutine is unable to allocate space for a new buffer.

List of Security and Auditing Subroutines and Subroutines Overview in AIX 5L Version 5.3 General Programming Concepts: Writing and Debugging Programs.

auditproc Subroutine

Purpose
Gets or sets the audit state of a process.

Library
Standard C Library (libc.a)

Syntax

```
#include <sys/audit.h>

int auditproc (ProcessID, Command, Argument, Length)
```

```
int ProcessID;
int Command;
char * Argument;
int Length;
```

Description

The auditproc subroutine queries or sets the auditing state of a process. There are two parts to the auditing state of a process:

- The list of classes to be audited for this process. Classes are defined by the auditevents subroutine. Each class includes a set of audit events. When a process causes an audit event, that event may be logged in the audit trail if it is included in one or more of the audit classes of the process.
- The audit status of the process. Auditing for a process may be suspended or resumed. Functions that generate an audit record can first check to see whether auditing is suspended. If process auditing is suspended, no audit events are logged for a process. For more information, see the auditlog subroutine.
Parameters

ProcessID  The process ID of the process to be affected. If ProcessID is 0, the auditproc subroutine affects the current process.

Command   The action to be taken. Defined in the audit.h file, valid values include:

AUDIT_KLIST_EVENTS  Sets the list of audit classes to be audited for the process and also sets the user’s default audit classes definition within the kernel. The Argument parameter is a pointer to a list of null-terminated audit class names. The Length parameter is the length of this list, including null bytes.

AUDIT_QEVENTS  Returns the list of audit classes defined for the current process if ProcessID is 0. Otherwise, it returns the list of audit classes defined for the specified process ID. The Argument parameter is a pointer to a character buffer. The Length parameter specifies the size of this buffer. On return, this buffer contains a list of null-terminated audit class names. A null name terminates the list.

AUDIT_EVENTS  Sets the list of audit classes to be audited for the process. The Argument parameter is a pointer to a list of null-terminated audit class names. The Length parameter is the length of this list, including null bytes.

AUDIT_QSTATUS  Returns the audit status of the current process. You can only check the status of the current process. If the ProcessID parameter is nonzero, a -1 is returned and the errno global variable is set to EINVAL. The Length and Argument parameters are ignored. A return value of AUDIT_SUSPEND indicates that auditing is suspended. A return value of AUDIT_RESUME indicates normal auditing for this process.

Argument    A character pointer for the audit class buffer for an AUDIT_EVENT or AUDIT_QEVENTS value of the Command parameter or an integer defining the audit status to be set for an AUDIT_STATUS operation.

Length      Size of the audit class character buffer.

Return Values

The auditproc subroutine returns the following values upon successful completion:

- The previous audit status (AUDIT_SUSPEND or AUDIT_RESUME), if the call queried or set the audit status (the Command parameter specified AUDIT_QSTATUS or AUDIT_STATUS)
- A value of 0 if the call queried or set audit events (the Command parameter specified AUDIT_QEVENTS or AUDIT_EVENTS)

Error Codes

If the auditproc subroutine fails if one or more of the following are true:

EINVAL    An invalid value was specified for the Command parameter.
EINVAL    The Command parameter is set to the AUDIT_QSTATUS or AUDIT_STATUS value and the pid value is nonzero.
EINVAL    The Command parameter is set to the AUDIT_STATUS value and the Argument parameter is not set to AUDIT_SUSPEND or AUDIT_RESUME.
ENOSPC     The Command parameter is AUDIT_QEVENTS, and the buffer size is insufficient. In this case, the first word of the Argument parameter is set to the required size.
The Command parameter is AUDIT_QEVENTS or AUDIT_EVENTS and the Argument parameter points to a location outside of the process’ allocated address space.

ENOMEM
Memory allocation failed.

EPERM
The caller does not have root user authority.

Related Information
The audit subroutine, auditbin subroutine, auditevents subroutine, auditlog subroutine, auditevents subroutine, auditobj subroutine, auditwrite subroutine.

List of Security and Auditing Subroutines and Subroutines Overview in AIX 5L Version 5.3 General Programming Concepts: Writing and Debugging Programs.

auditread, auditread_r Subroutines

Purpose
Reads an audit record.

Library
Security Library (libc.a)

Syntax
```c
#include <sys/audit.h>
#include <stdio.h>
char *auditread (FILE *FilePointer, struct aud_rec *AuditRecord);

FILE *FilePointer;
struct aud_rec *AuditRecord;

char *auditread_r (FILE *FilePointer, struct aud_rec *AuditRecord, size_t RecordSize, void **StreamInfo);

FILE *FilePointer;
struct aud_rec *AuditRecord;
size_t RecordSize;
void **StreamInfo;
```

Description
The auditread subroutine reads the next audit record from the specified file descriptor. Bins on this input stream are unpacked and uncompressed if necessary.

The auditread subroutine can not be used on more than one FilePointer as the results can be unpredictable. Use the auditread_r subroutine instead.

The auditread_r subroutine reads the next audit from the specified file descriptor. This subroutine is thread safe and can be used to handle multiple open audit files simultaneously by multiple threads of execution.

The auditread_r subroutine is able to read multiple versions of audit records. The version information contained in an audit record is used to determine the correct size and format of the record. When an input record header is larger than AuditRecord, an error is returned. In order to provide for binary compatibility with previous versions, if RecordSize is the same size as the original (struct aud_rec), the input record is converted to the original format and returned to the caller.
Parameters

- **FilePointer**: Specifies the file descriptor from which to read.
- **AuditRecord**: Specifies the buffer to contain the header. The first short in this buffer must contain a valid number for the header.
- **RecordSize**: The size of the buffer referenced by **AuditRecord**.
- **StreamInfo**: A pointer to an opaque datatype used to hold information related to the current value of **FilePointer**. For each new value of **FilePointer**, a new **StreamInfo** pointer must be used. **StreamInfo** must be initialized to NULL by the user and is initialized by **auditread_r** when first used. When **FilePointer** has been closed, the value of **StreamInfo** can be passed to the **free** subroutine to be deallocated.

Return Values

If the **auditread** subroutine completes successfully, a pointer to a buffer containing the tail of the audit record is returned. The length of this buffer is returned in the ah_length field of the header file. If this subroutine is unsuccessful, a null pointer is returned and the **errno** global variable is set to indicate the error.

Error Codes

The **auditread** subroutine fails if one or more of the following is true:

- **EBADF**: The **FilePointer** value is not valid.
- **ENOSPC**: The **auditread** subroutine is unable to allocate space for the tail buffer.

Other error codes are returned by the **read** subroutine.

Related Information

The **auditpack** subroutine.

List of Security and Auditing Subroutines and Subroutines Overview in AIX 5L Version 5.3 General Programming Concepts: Writing and Debugging Programs.

**auditwrite Subroutine**

**Purpose**

Writes an audit record.

**Library**

Security Library (**libc.a**)

**Syntax**

```c
#include <sys/audit.h>
#include <stdio.h>

int auditwrite (Event, Result, Buffer1, Length1, Buffer2, Length2, ...
```

```c
char * Event;
int Result;
char * Buffer1, *Buffer2 ...;
int Length1, Length2 ...;
```
Description

The `auditwrite` subroutine builds the tail of an audit record and then writes it with the `auditlog` subroutine. The tail is built by gathering the specified buffers. The last buffer pointer must be a null.

If the `auditwrite` subroutine is to be called from a program invoked from the `inittab` file, the `setpcred` subroutine should be called first to establish the process’ credentials.

Parameters

- **Event**
  Specifies the name of the event to be logged.

- **Result**
  Specifies the audit status of the event. Valid values are defined in the `sys/audit.h` file and are listed in the `auditlog` subroutine.

- **Buffer1, Buffer2**
  Specifies the character buffers containing audit tail information. Note that numerical values must be passed by reference. The correct size can be computed with the `sizeof` C function.

- **Length1, Length2**
  Specifies the lengths of the corresponding buffers.

Return Values

If the `auditwrite` subroutine completes successfully, a value of 0 is returned. Otherwise, a value of -1 is returned and the `errno` global variable is set to indicate the error.

Error Codes

The `auditwrite` subroutine fails if the following is true:

- `ENOSPC` The `auditwrite` subroutine is unable to allocate space for the tail buffer.

Other error codes are returned by the `auditlog` subroutine.

Related Information

The `auditlog` subroutine, `setpcred` subroutine.

The `inittab` file.

List of Security and Auditing Subroutines and Subroutines Overview in AIX 5L Version 5.3 General Programming Concepts: Writing and Debugging Programs.

authenticate Subroutine

Purpose

Verifies a user’s name and password.

Library

Security Library (libc.a)

Syntax

```c
#include <usersec.h>

int authenticate (char *UserName, char *Response, int *Reenter, char **Message);
```

authenticate subroutine

```
int authenticate (char *UserName, char *Response, int *Reenter, char **Message);
```
Description
The authenticate subroutine maintains requirements users must satisfy to be authenticated to the system. It is a recallable interface that prompts for the user’s name and password. The user must supply a character string at the prompt issued by the Message parameter. The Response parameter returns the user’s response to the authenticate subroutine. The calling program makes no assumptions about the number of prompt messages the user must satisfy for authentication.

The Reenter parameter indicates when a user has satisfied all prompt messages. The parameter remains nonzero until a user has passed all prompts. After the returned value of Reenter is 0, the return code signals whether authentication has succeeded or failed. When progressing through prompts for a user, the value of Reenter must be maintained by the caller between invocations of authenticate.

The authenticate subroutine ascertains the authentication domains the user can attempt. The subroutine reads the SYSTEM line from the user’s stanza in the /etc/security/user file. Each token that appears in the SYSTEM line corresponds to a method that can be dynamically loaded and processed. Likewise, the system can provide multiple or alternate authentication paths.

The authenticate routine maintains internal state information concerning the next prompt message presented to the user. If the calling program supplies a different user name before all prompts are complete for the user, the internal state information is reset and prompt messages begin again. The calling program maintains the value of the Reenter parameter while processing prompts for a given user.

If the user has no defined password, or the SYSTEM grammar explicitly specifies no authentication required, the user is not required to respond to any prompt messages. Otherwise, the user is always initially prompted to supply a password.

The authenticate subroutine can be called initially with the cleartext password in the Response parameter. If the user supplies a password during the initial invocation but does not have a password, authentication fails. If the user wants the authenticate subroutine to supply a prompt message, the Response parameter is a null pointer on initial invocation.

The authenticate subroutine sets the AUTHSTATE environment variable used by name resolution subroutines, such as the getpwnam subroutine. This environment variable indicates the registry to which user authenticated. Values for the AUTHSTATE environment variable include DCE, compat, and token names that appear in a SYSTEM grammar. A null value can exist if the cron daemon or other utilities that do not require authentication is called.

Parameters

UserName Points to the user’s name that is to be authenticated.
Response Specifies a character string containing the user’s response to an authentication prompt.
Reenter Points to a Boolean value that signals whether the authenticate subroutine has completed processing. If the Reenter parameter is a nonzero value, the authenticate subroutine expects the user to satisfy the prompt message provided by the Message parameter. If the Reenter parameter is 0, the authenticate subroutine has completed processing.
Message Points to a pointer that the authenticate subroutine allocates memory for and fills in. This string is suitable for printing and issues prompt messages (if the Reenter parameter is a nonzero value). It also issues informational messages such as why the user failed authentication (if the Reenter parameter is 0). The calling application is responsible for freeing this memory.

Return Values
Upon successful completion, the authenticate subroutine returns a value of 0. If this subroutine fails, it returns a value of 1.
Error Codes
The authenticate subroutine is unsuccessful if one of the following values is true:

<table>
<thead>
<tr>
<th>Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ENOENT</td>
<td>Indicates that the user is unknown to the system.</td>
</tr>
<tr>
<td>ESAD</td>
<td>Indicates that authentication is denied.</td>
</tr>
<tr>
<td>EINVAL</td>
<td>Indicates that the parameters are not valid.</td>
</tr>
<tr>
<td>ENOMEM</td>
<td>Indicates that memory allocation (malloc) failed.</td>
</tr>
</tbody>
</table>

Note: The DCE mechanism requires credentials on successful authentication that apply only to the authenticate process and its children.

Related Information
The ckuserID subroutine.

authenticatex Subroutine

Purpose
Verifies a user’s name and password.

Library
Security Library (libc.a)

Syntax
```c
#include <usersec.h>

int authenticatex (UserName, Response, Reenter, Message, State)
char *UserName;
char *Response;
int *Reenter;
char **Message;
void **State;
```

Description
The authenticatex subroutine maintains requirements that users must satisfy to be authenticated to the system. It is a recallable interface that prompts for the user’s name and password. The user must supply a character string at the prompt issued by the Message parameter. The Response parameter returns the user’s response to the authenticatex subroutine. The calling program makes no assumptions about the number of prompt messages the user must satisfy for authentication. The authenticatex subroutine maintains information about the results of each part of the authentication process in the State parameter. This parameter can be shared with the chpassx, loginrestrictionsx and passwdexpiredx subroutines. The proper sequence of library routines for authenticating a user in order to create a new session is:

1. Call the loginrestrictionsx subroutine to determine which administrative domains allow the user to log in.
2. Call the authenticatex subroutine to perform authentication using those administrative domains that grant login access.
3. Call the passwdexpiredx subroutine to determine if any of the passwords used during the authentication process have expired and must be changed in order for the user to be granted access.
4. If the passwdexpiredx subroutine indicated that one or more passwords have expired and must be changed by the user, call the chpassx subroutine to update all of the passwords that were used for the authentication process.
The Reenter parameter remains a nonzero value until the user satisfies all prompt messages or answers incorrectly. When the Reenter parameter is 0, the return code signals whether authentication passed or failed. The value of the Reenter parameter must be 0 on the initial call. A nonzero value for the Reenter parameter must be passed to the authenticatex subroutine on subsequent calls. A new authentication can be begun by calling the authenticatex subroutine with a 0 value for the Reenter parameter or by using a different value for UserName.

The State parameter contains information about the authentication process. The State parameter from an earlier call to loginrestrictionsx can be used to control how authentication is performed. Administrative domains that do not permit the user to log in cause those administrative domains to be ignored during authentication even if the user has the correct authentication information.

The authenticatex subroutine ascertains the authentication domains the user can attempt. The subroutine uses the SYSTEM attribute for the user. Each token that is displayed in the SYSTEM line corresponds to a method that can be dynamically loaded and processed. Likewise, the system can provide multiple or alternate authentication paths.

The authenticatex subroutine maintains internal state information concerning the next prompt message presented to the user. If the calling program supplies a different user name before all prompts are complete for the user, the internal state information is reset and prompt messages begin again. The authenticatex subroutine requires that the State parameter be initialized to reference a null value when changing user names or that the State parameter from an earlier call to loginrestrictionsx for the new user be provided.

If the user has no defined password, or the SYSTEM grammar explicitly specifies no authentication required, the user is not required to respond to any prompt messages. Otherwise, the user is always initially prompted to supply a password.

The authenticatex subroutine can be called initially with the cleartext password in the Response parameter. If the user supplies a password during the initial invocation but does not have a password, authentication fails. If the user wants the authenticatex subroutine to supply a prompt message, the Response parameter is a null pointer on initial invocation.

The authenticatex subroutine sets the AUTHSTATE environment variable used by name resolution subroutines, such as the getpwnam subroutine. This environment variable indicates the first registry to which the user authenticated. Values for the AUTHSTATE environment variable include DCE, compat, and token names that appear in a SYSTEM grammar. A null value can exist if the cron daemon or another utility that does not require authentication is called.

Parameters

Message Points to a pointer that the authenticatex subroutine allocates memory for and fills in. This string is suitable for printing and issues prompt messages (if the Reenter parameter is a nonzero value). It also issues informational messages, such as why the user failed authentication (if the Reenter parameter is 0). The calling application is responsible for freeing this memory.

Reenter Points to an integer value that signals whether the authenticatex subroutine has completed processing. If the integer referenced by the Reenter parameter is a nonzero value, the authenticatex subroutine expects the user to satisfy the prompt message provided by the Message parameter. If the integer referenced by the Reenter parameter is 0, the authenticatex subroutine has completed processing. The initial value of the integer referenced by Reenter must be 0 when the authenticatex function is initially invoked and must not be modified by the calling application until the authenticationx subroutine has completed processing.

Response Specifies a character string containing the user’s response to an authentication prompt.
State  Points to a pointer that the **authenticatex** subroutine allocates memory for and fills in. The State parameter can also be the result of an earlier call to the **loginrestrictionsx** subroutine. This parameter contains information about the results of the authentication process for each term in the user’s **SYSTEM** attribute. The calling application is responsible for freeing this memory when it is no longer needed for a subsequent call to the **passwdexpiredx** or **chpassx** subroutines.

*UserName*  Points to the user’s name that is to be authenticated.

**Return Values**
Upon successful completion, the **authenticatex** subroutine returns a value of 0. If this subroutine fails, it returns a value of 1.

**Error Codes**
The **authenticatex** subroutine is unsuccessful if one of the following values is true:

- **EINVAL**  The parameters are not valid.
- **ENOENT**  The user is unknown to the system.
- **ENOMEM**  Memory allocation (malloc) failed.
- **ESAD**  Authentication is denied.

**Note:**  Additional information about the behavior of a loadable authentication module can be found in the documentation for that module.

**Related Information**

---

**basename Subroutine**

**Purpose**
Return the last element of a path name.

**Library**
Standard C Library (**libc.a**)

**Syntax**
```
#include <libgen.h>

char *basename (char *path)
```

**Description**
Given a pointer to a character string that contains a path name, the **basename** subroutine deletes trailing “/” characters from *path*, and then returns a pointer to the last component of *path*. The “/” character is defined as trailing if it is not the first character in the string.

If *path* is a null pointer or points to an empty string, a pointer to a static constant “.” is returned.

**Return Values**
The **basename** function returns a pointer to the last component of *path*. 
The **basename** function returns a pointer to a static constant "." if *path* is a null pointer or points to an empty string.

The **basename** function may modify the string pointed to by *path* and may return a pointer to static storage that may then be overwritten by a subsequent call to the **basename** subroutine.

### Examples

<table>
<thead>
<tr>
<th>Input string</th>
<th>Output string</th>
</tr>
</thead>
<tbody>
<tr>
<td>&quot;/usr/lib&quot;</td>
<td>&quot;lib&quot;</td>
</tr>
<tr>
<td>&quot;/usr/&quot;</td>
<td>&quot;usr&quot;</td>
</tr>
<tr>
<td>&quot;/&quot;</td>
<td>&quot;/&quot;</td>
</tr>
</tbody>
</table>

### Related Information

The **dirname** (**dirname Subroutine** on page 213) subroutine.

---

**bcopy, bcmp, bzero or ffs Subroutine**

#### Purpose
Performs bit and byte string operations.

#### Library
Standard C Library (**libc.a**)

#### Syntax

```c
#include <strings.h>

void bcopy (Source, Destination, Length)
const void *Source,
char *Destination;
size_t Length;

int bcmp (String1, String2, Length)
const void *String1, *String2;
size_t Length;

void bzero (String, Length)
char *String;
int Length;

int ffs (Index)
int Index;
```

#### Description

**Note:** The **bcopy** subroutine takes parameters backwards from the **strcpy** subroutine.

The **bcopy**, **bcmp**, and **bzero** subroutines operate on variable length strings of bytes. They do not check for null bytes as do the **string** routines.

The **bcopy** subroutine copies the value of the *Length* parameter in bytes from the string in the *Source* parameter to the string in the *Destination* parameter.
The **bcmp** subroutine compares the byte string in the *String1* parameter against the byte string of the *String2* parameter, returning a zero value if the two strings are identical and a nonzero value otherwise. Both strings are assumed to be *Length* bytes long.

The **bzero** subroutine zeroes out the string in the *String* parameter for the value of the *Length* parameter in bytes.

The **ffs** subroutine finds the first bit set in the *Index* parameter passed to it and returns the index of that bit. Bits are numbered starting at 1. A return value of 0 indicates that the value passed is 0.

**Related Information**


List of String Manipulation Subroutines and Subroutines, Example Programs, and Libraries in AIX 5L Version 5.3 General Programming Concepts: Writing and Debugging Programs.

---

**bessel: j0, j1, jn, y0, y1, or yn Subroutine**

**Purpose**

Computes Bessel functions.

**Libraries**

IEEE Math Library (**libm.a**)  
or System V Math Library (**libmsaa.a**)  

**Syntax**

```c
#include <math.h>

double j0 (x)
    double x;

double j1 (x)
    double x;

double jn (n, x)
    int n;
    double x;

double y0 (x)
    double x;

double y1 (x)
    double x;

double yn (n, x)
    int n;
    double x;
```

**Description**

Bessel functions are used to compute wave variables, primarily in the field of communications.

The **j0** subroutine and **j1** subroutine return Bessel functions of *x* of the first kind, of orders 0 and 1, respectively. The **jn** subroutine returns the Bessel function of *x* of the first kind of order *n*.  

Base Operating System (BOS) Runtime Services (A-P) 119
The \( y_0 \) subroutine and \( y_1 \) subroutine return the Bessel functions of \( x \) of the second kind, of orders 0 and 1, respectively. The \( y_n \) subroutine returns the Bessel function of \( x \) of the second kind of order \( n \). The value of \( x \) must be positive.

**Note:** Compile any routine that uses subroutines from the \texttt{libm.a} library with the \texttt{-lm} flag. To compile the \texttt{j0.c} file, for example:
```
cc j0.c -lm
```

**Parameters**

\( x \)  
Specifies some double-precision floating-point value.

\( n \)  
Specifies some integer value.

**Return Values**

When using \texttt{libm.a} (\texttt{-lm}), if \( x \) is negative, \( y_0 \), \( y_1 \), and \( y_n \) return the value NaNQ. If \( x \) is 0, \( y_0 \), \( y_1 \), and \( y_n \) return the value \texttt{-HUGE VAL}.

When using \texttt{libmsaa.a} (\texttt{-lmsaa}), values too large in magnitude cause the functions \( j_0 \), \( j_1 \), \( y_0 \), and \( y_1 \) to return 0 and to set the \texttt{errno} global variable to ERANGE. In addition, a message indicating TLOSS error is printed on the standard error output.

Nonpositive values cause \( y_0 \), \( y_1 \), and \( y_n \) to return the value \texttt{-HUGE} and to set the \texttt{errno} global variable to \texttt{EDOM}. In addition, a message indicating argument DOMAIN error is printed on the standard error output.

These error-handling procedures may be changed with the \texttt{matherr} subroutine when using \texttt{libmsaa.a} (\texttt{-lmsaa}).

**Related Information**

The \texttt{matherr} \texttt{“matherr Subroutine” on page 780} subroutine.

**bindprocessor Subroutine**

**Purpose**

Binds kernel threads to a processor.

**Library**

Standard C library (\texttt{libc.a})

**Syntax**

```
#include <sys/processor.h>

int bindprocessor ( \texttt{What}, \texttt{Who}, \texttt{Where} )
int \texttt{What};
int \texttt{Who};
cpu_t \texttt{Where};
```
Description
The **bindprocessor** subroutine binds a single kernel thread, or all kernel threads in a process, to a processor, forcing the bound threads to be scheduled to run on that processor. It is important to understand that a process itself is not bound, but rather its kernel threads are bound. Once kernel threads are bound, they are always scheduled to run on the chosen processor, unless they are later unbound. When a new thread is created, it has the same bind properties as its creator. This applies to the initial thread in the new process created by the **fork** subroutine: the new thread inherits the bind properties of the thread which called **fork**. When the **exec** subroutine is called, thread properties are left unchanged.

The **bindprocessor** subroutine will fail if the target process has a **Resource Attachment**.

Programs that use processor bindings should become Dynamic Logical Partitioning (DLPAR) aware. Refer to [Dynamic Logical Partitioning](AIX 5L Version 5.3 General Programming Concepts: Writing and Debugging Programs) for more information.

Parameters

<table>
<thead>
<tr>
<th>What</th>
<th>Specifies whether a process or a thread is being bound to a processor. The <strong>What</strong> parameter can take one of the following values:</th>
</tr>
</thead>
<tbody>
<tr>
<td>BINDPROCESS</td>
<td>A process is being bound to a processor.</td>
</tr>
<tr>
<td>BINDTHREAD</td>
<td>A thread is being bound to a processor.</td>
</tr>
</tbody>
</table>

| Who | Indicates a process or thread identifier, as appropriate for the **What** parameter, specifying the process or thread which is to be bound to a processor. |

| Where | If the **Where** parameter is a bind CPU identifier, it specifies the processor to which the process or thread is to be bound. A value of **PROCESSOR_CLASS_ANY** unbinds the specified process or thread, which will then be able to run on any processor. |

The **sysconf** subroutine can be used to retrieve information about the number of online processors in the system.

Return Values
On successful completion, the **bindprocessor** subroutine returns 0. Otherwise, a value of -1 is returned, and the **errno** global variable is set to indicate the error.

Error Codes
The **bindprocessor** subroutine is unsuccessful if one of the following is true:

<table>
<thead>
<tr>
<th>Error Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EINVAL</td>
<td>The <strong>What</strong> parameter is invalid, or the <strong>Where</strong> parameter indicates an invalid processor number or a processor class which is not currently available.</td>
</tr>
<tr>
<td>ESRCH</td>
<td>The specified process or thread does not exist.</td>
</tr>
<tr>
<td>EPERM</td>
<td>The caller does not have root user authority, and the <strong>Who</strong> parameter specifies either a process, or a thread belonging to a process, having a real or effective user ID different from that of the calling process. The target process has a <strong>Resource Attachment</strong>.</td>
</tr>
</tbody>
</table>

Related Information
The **bindprocessor** command.

The **exec** ("exec, execl, execlp, execv, execve, execvp, or exec Subroutine" on page 235) subroutine, **fork** ("fork, f_fork, or vfork Subroutine" on page 287) subroutine, **sysconf** subroutine, **thread_self** subroutine.
brk or sbrk Subroutine

Purpose
Changes data segment space allocation.

Library
Standard C Library (libc.a)

Syntax
```c
#include <unistd.h>

int brk ( char *EndDataSegment);

void *sbrk ( intptr_t Increment);
```

Description
The `brk` and `sbrk` subroutines dynamically change the amount of space allocated for the data segment of the calling process. (For information about segments, see the exec subroutine. For information about the maximum amount of space that can be allocated, see the ulimit and getrlimit subroutines.)

The change is made by resetting the break value of the process, which determines the maximum space that can be allocated. The break value is the address of the first location beyond the current end of the data region. The amount of available space increases as the break value increases. The available space is initialized to a value of 0 at the time it is used. The break value can be automatically rounded up to a size appropriate for the memory management architecture.

The `brk` subroutine sets the break value to the value of the `EndDataSegment` parameter and changes the amount of available space accordingly.

The `sbrk` subroutine adds to the break value the number of bytes contained in the `Increment` parameter and changes the amount of available space accordingly. The `Increment` parameter can be a negative number, in which case the amount of available space is decreased.

Parameters
- `EndDataSegment`: Specifies the effective address of the maximum available data.
- `Increment`: Specifies any integer.

Return Values
Upon successful completion, the `brk` subroutine returns a value of 0, and the `sbrk` subroutine returns the old break value. If either subroutine is unsuccessful, a value of -1 is returned and the `errno` global variable is set to indicate the error.

Error Codes
The `brk` subroutine and the `sbrk` subroutine are unsuccessful and the allocated space remains unchanged if one or more of the following are true:
The requested change allocates more space than is allowed by a system-imposed maximum. (For information on the system-imposed maximum on memory space, see the `ulimit` system call.)

The requested change sets the break value to a value greater than or equal to the start address of any attached shared-memory segment. (For information on shared memory operations, see the `shmat` subroutine.)

**Related Information**

The `exec` ("exec: execl, execle, execvp, or exec Subroutine" on page 235) subroutines, `getrlimit` ("getrlimit, getrlimit64, setrlimit, setrlimit64, or vlimit Subroutine" on page 419) subroutine, `shmat` subroutine, `shmdt` subroutine, `ulimit` subroutine.

The `_end` ("_end, _etext, or _edata Identifier" on page 223), `_etext` ("_end, _etext, or _edata Identifier" on page 223), or `_edata` ("_end, _etext, or _edata Identifier" on page 223) identifier.

Subroutine Overview in AIX 5L Version 5.3 General Programming Concepts: Writing and Debugging Programs.

---

### bsearch Subroutine

**Purpose**

Performs a binary search.

**Library**

Standard C Library (`libc.a`)

**Syntax**

```c
#include <stdlib.h>

void *bsearch (const void *Key, const void *Base, size_t NumberOfElements, size_t Size, int (*ComparisonPointer)(const void *, const void *));
```

**Description**

The `bsearch` subroutine is a binary search routine.

The `bsearch` subroutine searches an array of `NumberOfElements` objects, the initial member of which is pointed to by the `Base` parameter, for a member that matches the object pointed to by the `Key` parameter. The size of each member in the array is specified by the `Size` parameter.

The array must already be sorted in increasing order according to the provided comparison function `ComparisonPointer` parameter.

**Parameters**

- `Key` Points to the object to be sought in the array.
- `Base` Points to the element at the base of the table.
- `NumberOfElements` Specifies the number of elements in the array.
ComparisonPointer
Points to the comparison function, which is called with two arguments that point to
the Key parameter object and to an array member, in that order.

Size
Specifies the size of each member in the array.

Return Values
If the Key parameter value is found in the table, the bsearch subroutine returns a pointer to the element
found.

If the Key parameter value is not found in the table, the bsearch subroutine returns the null value. If two
members compare as equal, the matching member is unspecified.

For the ComparisonPointer parameter, the comparison function compares its parameters and returns a
value as follows:
• If the first parameter is less than the second parameter, the ComparisonPointer parameter returns a
  value less than 0.
• If the first parameter is equal to the second parameter, the ComparisonPointer parameter returns a
  value of 0.
• If the first parameter is greater than the second parameter, the ComparisonPointer parameter returns a
  value greater than 0.

The comparison function need not compare every byte, so arbitrary data can be contained in the elements
in addition to the values being compared.

The Key and Base parameters should be of type pointer-to-element and cast to type pointer-to-character.
Although declared as type pointer-to-character, the value returned should be cast into type
pointer-to-element.

Related Information
The hsearch subroutine, lsearch subroutine, bsearch subroutine, lsort subroutine.

Knuth, Donald E.; The Art of Computer Programming, Volume 3. Reading, Massachusetts,
Addison-Wesley, 1981.

Searching and Sorting Example Program and Subroutines Overview in AIX 5L Version 5.3 General
Programming Concepts: Writing and Debugging Programs.

btowc Subroutine

Purpose
Single-byte to wide-character conversion.

Library
Standard Library (libc.a)

Syntax
#include <stdio.h>
#include <wchar.h>

wint_t btowc (intc);
Description
The `btowc` function determines whether `c` constitutes a valid (one-byte) character in the initial shift state.

The behavior of this function is affected by the LC_CTYPE category of the current locale.

Return Values
The `btowc` function returns WEOF if `c` has the value EOF or if (unsigned char) `c` does not constitute a valid (one-byte) character in the initial shift state. Otherwise, it returns the wide-character representation of that character.

Related Information
The `wctob` subroutine.

buildproclist Subroutine

Purpose
Retrieves a list of process transaction records based on the criteria specified.

Library
The `libaacct.a` library.

Syntax
```
#define <sys/aacct.h>
int buildproclist(crit, crit_list, n_crit, p_list, sublist)
int crit;
union proc_crit *crit_list;
int n_crit;
struct aacct_tran_rec *p_list;
struct aacct_tran_rec **sublist;
```

Description
The `buildproclist` subroutine retrieves a subset of process transaction records from the master process transaction records that are given as input based on the selection criteria provided. This selection criteria can be one of the following values defined in `sys/aacct.h`:

- CRIT_UID
- CRIT_GID
- CRIT_PROJ
- CRIT_CMD

For example, if the criteria is specified as CRIT_UID, the list of process transaction records for specific user IDs will be retrieved. The list of user IDs are passed through the crit_list argument of type union proc_crit. Based on the specified criteria, the caller has to pass an array of user IDs, group IDs, project IDs or command names in this union.

Usually, the master list of transaction records is obtained by a prior call to the `getproclist` subroutine.

Parameters
- `crit` Integer value representing the selection criteria for the process records.
- `crit_list` Pointer to union proc_crit where the data for the selection criteria is passed.
- `n_crit` Number of elements to be considered for the selection, such as the number of user IDs.
- `p_list` Master list of process transaction records.
sublist

Pointer to the linked list of acct_tran_rec structures, which hold the retrieved process transaction records.

Security
No restrictions. Any user can call this function.

Return Values

<table>
<thead>
<tr>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>The call to the subroutine was successful.</td>
</tr>
<tr>
<td>-1</td>
<td>The call to the subroutine failed.</td>
</tr>
</tbody>
</table>

Error Codes

<table>
<thead>
<tr>
<th>Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EINVAL</td>
<td>The passed pointer is NULL.</td>
</tr>
<tr>
<td>ENOMEM</td>
<td>Insufficient memory.</td>
</tr>
<tr>
<td>EPERM</td>
<td>Permission denied. Unable to read the data file.</td>
</tr>
</tbody>
</table>

Related Information

The "buildproclist Subroutine" on page 125, "buildtranlist or freetranlist Subroutine," "getfilehdr Subroutine" on page 362.

The acctrp command.

Understanding the Advanced Accounting Subsystem

buildtranlist or freetranlist Subroutine

Purpose
Read the advanced accounting records from the advanced accounting data file.

Library
The libaacct.a library.

Syntax

```c
#define <sys/aacct.h>
builttranlist(char *filename, int trid[], int ntrids, long begin_time, long end_time, struct acct_tran_rec **tran_list);
char *filename;
unsigned int trid[];
unsigned int ntrids;
long long begin_time;
long long end_time;
struct acct_tran_rec **tran_list;
free tranlist(struct tran_list);
```

Description
The builttranlist subroutine retrieves the transaction records of the specified transaction type from the accounting data file. The required transaction IDs are passed as arguments, and these IDs are defined in sys/aacct.h. The list of transaction records are returned to the calling program through the tran_list pointer argument.
This API can be called multiple times with different accounting data file names to generate a consolidated list of transaction records from multiple data files. It appends the new file data to the end of the linked list pointed to by the *tran_list* argument. In addition, it internally sorts the transaction records based on the time of transaction so users can get a time-sorted list of transaction records from this routine. This subroutine can also be used to retrieve the intended transaction records for a particular interval of time by specifying the begin and end times of this interval as arguments.

The *freetranlist* subroutine frees the memory allocated to these transaction records. It can be used to deallocate memory that has been allocated to the transaction record lists created by routines such as *buildtranlist*, *getproclist*, *getlparlist*, and *getarmlist*.

**Parameters**

- **begin_time**: Specifies the start timestamp for collecting records in a particular intervals. The input is in seconds since EPOCH. Specifying -1 retrieves all the records.
- **end_time**: Specifies the end timestamp for collecting records in a particular intervals. The input is in seconds since EPOCH. Specifying -1 retrieves all the records.
- **filename**: Name of the advanced accounting data file.
- **ntrids**: Count of transaction IDs passed in the array *trid*.
- **tran_list**: Pointer to the linked list of *aacct_tran_rec* structures that are to be returned to the caller or freed.
- **trid**: An array of transaction record type identifiers.

**Security**

No restrictions. Any user can call this function.

**Return Values**

- **0**: The call to the subroutine was successful.
- **-1**: The call to the subroutine failed.

**Error Codes**

- **EINVAL**: The passed pointer is NULL.
- **ENOENT**: Specified data file does not exist.
- **ENOMEM**: Insufficient memory.
- **EPERM**: Permission denied. Unable to read the data file.

**Related Information**

The [buildproclist Subroutine](#) on page 125, [getproclist, getlparlist, or getarmlist Subroutine](#) on page 409.

**_check_lock Subroutine**

**Purpose**

Conditionally updates a single word variable atomically.

**Library**

Standard C library (*libc.a*)
Syntax
#include <sys/atomic_op.h>

boolean_t _check_lock (word_addr, old_val, new_val);
atomic_p word_addr;
int old_val;
int new_val;

Parameters

word_addr Specifies the address of the single word variable.
old_val Specifies the old value to be checked against the value of the single word variable.
new_val Specifies the new value to be conditionally assigned to the single word variable.

Description
The _check_lock subroutine performs an atomic (uninterruptible) sequence of operations. The compare_and_swap subroutine is similar, but does not issue synchronization instructions and therefore is inappropriate for updating lock words.

Note: The word variable must be aligned on a full word boundary.

Return Values

FALSE Indicates that the single word variable was equal to the old value and has been set to the new value.
TRUE Indicates that the single word variable was not equal to the old value and has been left unchanged.

Related Information
The _clear_lock subroutine.

_clear_lock Subroutine

Purpose
Stores a value in a single word variable atomically.

Library
Standard C library (libc.a)

Syntax
#include <sys/atomic_op.h>

void _clear_lock (word_addr, value);
atomic_p word_addr;
int value

Parameters

word_addr Specifies the address of the single word variable.
value Specifies the value to store in the single word variable.
Description
The _clear_lock subroutine performs an atomic (uninterruptible) sequence of operations.

This subroutine has no return values.

Note: The word variable must be aligned on a full word boundary.

Related Information
The _check_lock subroutine.

---

cabs, cabsf, or cabsl Subroutine

Purpose
Returns a complex absolute value.

Syntax
```c
#include <complex.h>

double cabs (double complex z);
float cabsf (float complex z);
long double cabsl (long double complex z);
```

Description
The cabs, cabsf, or cabsl subroutines compute the complex absolute value (also called norm, modulus, or magnitude) of the z parameter.

Parameters
- **z** Specifies the value to be computed.

Return Values
Returns the complex absolute value.

---

cacos, cacosf, or cacosl Subroutine

Purpose
Computes the complex arc cosine.

Syntax
```c
#include <complex.h>

double complex cacos (double complex z);
double complex cacosf (float complex z);
```

Base Operating System (BOS) Runtime Services (A-P) 129
Description
The cacos, cacosf, or cacosl subroutine computes the complex arc cosine of \( z \), with branch cuts outside the interval \([-1, +1]\) along the real axis.

Parameters
\( z \) Specifies the value to be computed.

Return Values
The cacos, cacosf, or cacosl subroutine returns the complex arc cosine value, in the range of a strip mathematically unbounded along the imaginary axis and in the interval \([0, \pi]\) along the real axis.

cacosh, cacoshf, or cacoshl Subroutines

Purpose
Computes the complex arc hyperbolic cosine.

Syntax
```c
#include <complex.h>

double complex cacosh (double complex z);
double complex cacosh (float complex z);
float complex cacoshf (float complex z);
long double complex cacoshl (long double complex z);
```

Description
The cacosh, cacoshf, or cacoshl subroutine computes the complex arc hyperbolic cosine of the \( z \) parameter, with a branch cut at values less than 1 along the real axis.

Parameters
\( z \) Specifies the value to be computed.

Return Values
The cacosh, cacoshf, or cacoshl subroutine returns the complex arc hyperbolic cosine value, in the range of a half-strip of non-negative values along the real axis and in the interval \([-i \pi , +i \pi]\) along the imaginary axis.

Related Information
The "ccosh, ccoshf, or ccoshl Subroutine" on page 139.
carg, cargf, or cargl Subroutine

Purpose
Returns the complex argument value.

Syntax
#include <complex.h>

double carg (double complex z);
float cargf (float complex z);
long double cargl (long double complex z);

Description
The carg, cargf, or cargl subroutine computes the argument (also called phase angle) of the z parameter, with a branch cut along the negative real axis.

Parameters
z
Specifies the value to be computed.

Return Values
The carg, cargf, or cargl subroutine returns the value of the argument in the interval [-pi, +pi].

Related Information
The "cimag, cimagf, or cimagl Subroutine" on page 163, "conj, conjf, or conjl Subroutine" on page 182, and "cproj, cprojf, or cprojl Subroutine" on page 189.

casin, casinf, or casinl Subroutine

Purpose
Computes the complex arc sine.

Syntax
#include <complex.h>

double complex casin (double complex z);
float complex casinf (float complex z);
long double complex casinl (long double complex z);

Description
The casin, casinf, or casinl subroutine computes the complex arc sine of the z parameter, with branch cuts outside the interval [-1, +1] along the real axis.
Parameters

z Specifies the value to be computed.

Return Values
The casin, casinf, or casinl subroutine returns the complex arc sine value, in the range of a strip mathematically unbounded along the imaginary axis and in the interval [-pi/2, +pi/2] along the real axis.

Related Information
The “csin, csinf, or csinl Subroutine” on page 193.

casinh, casinfh, or casinlh Subroutine

Purpose
Computes the complex arc hyperbolic sine.

Syntax
#include <complex.h>

double complex casinh (z)
double complex z;

float complex casinhf (z)
float complex z;

long double complex casinhl (z)
long double complex z;

Description
The casinh, casinhf, and casinlh subroutines compute the complex arc hyperbolic sine of the z parameter, with branch cuts outside the interval [-i, +i] along the imaginary axis.

Parameters

z Specifies the value to be computed.

Return Values
The casinh, casinhf, and casinlh subroutines return the complex arc hyperbolic sine value, in the range of a strip mathematically unbounded along the real axis and in the interval [-i pi/2, +i pi/2] along the imaginary axis.

Related Information
The “casin, casinf, or casinl Subroutine” on page 131.

catan, catanf, or catanl Subroutine

Purpose
Computes the complex arc tangent.
Syntax
#include <complex.h>

double complex catan (z)
double complex z;

float complex catanf (z)
float complex z;

long double complex catanl (z)
long double complex z;

Description
The catan, catanf, and catanl subroutines compute the complex arc tangent of z, with branch cuts outside the interval \([-i, +i]\) along the imaginary axis.

Parameters
z Specifies the value to be computed.

Return Values
The catan, catanf, and catanl subroutines return the complex arc tangent value, in the range of a strip mathematically unbounded along the imaginary axis and in the interval \([-\pi/2, +\pi/2]\) along the real axis.

Related Information
"catanh, catanhf, or catanhl Subroutine"

catanh, catanhf, or catanhl Subroutine

Purpose
Computes the complex arc hyperbolic tangent.

Syntax
#include <complex.h>

double complex catanh (z)
double complex z;

float complex catanhf (z)
float complex z;

long double complex catanhl (z)
long double complex z;

Description
The catanh, catanhf, and catanhl subroutines compute the complex arc hyperbolic tangent of z, with branch cuts outside the interval \([-1, +1]\) along the real axis.

Parameters
z Specifies the value to be computed.
Return Values
The catanh, catanhf, and catanhl subroutines return the complex arc hyperbolic tangent value, in the range of a strip mathematically unbounded along the real axis and in the interval \([-i \pi/2, +i \pi/2]\) along the imaginary axis.

Related Information
“catan, catanf, or catanl Subroutine" on page 132

catclose Subroutine

Purpose
Closes a specified message catalog.

Library
Standard C Library (libc.a)

Syntax
```
#include <nl_types.h>

int catclose (CatalogDescriptor)
    nl_catd CatalogDescriptor;
```

Description
The catclose subroutine closes a specified message catalog. If your program accesses several message catalogs and you reach the maximum number of opened catalogs (specified by the NL_MAXOPEN constant), you must close some catalogs before opening additional ones. If you use a file descriptor to implement the nl_catd data type, the catclose subroutine closes that file descriptor.

The catclose subroutine closes a message catalog only when the number of calls it receives matches the total number of calls to the catopen subroutine in an application. All message buffer pointers obtained by prior calls to the catgets subroutine are not valid when the message catalog is closed.

Parameters

CatalogDescriptor Points to the message catalog returned from a call to the catopen subroutine.

Return Values
The catclose subroutine returns a value of 0 if it closes the catalog successfully, or if the number of calls it receives is fewer than the number of calls to the catopen subroutine.

The catclose subroutine returns a value of -1 if it does not succeed in closing the catalog. The catclose subroutine is unsuccessful if the number of calls it receives is greater than the number of calls to the catopen subroutine, or if the value of the CatalogDescriptor parameter is not valid.

Related Information
The catgets subroutine, catopen subroutine.

For more information about the Message Facility, see Message Facility Overview for Programming in AIX 5L Version 5.3 General Programming Concepts: Writing and Debugging Programs.
catgets Subroutine

Purpose
Retrieves a message from a catalog.

Library
Standard C Library (libc.a)

Syntax
#include <nl_types>

char *catgets (CatalogDescriptor, SetNumber, MessageNumber, String)

Description
The catgets subroutine retrieves a message from a catalog after a successful call to the catopen subroutine. If the catgets subroutine finds the specified message, it loads it into an internal character string buffer, ends the message string with a null character, and returns a pointer to the buffer.

The catgets subroutine uses the returned pointer to reference the buffer and display the message. However, the buffer can not be referenced after the catalog is closed.

Parameters

<table>
<thead>
<tr>
<th>CatalogDescriptor</th>
<th>Specifies a catalog description that is returned by the catopen subroutine.</th>
</tr>
</thead>
<tbody>
<tr>
<td>SetNumber</td>
<td>Specifies the set ID.</td>
</tr>
<tr>
<td>MessageNumber</td>
<td>Specifies the message ID. The SetNumber and MessageNumber parameters specify a particular message to retrieve in the catalog.</td>
</tr>
<tr>
<td>String</td>
<td>Specifies the default character-string buffer.</td>
</tr>
</tbody>
</table>

Return Values
If the catgets subroutine is unsuccessful for any reason, it returns the user-supplied default message string specified by the String parameter.

Related Information
The catclose subroutine, catopen subroutine. 

For more information about the Message Facility, see Message Facility Overview for Programming in AIX 5L Version 5.3 General Programming Concepts: Writing and Debugging Programs.

For more information about subroutines and libraries, see Subroutines Overview in AIX 5L Version 5.3 General Programming Concepts: Writing and Debugging Programs.
**catopen Subroutine**

**Purpose**
Opens a specified message catalog.

**Library**
Standard C Library (libc.a)

**Syntax**
```
#include <nl_types.h>

nl_catd catopen (const char *CatalogName; int Parameter;
```

**Description**
The `catopen` subroutine opens a specified message catalog and returns a catalog descriptor used to retrieve messages from the catalog. The contents of the catalog descriptor are complete when the `catgets` subroutine accesses the message catalog. The `nl_catd` data type is used for catalog descriptors and is defined in the `nl_types.h` file.

If the catalog file name referred to by the `CatalogName` parameter contains a leading / (slash), it is assumed to be an absolute path name. If the catalog file name is not an absolute path name, the user environment determines which directory paths to search. The `NLSPATH` environment variable defines the directory search path. When this variable is used, the `setlocale` subroutine must be called before the `catopen` subroutine.

A message catalog descriptor remains valid in a process until that process or a successful call to one of the `exec` functions closes it.

You can use two special variables, `%N` and `%L`, in the `NLSPATH` environment variable. The `%N` variable is replaced by the catalog name referred to by the call that opens the message catalog. The `%L` variable is replaced by the value of the `LC_MESSAGES` category.

The value of the `LC_MESSAGES` category can be set by specifying values for the `LANG`, `LC_ALL`, or `LC_MESSAGES` environment variable. The value of the `LC_MESSAGES` category indicates which locale-specific directory to search for message catalogs. For example, if the `catopen` subroutine specifies a catalog with the name `mycmd`, and the environment variables are set as follows:

```
NLSPATH=../%N:/%N:/system/nls/%L/%N:/system/nls/%N LANG=fr_FR
```

then the application searches for the catalog in the following order:

```
../mycmd
./mycmd
/system/nls/fr_FR/mycmd
/system/nls/mycmd
```

If you omit the `%N` variable in a directory specification within the `NLSPATH` environment variable, the application assumes that it defines a catalog name and opens it as such and will not traverse the rest of the search path.

If the `NLSPATH` environment variable is not defined, the `catopen` subroutine uses the default path. See the `/etc/environment` file for the `NLSPATH` default path. If the `LC_MESSAGES` category is set to the
default value C, and the LC_FASTMSG environment variable is set to true, then subsequent calls to the catgets subroutine generate pointers to the program-supplied default text.

The catopen subroutine treats the first file it finds as a message file. If you specify a non-message file in a NLSPATH, for example, /usr/bin/ls, catopen treats /usr/bin/ls as a message catalog. Thus no messages are found and default messages are returned. If you specify /tmp in a NLSPATH, /tmp is opened and searched for messages and default messages are displayed.

Parameters

CatalogName

Specifies the catalog file to open.

Parameter

Determines the environment variable to use in locating the message catalog. If the value of the Parameter parameter is 0, use the LANG environment variable without regard to the LC_MESSAGES category to locate the catalog. If the value of the Parameter parameter is the NL_CAT_LOCALE macro, use the LC_MESSAGES category to locate the catalog.

Return Values

The catopen subroutine returns a catalog descriptor. If the LC_MESSAGES category is set to the default value C, and the LC_FASTMSG environment variable is set to true, the catopen subroutine returns a value of -1.

If the LCMESSAGES category is not set to the default value C but the catopen subroutine returns a value of -1, an error has occurred during creation of the structure of the nl_catd data type or the catalog name referred to by the CatalogName parameter does not exist.

Related Information

The catclose ("catclose Subroutine" on page 134) subroutine, catgets ("catgets Subroutine" on page 135) subroutine, exec ("exec: execl, execle, execcl, execvp, or exec Subroutine" on page 235) subroutines, setlocale subroutine.

The environment file.

For more information about the Message Facility, see the Message Facility Overview for Programming in AIX 5L Version 5.3 General Programming Concepts: Writing and Debugging Programs.

For more information about subroutines and libraries, see the Subroutines Overview in AIX 5L Version 5.3 General Programming Concepts: Writing and Debugging Programs.

cbrtf, cbrtl, or cbart Subroutine

Purpose

Computes the cube root.

Syntax

```c
#include <math.h>

float cbrtf (x) 
float x;

long double cbrtl (x)
```
long double x;
double cbrt (x)
double x;

Description
The cbrtf, cbrtl, and cbrt subroutines compute the real cube root of the x argument.

Parameters
x
  Specifies the value to be computed.

Return Values
Upon successful completion, the cbrtf, cbrtl, and cbrt subroutines return the cube root of x.
If x is NaN, an NaN is returned.
If x is ±0 or ±Inf, x is returned.

Related Information
math.h in AIX 5L Version 5.3 Files Reference.

ccos, ccosf, or ccosl Subroutine

Purpose
Computes the complex cosine.

Syntax
#include <complex.h>

double complex ccos (z)
double complex z;

float complex ccosf (z)
float complex z;

long double complex ccosl (z)
long double complex z;

Description
The ccos, ccosf, and ccosl subroutines compute the complex cosine of z.

Parameters
z
  Specifies the value to be computed.

Return Values
The ccos, ccosf, and ccosl subroutines return the complex cosine value.

Related Information
“cacos, cacosf, or cacosl Subroutine” on page 129
ccosh, ccoshf, or ccoshl Subroutine

Purpose
Computes the complex hyperbolic cosine.

Syntax
#include <complex.h>

double complex ccosh (z);
double complex z;

float complex ccoshf (z);
float complex z;

long double complex ccoshl (z);
long double complex z;

Description
The ccosh, ccoshf, and ccoshl subroutines compute the complex hyperbolic cosine of z.

Parameters
z Specifies the value to be computed.

Return Values
The ccosh, ccoshf, and ccoshl subroutines return the complex hyperbolic cosine value.

Related Information
"cacosh, cacoshf, or cacoshl Subroutines" on page 130

ccsidtocs or cstoccssid Subroutine

Purpose
Provides conversion between coded character set IDs (CCSID) and code set names.

Library
The iconv Library (libiconv.a)

Syntax
#include <iconv.h>

CCSID cctoccssid (* Codeset)
const char *Codeset;

char *ccsidtocs (CCSID CCSID);

Description
The cctoccssid subroutine returns the CCSID of the code set specified by the Codeset parameter. The cccsidtocs subroutine returns the code set name of the CCSID specified by CCSID parameter. CCSIDs are registered IBM coded character set IDs.
Parameters

*Codeset*
Specifies the code set name to be converted to its corresponding CCSID.

*CCSID*
Specifies the CCSID to be converted to its corresponding code set name.

Return Values

If the code set is recognized by the system, the `cstoccsid` subroutine returns the corresponding CCSID. Otherwise, null is returned.

If the CCSID is recognized by the system, the `ccsidtocs` subroutine returns the corresponding code set name. Otherwise, a null pointer is returned.

Related Information

For more information about code set conversion, see [Converters Overview for Programming in AIX 5L Version 5.3 General Programming Concepts: Writing and Debugging Programs](#).

The [National Language Support Overview for Programming in AIX 5L Version 5.3 General Programming Concepts: Writing and Debugging Programs](#).

[Subroutines Overview](#) in AIX 5L Version 5.3 General Programming Concepts: Writing and Debugging Programs.

ceil, ceilf, or ceill Subroutine

**Purpose**

Computes the ceiling value.

**Syntax**

```c
#include <math.h>

float ceilf (x)
float x;

long double ceill (x)
long double x;

double ceil (x)
double x;
```

**Description**

The `ceilf`, `ceill`, and `ceil` subroutines compute the smallest integral value not less than `x`.

An application wishing to check for error situations should set the `errno` global variable to zero and call `feclearexcept(FE_ALL_EXCEPT)` before calling these functions. Upon return, if `errno` is nonzero or `fetestexcept(FE_INVALID | FE_DIVBYZERO | FE_OVERFLOW | FE_UNDERFLOW)` is nonzero, an error has occurred.

**Parameters**

`x`
Specifies the smallest integral value to be computed.
Return Values
Upon successful completion, the `ceil`, `ceilf`, and `ceil` subroutines return the smallest integral value not less than \( x \), expressed as a type `float`, `long double`, or `double`, respectively.

If \( x \) is NaN, a NaN is returned.

If \( x \) is \( \pm 0 \) or \( \pm \infty \), \( x \) is returned.

If the correct value would cause overflow, a range error occurs and the `ceil`, `ceilf`, and `ceil` subroutines return the value of the macro `HUGE_VALF`, `HUGE_VALL`, and `HUGE_VAL`, respectively.

Related Information
- "feclearexcept Subroutine" on page 262
- "fetestexcept Subroutine" on page 270
- "floor, floorf, floorl, nearest, trunc, itrunc, or ultrunc Subroutine" on page 274
- "class, _class, finite, isnan, or unordered Subroutines" on page 167

`math.h` in AIX 5L Version 5.3 Files Reference.

cexp, cexpf, or cexpl Subroutine

Purpose
Performs complex exponential computations.

Syntax
```c
#include <complex.h>

double complex cexp (z)
double complex z;

float complex cexpf (z)
float complex z;

long double complex cexpl (z)
long double complex z;
```

Description
The `cexp`, `cexpf`, and `cexpl` subroutines compute the complex exponent of \( z \), defined as \( e^z \).

Parameters
\( z \) Specifies the value to be computed.

Return Values
The `cexp`, `cexpf`, and `cexpl` subroutines return the complex exponential value of \( z \).

Related Information
The "clog, clogf, or clogl Subroutine" on page 174.
cfgetospeed, cfsetospeed, cfgetispeed, or cfsetispeed Subroutine

Purpose
Gets and sets input and output baud rates.

Library
Standard C Library (libc.a)

Syntax
#include <termios.h>

speed_t cfgetospeed (TermiosPointer)
const struct termios *TermiosPointer;

int cfsetospeed (TermiosPointer, Speed)
struct termios *TermiosPointer;
speed_t Speed;

speed_t cfgetispeed (TermiosPointer)
const struct termios *TermiosPointer;

int cfsetispeed (TermiosPointer, Speed)
struct termios *TermiosPointer;
speed_t Speed;

Description
The baud rate subroutines are provided for getting and setting the values of the input and output baud rates in the termios structure. The effects on the terminal device described below do not become effective and not all errors are detected until the tcsetattr function is successfully called.

The input and output baud rates are stored in the termios structure. The supported values for the baud rates are shown in the table that follows this discussion.

The termios.h file defines the type speed_t as an unsigned integral type.

The cfgetospeed subroutine returns the output baud rate stored in the termios structure pointed to by the TermiosPointer parameter.

The cfsetospeed subroutine sets the output baud rate stored in the termios structure pointed to by the TermiosPointer parameter to the value specified by the Speed parameter.

The cfgetispeed subroutine returns the input baud rate stored in the termios structure pointed to by the TermiosPointer parameter.

The cfsetispeed subroutine sets the input baud rate stored in the termios structure pointed to by the TermiosPointer parameter to the value specified by the Speed parameter.

Certain values for speeds have special meanings when set in the termios structure and passed to the tcsetattr function. These values are discussed in the tcsetattr subroutine.

The following table lists possible baud rates:

<table>
<thead>
<tr>
<th>Baud Rate Values</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>B0</td>
<td>Hang up</td>
</tr>
</tbody>
</table>
### Baud Rate Values

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>B5</td>
<td>50 baud</td>
</tr>
<tr>
<td>B75</td>
<td>75 baud</td>
</tr>
<tr>
<td>B110</td>
<td>110 baud</td>
</tr>
<tr>
<td>B134</td>
<td>134 baud</td>
</tr>
<tr>
<td>B150</td>
<td>150 baud</td>
</tr>
<tr>
<td>B200</td>
<td>200 baud</td>
</tr>
<tr>
<td>B300</td>
<td>300 baud</td>
</tr>
<tr>
<td>B600</td>
<td>600 baud</td>
</tr>
<tr>
<td>B1200</td>
<td>1200 baud</td>
</tr>
<tr>
<td>B1800</td>
<td>1800 baud</td>
</tr>
<tr>
<td>B2400</td>
<td>2400 baud</td>
</tr>
<tr>
<td>B4800</td>
<td>4800 baud</td>
</tr>
<tr>
<td>B9600</td>
<td>9600 baud</td>
</tr>
<tr>
<td>B19200</td>
<td>19200 baud</td>
</tr>
<tr>
<td>B38400</td>
<td>38400 baud</td>
</tr>
</tbody>
</table>

The **termios.h** file defines the name symbols of the table.

### Parameters

- **TermiosPointer** Points to a **termios** structure.
- **Speed** Specifies the baud rate.

### Return Values

The **cfgetospeed** and **cfgetispeed** subroutines return exactly the value found in the **termios** data structure, without interpretation.

Both the **cfsetospeed** and **cfsetispeed** subroutines return a value of 0 if successful and -1 if unsuccessful.

### Examples

To set the output baud rate to 0 (which forces modem control lines to stop being asserted), enter:

```c
cfsetospeed(&my_termios, B0);
tcsetattr(stdout, TCSADRAIN, &my_termios);
```

### Related Information

- The **tcsetattr** subroutine.
- The **termios.h** file.

[Input and Output Handling Programmer's Overview](#) in *AIX 5L Version 5.3 General Programming Concepts: Writing and Debugging Programs.*
**chacl or fchacl Subroutine**

**Purpose**
Changes the AIXC ACL type access control information of a file.

**Library**
Standard C Library (*libc.a*)

**Syntax**

```c
#include <sys/acl.h>
#include <sys/mode.h>

int chacl (Path, ACL, ACLSize)
char *Path;
struct acl *ACL;
int ACLSize;

int fchacl (FileDescriptor, ACL, ACLSize)
int FileDescriptor;
struct acl *ACL;
int ACLSize;
```

**Description**
The `chacl` and `fchacl` subroutines set the access control attributes of a file according to the AIXC ACL Access Control List (ACL) structure pointed to by the `ACL` parameter. Note that these routines could fail if the current ACL associated with the file system object is of a different type or if the underlying physical file system does not support AIXC ACL type. It is strongly recommended that applications stop using these interfaces and instead make use of `aclx_get/aclx_fget` and `aclx_put/aclx_fput` subroutines to change the ACL.

**Parameters**

- **Path**
  Specifies the path name of the file.
ACL Specifies the AIXC ACL to be established on the file. The format of an AIXC ACL is defined in the `sys/acl.h` file and contains the following members:

`acl_len` Specifies the size of the ACL (Access Control List) in bytes, including the base entries.

Note: The entire ACL for a file cannot exceed one memory page (4096 bytes).

`acl_mode` Specifies the file mode.

The following bits in the `acl_mode` member are defined in the `sys/mode.h` file and are significant for this subroutine:

- **S_ISUID** Enables the `setuid` attribute on an executable file.
- **S_ISGID** Enables the `setgid` attribute on an executable file. Enables the group-inheritance attribute on a directory.
- **S_ISVTX** Enables linking restrictions on a directory.
- **S_ISACL** Enables extended ACL entry processing. If this attribute is not set, only the base entries (owner, group, and default) are used for access authorization checks.

Other bits in the mode, including the following, are ignored:

- **u_access** Specifies access permissions for the file owner.
- **g_access** Specifies access permissions for the file group.
- **o_access** Specifies access permissions for the default class of others.

`acl_ext[]` Specifies an array of the extended entries for this access control list.

The members for the base ACL (owner, group, and others) can contain the following bits, which are defined in the `sys/access.h` file:

- **R_ACC** Allows read permission.
- **W_ACC** Allows write permission.
- **X_ACC** Allows execute or search permission.

*FileDescriptor* Specifies the file descriptor of an open file.

*ACLSize* Specifies the size of the buffer containing the ACL.

Note: The `chacl` subroutine requires the `Path`, `ACL`, and `ACLSize` parameters. The `fchacl` subroutine requires the `FileDescriptor`, `ACL`, and `ACLSize` parameters.

**ACL Data Structure for chacl**

Each access control list structure consists of one `struct acl` structure containing one or more `struct acl_entry` structures with one or more `struct ace_id` structures.

If the `struct ace_id` structure has `id_type` set to `ACEID_USER` or `ACEID_GROUP`, there is only one `id_data` element. To add multiple IDs to an ACL you must specify multiple `struct ace_id` structures when `id_type` is set to `ACEID_USER` or `ACEID_GROUP`. In this case, no error is returned for the multiple
elements, and the access checking examines only the first element. Specifically, the error value **EINVAL** is not returned for **acl_len** being incorrect in the ACL structure although more than one uid or gid is specified.

### Return Values
Upon successful completion, the **chacl** and **fchac1** subroutines return a value of 0. If the **chacl** or **fchac1** subroutine fails, a value of -1 is returned, and the **errno** global variable is set to indicate the error.

### Error Codes
The **chacl** subroutine fails and the access control information for a file remains unchanged if one or more of the following are true:

- **ENOTDIR** A component of the *Path* prefix is not a directory.
- **ENOENT** A component of the *Path* does not exist or has the disallow truncation attribute (see the **ulimit** subroutine).
- **ENOENT** The *Path* parameter was null.
- **EACCES** Search permission is denied on a component of the *Path* prefix.
- **EFAULT** The *Path* parameter points to a location outside of the allocated address space of the process.
- **ESTALE** The process’ root or current directory is located in a virtual file system that has been unmounted.
- **ELOOP** Too many symbolic links were encountered in translating the *Path* parameter.
- **ENOENT** A symbolic link was named, but the file to which it refers does not exist.
- **ENAMETOOLONG** A component of the *Path* parameter exceeded 255 characters, or the entire *Path* parameter exceeded 1023 characters.

The **chacl** or **fchac1** subroutine fails and the access control information for a file remains unchanged if one or more of the following are true:

- **EROFS** The file specified by the *Path* parameter resides on a read-only file system.
- **EFAULT** The *ACL* parameter points to a location outside of the allocated address space of the process.
- **EINVAL** The *ACL* parameter does not point to a valid ACL.
- **EINVAL** The *acl_len* member in the ACL is not valid.
- **EIO** An I/O error occurred during the operation.
- **ENOSPC** The size of the *ACL* parameter exceeds the system limit of one memory page (4KB).
- **EPERM** The effective user ID does not match the ID of the owner of the file, and the invoker does not have root user authority.

The **fchac1** subroutine fails and the file permissions remain unchanged if the following is true:

- **EBADF** The file descriptor *FileDescriptor* is not valid.

If Network File System (NFS) is installed on your system, the **chacl** and **fchac1** subroutines can also fail if the following is true:

- **ETIMEDOUT** The connection timed out.

### Security
Access Control: The invoker must have search permission for all components of the *Path* prefix.
## Auditing Events:

<table>
<thead>
<tr>
<th>Event</th>
<th>Information</th>
</tr>
</thead>
<tbody>
<tr>
<td>chacl</td>
<td>Path</td>
</tr>
<tr>
<td>fchacl</td>
<td>FileDescriptor</td>
</tr>
</tbody>
</table>

## Related Information

The `acl_chg` ("acl_chg or acl_fchg Subroutine" on page 8) subroutine, `acl_get` ("acl_get or acl_fget Subroutine" on page 10) subroutine, `acl_put` ("acl_put or acl_fput Subroutine" on page 12) subroutine, `acl_set" (acl_set or acl_fset Subroutine" on page 14) subroutine, `chmod" (chmod or fchmod Subroutine" on page 148) subroutine, `stat` subroutine, `statacl` subroutine.

"aclx_get or aclx_fget Subroutine" on page 17, "aclx_put or aclx_fput Subroutine" on page 25.

The `aclget` command, `aclput` command.

List of Security and Auditing Subroutines and Subroutines Overview in AIX 5L Version 5.3 General Programming Concepts: Writing and Debugging Programs.

## chdir Subroutine

### Purpose
Changes the current directory.

### Library
Standard C Library (`libc.a`)

### Syntax
```c
#include <unistd.h>

int chdir (const char *Path);
```

### Description
The `chdir` subroutine changes the current directory to the directory indicated by the `Path` parameter.

### Parameters
- **Path** A pointer to the path name of the directory. If the `Path` parameter refers to a symbolic link, the `chdir` subroutine sets the current directory to the directory pointed to by the symbolic link. If Network File System (NFS) is installed on the system, this path can cross into another node.

The current directory, also called the current working directory, is the starting point of searches for path names that do not begin with a / (slash). The calling process must have search access to the directory specified by the `Path` parameter.

### Return Values
Upon successful completion, the `chdir` subroutine returns a value of 0. Otherwise, a value of -1 is returned and the `errno` global variable is set to identify the error.
Error Codes
The `chdir` subroutine fails and the current directory remains unchanged if one or more of the following are true:

- **EACCES**  
  Search access is denied for the named directory.
- **ENOENT**  
  The named directory does not exist.
- **ENOTDIR**  
  The path name is not a directory.

The `chdir` subroutine can also be unsuccessful for other reasons. See Appendix A, Base Operating System Error Codes for Services That Require Path-Name Resolution for a list of additional error codes.

If NFS is installed on the system, the `chdir` subroutine can also fail if the following is true:

- **ETIMEDOUT**  
  The connection timed out.

Related Information
The `chroot` subroutine.

The `cd` command.

Files, Directories, and File Systems for Programmers in AIX 5L Version 5.3 General Programming Concepts: Writing and Debugging Programs.

chmod or fchmod Subroutine

Purpose
Changes file system object’s base file mode bits.

Library
Standard C Library (`libc.a`)

Syntax
```
#include <sys/stat.h>

int chmod (const char *Path, mode_t Mode);
int fchmod (int FileDescriptor, Mode);
```

Description
The `chmod` subroutine sets the access permissions of the file specified by the `Path` parameter. If Network File System (NFS) is installed on your system, this path can cross into another node.
Use the \texttt{fchmod} subroutine to set the access permissions of an open file pointed to by the \textit{FileDescriptor} parameter.

If \textit{FileDescriptor} references a shared memory object, the \texttt{fchmod} subroutine affects the \texttt{S\_IRUSR}, \texttt{S\_IWUSR}, \texttt{S\_IRGRP}, \texttt{S\_IWGRP}, \texttt{S\_IROTH}, and \texttt{S\_IWOTH} file permission bits.

The access control information is set according to the \textit{Mode} parameter. Note that these routines will replace any existing ACL associated with the file system object.

\section*{Parameters}

\begin{itemize}
\item \textit{FileDescriptor} \hspace{1cm} Specifies the file descriptor of an open file or shared memory object.
\end{itemize}
**Mode**

Specifies the bit pattern that determines the access permissions. The *Mode* parameter is constructed by logically ORing one or more of the following values, which are defined in the `sys/mode.h` file:

- **S_ISUID**
  Enables the *setuid* attribute for an executable file. A process executing this program acquires the access rights of the owner of the file.

- **S_ISGID**
  Enables the *setgid* attribute for an executable file. A process executing this program acquires the access rights of the group of the file. Also, enables the group-inheritance attribute for a directory. Files created in this directory have a group equal to the group of the directory.

The following attributes apply only to files that are directly executable. They have no meaning when applied to executable text files such as shell scripts and *awk* scripts.

- **S_ISVTX**
  Enables the *link/unlink* attribute for a directory. Files cannot be linked to in this directory. Files can only be unlinked if the requesting process has write permission for the directory and is either the owner of the file or the directory.

- **S_ISVTX**
  Enables the *save text* attribute for an executable file. The program is not unmapped after usage.

- **S_ENFMT**
  Enables enforcement-mode record locking for a regular file. File locks requested with the *lockf* subroutine are enforced.

- **S_IRUSR**
  Permits the file’s owner to read it.

- **S_IWUSR**
  Permits the file’s owner to write to it.

- **S_IXUSR**
  Permits the file’s owner to execute it (or to search the directory).

- **S_IRGRP**
  Permits the file’s group to read it.

- **S_IWGRP**
  Permits the file’s group to write to it.

- **S_IXGRP**
  Permits the file’s group to execute it (or to search the directory).

- **S_IROTH**
  Permits others to read the file.

- **S_IWOTH**
  Permits others to write to the file.

- **S_IXOTH**
  Permits others to execute the file (or to search the directory).

Other mode values exist that can be set with the *mknod* subroutine but not with the *chmod* subroutine.

**Path**

Specifies the full path name of the file.

**Return Values**

Upon successful completion, the *chmod* subroutine and *fchmod* subroutines return a value of 0. If the *chmod* subroutine or *fchmod* subroutine is unsuccessful, a value of -1 is returned, and the *errno* global variable is set to identify the error.
Error Codes

The chmod subroutine is unsuccessful and the file permissions remain unchanged if one of the following is true:

- **ENOTDIR**: A component of the Path prefix is not a directory.
- **EACCES**: Search permission is denied on a component of the Path prefix.
- **EFAULT**: The Path parameter points to a location outside of the allocated address space of the process.
- **ELOOP**: Too many symbolic links were encountered in translating the Path parameter.
- **ENOENT**: The named file does not exist.
- **ENAMETOOLONG**: A component of the Path parameter exceeded 255 characters, or the entire Path parameter exceeded 1023 characters.

The fchmod subroutine is unsuccessful and the file permissions remain unchanged if one of the following is true:

- **EBADF**: The value of the FileDescriptor parameter is not valid.

The chmod or fchmod subroutine is unsuccessful and the access control information for a file remains unchanged if one of the following is true:

- **EPERM**: The effective user ID does not match the owner of the file, and the process does not have appropriate privileges.
- **EROFS**: The named file resides on a read-only file system.
- **EIO**: An I/O error occurred during the operation.

If NFS is installed on your system, the chmod and fchmod subroutines can also be unsuccessful if the following is true:

- **ESTALE**: The root or current directory of the process is located in a virtual file system that has been unmounted.
- **ETIMEDOUT**: The connection timed out.

Security

Access Control: The invoker must have search permission for all components of the Path prefix.

If you receive the EBUSY error, toggle the enforced locking attribute in the Mode parameter and retry your operation. The enforced locking attribute should never be used on a file that is part of the Trusted Computing Base.

Related Information

The acl_chg subroutine, acl_get subroutine, acl_set subroutine, aclx_get subroutine, aclx_set subroutine, chacl subroutine, chown subroutine, fchown subroutine, fchownx subroutine, fchacl subroutine, lchown subroutine, lchownx subroutine, fchown subroutine, fchownx subroutine, lchown subroutine, lchownx subroutine, chown subroutine, fchown subroutine, fchownx subroutine, lchown subroutine, lchownx subroutine, and the aclget command, aclput command, chmod command.

List of Security and Auditing Subroutines and Subroutines Overview in AIX 5L Version 5.3 General Programming Concepts: Writing and Debugging Programs.

chown, fchown, lchown, chownx, or fchownx Subroutine

Purpose

Changes file ownership.

Library

Standard C Library (libc.a)
Syntax

Syntax for the chown, fchown, and lchown Subroutines:

```c
#include <sys/types.h>
#include <unistd.h>

int chown (const char *Path, uid_t Owner, gid_t Group);
int fchown (FileDescriptor, uid_t Owner, gid_t Group);
int lchown (const char *fname, uid_t uid, gid_t gid);

Syntax for the chownx and fchownx Subroutines:

#include <sys/types.h>
#include <sys/chownx.h>

int chownx (Path, Owner, Group, Flags);
int fchownx (FileDescriptor, Owner, Group, Flags);
```

Description

The chown, chownx, fchown, fchownx, and lchown subroutines set the file owner and group IDs of the specified file system object. Root user authority is required to change the owner of a file.

A function lchown function sets the owner ID and group ID of the named file similarity to chown function except in the case where the named file is a symbolic link. In this case lchown function changes the ownership of the symbolic link file itself, while chown function changes the ownership of the file or directory to which the symbolic link refers.

Parameters

FileDescriptor       Specifies the file descriptor of an open file.
Flags

Specifies whether the file owner ID or group ID should be changed. This parameter is constructed by logically ORing the following values:

- **T_OWNER_AS_IS**
  - Ignores the value specified by the `Owner` parameter and leaves the owner ID of the file unaltered.

- **T_GROUP_AS_IS**
  - Ignores the value specified by the `Group` parameter and leaves the group ID of the file unaltered.

Group

Specifies the new group of the file. For the `chown`, `fchown`, and `lchown` commands, if this value is -1, the group is not changed. (A value of -1 indicates only that the group is not changed; it does not indicate a group that is not valid. An owner or group ID cannot be invalid.) For the `chownx` and `fchownx` commands, the subroutines change the Group to -1 if -1 is supplied for Group and `T_GROUP_AS_IS` is not set.

Owner

Specifies the new owner of the file. For the `chown`, `fchown`, and `lchown` commands, if this value is -1, the group is not changed. (A value of -1 indicates only that the group is not changed; it does not indicate a group that is not valid. An owner or group ID cannot be invalid.) For the `chownx` and `fchownx` commands, the subroutines change the Owner to -1 if -1 is supplied for Owner and `T_OWNER_AS_IS` is not set.

Path

Specifies the full path name of the file. If `Path` resolves to a symbolic link, the ownership of the file or directory pointed to by the symbolic link is changed.

Return Values

Upon successful completion, the `chown`, `chownx`, `fchown`, `fchownx`, and `lchown` subroutines return a value of 0. If the `chown`, `chownx`, `fchown`, `fchownx`, or `lchown` subroutine is unsuccessful, a value of -1 is returned and the `errno` global variable is set to indicate the error.

Error Codes

The `chown`, `chownx`, or `lchown` subroutine is unsuccessful and the owner and group of a file remain unchanged if one of the following is true:

- **EACCESS**
  - Search permission is denied on a component of the `Path` parameter.

- **EDQUOT**
  - The new group for the file system object cannot be set because the group’s quota of disk blocks or i-nodes has been exhausted on the file system.

- **EFAULT**
  - The `Path` parameter points to a location outside of the allocated address space of the process.

- **EINVAL**
  - The owner or group ID supplied is not valid.

- **ELOOP**
  - Too many symbolic links were encountered in translating the `Path` parameter.

- **ENAMETOOLONG**
  - A component of the `Path` parameter exceeded 255 characters, or the entire `Path` parameter exceeded 1023 characters.

- **ENOENT**
  - A symbolic link was named, but the file to which it refers does not exist; or a component of the `Path` parameter does not exist; or the process has the `disallow truncation` attribute set; or the `Path` parameter is null.

- **ENOTDIR**
  - A component of the path prefix is not a directory.

- **EPERM**
  - The effective user ID does not match the owner of the file, and the calling process does not have the appropriate privileges.

- **EROSFS**
  - The named file resides on a read-only file system.

- **ESTALE**
  - The root or current directory of the process is located in a virtual file system that has been unmounted.

The `fchown` or `fchownx` subroutine is unsuccessful and the file owner and group remain unchanged if one of the following is true:

- **EBADF**
  - The named file resides on a read-only file system.
The new group for the file system object cannot be set because the group’s quota of disk blocks or i-nodes has been exhausted on the file system.

EDQUOT

An I/O error occurred during the operation.

EIO

Security

Access Control: The invoker must have search permission for all components of the Path parameter.

chpass Subroutine

Purpose

Changes user passwords.

Library

Standard C Library (libc.a)

Thread Safe Security Library (libs_r.a)

Syntax

```c
int chpass (UserName, Response, Reenter, Message)
char *UserName;
char *Response;
int *Reenter;
char **Message;
```

Description

The chpass subroutine maintains the requirements that the user must meet to change a password. This subroutine is the basic building block for changing passwords and handles password changes for local, NIS, and DCE user passwords.

The Message parameter provides a series of messages asking for old and new passwords, or providing informational messages, such as the reason for a password change failing. The first Message prompt is a prompt for the old password. This parameter does not prompt for the old password if the user has a real user ID of 0 (zero) and is changing a local user, or if the user has no current password. The chpass subroutine does not prompt a user with root authority for an old password. It informs the program that no message was sent and that it should invoke chpass again. If the user satisfies the first Message parameter’s prompt, the system prompts the user to enter the new password. Each message is contained in the Message parameter and is displayed to the user. The Response parameter returns the user’s response to the chpass subroutine.

The Reenter parameter indicates when a user has satisfied all prompt messages. The parameter remains nonzero until a user has passed all prompts. After the returned value of Reenter is 0, the return code signals whether the password change has succeeded or failed. When progressing through prompts for a user, the value of Reenter must be maintained by the caller between invocations of chpass.

The chpass subroutine maintains internal state information concerning the next prompt message to present to the user. If the calling program supplies a different user name before all prompt messages are complete for the user, the internal state information is reset and prompt messages begin again. State information is also kept in the Reenter variable. The calling program must maintain the value of Reenter between calls to chpass.

The chpass subroutine determines the administration domain to use during password changes. It determines if the user is defined locally, defined in Network Information Service (NIS), or defined in...
Distributed Computing Environment (DCE). Password changes occur only in these domains. System administrators may override this convention with the registry value in the `/etc/security/user` file. If the registry value is defined, the password change can only occur in the specified domain. System administrators can use this registry value if the user is administered on a remote machine that periodically goes down. If the user is allowed to log in through some other authentication method while the server is down, password changes remain to follow only the primary server.

The `chpass` subroutine allows the user to change passwords in two ways. For normal (non-administrative) password changes, the user must supply the old password, either on the first call to the `chpass` subroutine or in response to the first message from `chpass`. If the user is root, real user ID of 0, local administrative password changes are handled by supplying a null pointer for the `Response` parameter during the initial call.

Users that are not administered locally are always queried for their old password.

The `chpass` subroutine is always in one of the following states:
1. Initial state: The caller invokes the `chpass` subroutine with NULL `response` parameter and receives the initial password prompt in the `message` parameter.
2. Verify initial password: The caller invokes the `chpass` subroutine with the results of prompting the user with earlier `message` parameter as the `response` parameter. The caller is given a prompt to enter the new password in the `message` parameter.
3. Enter new password: The caller invokes the `chpass` subroutine with the results of prompting user with the new password prompt in the `response` parameter. The caller will be given a prompt to repeat the new password in the `message` parameter.
4. Verify new password: The caller invokes the `chpass` subroutine with the results of prompting the user to repeat the new password in the `response` parameter. The `chpass` subroutine then performs the actual password change.

Any step in the above process can result in the `chpass` subroutine terminating the dialog. This is signalled when the `reenter` variable is set to 0. The return code indicates the nature of the failure.

**Note:** Set the setuid and owner to root for your own programs that use the `chpass` subroutine.

**Parameters**

- **UserName**: Specifies the user’s name whose password is to be changed.
- **Response**: Specifies a character string containing the user’s response to the last prompt.
- **Reenter**: Points to a Boolean value used to signal whether the `chpass` subroutine has completed processing. If the `Reenter` parameter is a nonzero value, the `chpass` subroutine expects the user to satisfy the prompt message provided by the `Message` parameter. If the `Reenter` parameter is 0, the `chpass` subroutine has completed processing.
- **Message**: Points to a pointer that the `chpass` subroutine allocates memory for and fills in. This replacement string is then suitable for printing and issues challenge messages (if the `Reenter` parameter is a nonzero value). The string can also issue informational messages such as why the user failed to change the password (if the `Reenter` parameter is 0). The calling application is responsible for freeing this memory.

**Return Values**

Upon successful completion, the `chpass` subroutine returns a value of 0. If the `chpass` subroutine is unsuccessful, it returns the following values:

- `-1`: Indicates the call failed in the thread safe library `libs_r.a`. `ERRNO` will indicate the failure code.
- `1`: Indicates that the password change was unsuccessful and the user should attempt again. This return value occurs if a password restriction is not met, such as if the password is not long enough.
Indicates that the password change was unsuccessful and the user should not attempt again. This return value occurs if the user enters an incorrect old password or if the network is down (the password change cannot occur).

**Error Codes**
The `chpass` subroutine is unsuccessful if one of the following values is true:

- **ENOENT**: Indicates that the user cannot be found.
- **ESAD**: Indicates that the user did not meet the criteria to change the password.
- **EPERM**: Indicates that the user did not have permission to change the password.
- **EINVAL**: Indicates that the parameters are not valid.
- **ENOMEM**: Indicates that memory allocation (malloc) failed.

**Related Information**
The `authenticate` subroutine.

---

### chpassx Subroutine

**Purpose**
Changes multiple method passwords.

**Library**
Standard C Library (`libc.a`)

Thread Safe Security Library (`libs_r.a`)

**Syntax**

```c
int chpassx (UserName, Response, Reenter, Message, State)
char * UserName;
char * Response;
int * Reenter;
char ** Message;
void ** State;
```

**Description**
The `chpassx` subroutine maintains the requirements that the user must meet to change a password. This subroutine is the basic building block for changing passwords, and it handles password changes for local, NIS, and loadable authentication module user passwords. It uses information provided by the `authenticate` and `passwdexpired` subroutines to indicate which passwords were used when a user authenticated and whether or not those passwords are expired.

The `Message` parameter provides a series of messages asking for old and new passwords, or providing informational messages, such as the reason for a password change failing. The first `Message` prompt is a prompt for the old password. This parameter does not prompt for the old password if the user has a real user ID of 0 and is changing a local user, or if the user has no current password. The `chpassx` subroutine does not prompt a user with root authority for an old password when only a local password is being changed. It informs the program that no message was sent and that it should invoke `chpass` again. If the user satisfies the first `Message` parameter’s prompt, the system prompts the user to enter the new password. Each message is contained in the `Message` parameter and is displayed to the user. The `Response` parameter returns the user’s response to the `chpass` subroutine.
The Reenter parameter remains a nonzero value until the user satisfies all of the prompt messages or until the user incorrectly responds to a prompt message. When the Reenter parameter is 0, the return code signals whether the password change completed or failed. The calling application must initialize the Reenter parameter to 0 before the first call to the chpassx subroutine and the application cannot modify the Reenter parameter until the sequence of chpassx subroutine calls has completed.

The authenticatex subroutine ascertains the authentication domains the user can attempt. The subroutine uses the SYSTEM attribute for the user. Each token that is displayed in the SYSTEM line corresponds to a method that can be dynamically loaded and processed. Likewise, the system can provide multiple or alternate authentication paths.

The State parameter contains information from an earlier call to the authenticatex or passwdexpiredx subroutines. That information indicates which administration domains were used when the user was authenticated and which passwords have expired and can be changed by the user. The State parameter must be initialized to null when the chpassx subroutine is not being called after an earlier call to the authenticatex or passwdexpiredx subroutines, or if the calling program does not wish to use the information from an earlier call.

The chpassx subroutine maintains internal state information concerning the next prompt message to present to the user. If the calling program supplies a different user name before all prompt messages are complete for the user, the internal state information is reset and prompt messages begin again.

The chpassx subroutine determines the administration domain to use during password changes. It determines if the user is defined locally, defined in Network Information Service (NIS), defined in Distributed Computing Environment (DCE), or defined in another administrative domain supported by a loadable authentication module. Password changes use the user's SYSTEM attribute and information in the State parameter. When the State parameter includes information from an earlier call to the authenticatex subroutine, only the administrative domains that were used for authentication are changed. When the State parameter includes information from an earlier call to the passwdexpiredx subroutine, only the administrative domains that have expired passwords are changed. The State parameter can contain information from calls to both authenticatex and passwdexpiredx, in which case passwords that were used for authentication are changed, even if they are not expired, so that passwords remain synchronized between administrative domains.

The chpassx subroutine allows the user to change passwords in two ways. For normal (nonadministrative) password changes, the user must supply the old password, either on the first call to the chpassx subroutine or in response to the first message from chpassx. If the user is root (with a real user ID of 0), local administrative password changes are handled by supplying a null pointer for the Response parameter during the initial call.

Users that are not administered locally are always queried for their old password.

The chpassx subroutine is always in one of three states: entering the old password, entering the new password, or entering the new password again. If any of these states do not need to be complied with, the chpassx subroutine returns a null challenge.

Parameters

**Message**

Points to a pointer that the chpassx subroutine allocates memory for and fills in. This replacement string is then suitable for printing and issues challenge messages (if the Reenter parameter is a nonzero value). The string can also issue informational messages, such as why the user failed to change the password (if the Reenter parameter is 0). The calling application is responsible for freeing this memory.
Reenter Points to an integer value used to signal whether the chpassx subroutine has completed processing. If the Reenter parameter is a nonzero value, the chpassx subroutine expects the user to satisfy the prompt message provided by the Message parameter. If the Reenter parameter is 0, the chpassx subroutine has completed processing.

Response Specifies a character string containing the user’s response to the last prompt.

State Points to a pointer that the chpassx subroutine allocates memory for and fills in. The State parameter can also be the result of an earlier call to the authenticatex or passwdexpiredx subroutines. This parameter contains information about each password that has been changed for the user. The calling application is responsible for freeing this memory after the chpassx subroutine has completed.

UserName Specifies the user’s name whose password is to be changed.

Return Values
Upon successful completion, the chpassx subroutine returns a value of 0. If this subroutine fails, it returns the following values:

-1 The call failed in the libs_r.a thread safe library. errno indicates the failure code.
1 The password change was unsuccessful and the user should try again. This return value occurs if a password restriction is not met (for example, the password is not long enough).
2 The password change was unsuccessful and the user should not try again. This return value occurs if the user enters an incorrect old password or if the network is down (the password change cannot occur).

Error Codes
The chpassx subroutine is unsuccessful if one of the following values is true:

EINVAL The parameters are not valid.
ENOENT The user cannot be found.
ENOMEM Memory allocation (malloc) failed.
EPERM The user did not have permission to change the password.
ESAD The user did not meet the criteria to change the password.

Related Information
The authenticatex Subroutine on page 115, passwdexpiredx Subroutine on page 964.

chprojattr Subroutine

Purpose
Updates and modifies the project attributes in kernel project registry for the given project.

Library
The libaacct.a library.

Syntax
<sys/aacct.h>
chprojattr(struct project *, int cmd)

Description
The chprojattr subroutine alters the attributes of a project defined in the kernel project registry. A pointer to struct project containing the project definition and the operation command is sent as input arguments. The following operations are permitted:
- PROJ_ENABLE_AGGR - Enables aggregation for the specified project
- PROJ_DISABLE_AGGR - Disables aggregation for the specified project

If PROJ_ENABLE_AGGR is passed, then the aggregation status bit is set to 1. If PROJ_DISABLE_AGGR is passed, then the aggregation status bit set to 0.

**Note:** To initialize the project structure, the user must call the `getprojdef` subroutine before calling the `chprojattr` subroutine.

**Parameters**

- `project` Pointer containing the project definition.
- `cmd` An integer command indicating whether to perform a set or clear operation.

**Security**

Only for privileged users. Privilege can be extended to nonroot users by granting the CAP_AACCT capability to a user.

**Return Values**

- 0 Success
- -1 Failure

**Error Codes**

- EINVAL Invalid arguments passed. The passed command flag is invalid or the passed pointer is NULL.
- ENONENT Project not found.

**Related Information**

The "addproj Subroutine" on page 31, "chprojattrdb Subroutine," "getproj Subroutine" on page 413, "getprojs Subroutine" on page 415, `rmproj Subroutine`.

---

### chprojattrdb Subroutine

**Purpose**

Updates the project attributes in the project database.

**Library**

The `libaacct.a` library.

**Syntax**

```c
#include <sys/aacct.h>

chprojattrdb(void *handle, struct project *project, int cmd)
```

**Description**

The `chprojattrdb` subroutine alters the attributes of the named project in the specified project database, which is controlled through the `handle` parameter. The following commands are permitted:

- **PROJ_ENABLE_AGGR** — Enables aggregation for the specified project
• **PROJ_DISABLE_AGGR** — Disables aggregation for the specified project

The project database must be initialized before calling this subroutine. The `projdballoc` subroutine is provided for this purpose. The `chprojattrdb` subroutine must be called after the `getprojdb` subroutine, which sets the record pointer to point to the project that needs to be modified.

**Note:** The `chprojattrdb` subroutine must be called after the `getprojdb` subroutine, which makes the named project the current project.

### Parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>handle</code></td>
<td>Pointer to the handle allocated for the project database.</td>
</tr>
<tr>
<td><code>project</code></td>
<td>Pointer containing the project definition.</td>
</tr>
<tr>
<td><code>cmd</code></td>
<td>An integer command indicating whether to perform a set or clear operation.</td>
</tr>
</tbody>
</table>

### Security

Only for privileged users. Privilege can be extended to nonroot users by granting the CAP_AACCT capability to a user.

### Return Values

<table>
<thead>
<tr>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Success</td>
</tr>
<tr>
<td>-1</td>
<td>Failure</td>
</tr>
</tbody>
</table>

### Error Codes

- **EINVAL** — Invalid arguments passed. The passed command flag is invalid or the passed pointer is NULL.
- **ENONENT** — Project not found.

### Related Information

The following subroutines are related:
- `addprojdb Subroutine` on page 32
- `chprojattr Subroutine` on page 158
- `getfirstprojdb Subroutine` on page 363
- `getnextprojdb Subroutine` on page 391
- `getprojdb Subroutine` on page 414
- `projdballoc Subroutine` on page 1158
- `projdbfinit Subroutine` on page 1159
- `projdbfree Subroutine` on page 1160
- `rmprojdb Subroutine`

---

### chroot Subroutine

**Purpose**

Changes the effective root directory.

**Library**

Standard C Library (`libc.a`)

**Syntax**

```c
#include <unistd.h>

int chroot (const char *Path);
char *Path;
```
Description
The chroot subroutine causes the directory named by the Path parameter to become the effective root directory. If the Path parameter refers to a symbolic link, the chroot subroutine sets the effective root directory to the directory pointed to by the symbolic link. If Network File System (NFS) is installed on your system, this path can cross into another node.

The effective root directory is the starting point when searching for a file’s path name that begins with / (slash). The current directory is not affected by the chroot subroutine.

The calling process must have root user authority in order to change the effective root directory. The calling process must also have search access to the new effective root directory.

The .. (double period) entry in the effective root directory is interpreted to mean the effective root directory itself. Thus, this directory cannot be used to access files outside the subtree rooted at the effective root directory.

Parameters
Path Pointer to the new effective root directory.

Return Values
Upon successful completion, a value of 0 is returned. Otherwise, a value of -1 is returned and the errno global variable is set to indicate the error.

Error Codes
The chroot subroutine fails and the effective root directory remains unchanged if one or more of the following are true:

- ENOENT The named directory does not exist.
- EACCES The named directory denies search access.
- EPERM The process does not have root user authority.

The chroot subroutine can be unsuccessful for other reasons. See Appendix A, Base Operating System Error Codes for Services that Require Path-Name Resolution (Appendix A, “Base Operating System Error Codes for Services That Require Path-Name Resolution,” on page 1323) for a list of additional errors.

If NFS is installed on the system, the chroot subroutine can also fail if the following is true:

- ETIMEDOUT The connection timed out.

Related Information
The chdir (“chdir Subroutine” on page 147) subroutine.

The chroot command.


Files, Directories, and File Systems for Programmers in AIX 5L Version 5.3 General Programming Concepts: Writing and Debugging Programs.
chssys Subroutine

Purpose
Modifies the subsystem objects associated with the SubsystemName parameter.

Library
System Resource Controller Library (libs.a)

Syntax
#include <sys/srcobj.h>
#include <spc.h>

int chssys(SubsystemName, SRCSubsystem)
char *SubsystemName;
struct SRCsubsys *SRCSubsystem;

Description
The chssys subroutine modifies the subsystem objects associated with the specified subsystem with the values in the SRCSubsys structure. This action modifies the objects associated with subsystem in the following object classes:
- Subsystem Environment
- Subserver Type
- Notify

The Subserver Type and Notify object classes are updated only if the subsystem name has been changed.

The SRCSubsys structure is defined in the /usr/include/sys/srcobj.h file.

The program running with this subroutine must be running with the group system.

Parameters
SRCSubsystem Points to the SRCSubsys structure.
SubsystemName Specifies the name of the subsystem.

Return Values
Upon successful completion, the chssys subroutine returns a value of 0. Otherwise, it returns a value of -1 and the odmerrno variable is set to indicate the error, or a System Resource Controller (SRC) error code is returned.

Error Codes
The chssys subroutine is unsuccessful if one or more of the following are true:
- SRC_NONAME No subsystem name is specified.
- SRC_NOPATH No subsystem path is specified.
- SRC_BADNSIG Invalid stop normal signal.
- SRC_BADFSIG Invalid stop force signal.
- SRC_NOCONTACT Contact not signal, sockets, or message queues.
- SRC_SSME Subsystem name does not exist.
- SRC_SUBEXIST New subsystem name is already on file.
- SRC_SYNEXIST New subsystem synonym name is already on file.
The specified SRCsubsys record does not exist.

Subsystem name is too long.

Synonym name is too long.

Command arguments are too long.

Subsystem path is too long.

stdin path is too long.

stdout path is too long.

stderr path is too long.

Group name is too long.

Security

Privilege Control: This command has the Trusted Path attribute. It has the following kernel privilege:

ATTR_SET_PROC_AUDIT kernel privilege

Files Accessed:

<table>
<thead>
<tr>
<th>Mode</th>
<th>File</th>
</tr>
</thead>
<tbody>
<tr>
<td>644</td>
<td>/etc/objrepos/SRCsubsys</td>
</tr>
<tr>
<td>644</td>
<td>/etc/objrepos/SRCsubsvr</td>
</tr>
<tr>
<td>644</td>
<td>/etc/objrepos/SRCnotify</td>
</tr>
</tbody>
</table>

Auditing Events:

<table>
<thead>
<tr>
<th>Event</th>
<th>Information</th>
</tr>
</thead>
<tbody>
<tr>
<td>SRC_Chssys</td>
<td></td>
</tr>
</tbody>
</table>

Files

/etc/objrepos/SRCsubsys  SRC Subsystem Configuration object class.
/etc/objrepos/SRCsubsvr  SRC Subserver Configuration object class.
/etc/objrepos/SRCnotify  SRC Notify Method object class.
/dev/SRC                   Specifies the AF_UNIX socket file.
/dev/.SRC-unix             Specifies the location for temporary socket files.

Related Information

The addssys (“addssys Subroutine” on page 33) subroutine, delssys (“delssys Subroutine” on page 211) subroutine.

The chssys command, mkssys command, rmssys command.

System Resource Controller in Operating system and device management.

Defining Your Subsystem to the SRC | List of SRC Subroutines | System Resource Controller (SRC)
Overview for Programmers in AIX 5L Version 5.3 General Programming Concepts: Writing and Debugging Programs.

cimag, cimagf, or cimagl Subroutine

Purpose

Performs complex imaginary computations.
Syntax
#include <complex.h>

double cimag (z)
double complex z;

float cimagf (z)
float complex z;

long double cimagl (z)
long double complex z;

Description
The cimag, cimagf, and cimagl subroutines compute the imaginary part of z.

Parameters
z Specifies the value to be computed.

Return Values
The cimag, cimagf, and cimagl subroutines return the imaginary part value (as a real).

Related Information
“carg, cargf, or cargl Subroutine” on page 131, “conj, conjf, or conjl Subroutine” on page 182, “cproj, cprojf,
or cprojl Subroutine” on page 189, and “creal, crealf, or creall Subroutine” on page 190.

ckuseracct Subroutine

Purpose
Checks the validity of a user account.

Library
Security Library (libc.a)

Syntax
#include <login.h>

int ckuseracct (Name, Mode, TTY)
char *Name;
int Mode;
char *TTY;

Description
Note: This subroutine is obsolete and is provided only for backwards compatibility. Use the
loginrestrictions subroutine, which performs a superset of the functions of the ckuseracct
subroutine, instead.

The ckuseracct subroutine checks the validity of the user account specified by the Name parameter. The
Mode parameter gives the mode of the account usage, and the TTY parameter defines the terminal being
used for the access. The ckuseracct subroutine checks for the following conditions:
• Account existence
• Account expiration

The *Mode* parameter specifies other mode-specific checks.

**Parameters**

<table>
<thead>
<tr>
<th>Name</th>
<th>Specifies the login name of the user whose account is to be validated.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Mode</td>
<td>Specifies the manner of usage. Valid values as defined in the <code>login.h</code> file are listed below. The <em>Mode</em> parameter must be one of these or 0:</td>
</tr>
<tr>
<td></td>
<td><code>-S_LOGIN</code> - Verifies that local logins are permitted for this account.</td>
</tr>
<tr>
<td></td>
<td><code>-S_SU</code> - Verifies that the <code>su</code> command is permitted and that the current process has a group ID that can invoke the <code>su</code> command to switch to the account.</td>
</tr>
<tr>
<td></td>
<td><code>-S_DAEMON</code> - Verifies the account can be used to invoke daemon or batch programs using the <code>src</code> or <code>cron</code> subsystems.</td>
</tr>
<tr>
<td></td>
<td><code>-S_RLOGIN</code> - Verifies the account can be used for remote logins using the <code>rlogind</code> or <code>telnetd</code> programs.</td>
</tr>
<tr>
<td>TTY</td>
<td>Specifies the terminal of the originating activity. If this parameter is a null pointer or a null string, no TTY origin checking is done.</td>
</tr>
</tbody>
</table>

**Security**

Files Accessed:

<table>
<thead>
<tr>
<th>Mode</th>
<th>File</th>
</tr>
</thead>
<tbody>
<tr>
<td>r</td>
<td><code>/etc/passwd</code></td>
</tr>
<tr>
<td>r</td>
<td><code>/etc/security/user</code></td>
</tr>
</tbody>
</table>

**Return Values**

If the account is valid for the specified usage, the `ckuseracct` subroutine returns a value of 0. Otherwise, a value of -1 is returned and the `errno` global variable is set to the appropriate error code.

**Error Codes**

The `ckuseracct` subroutine fails if one or more of the following are true:

- `ENOENT` - The user specified in the *Name* parameter does not have an account.
- `ESTALE` - The user’s account is expired.
- `EACCES` - The specified terminal does not have access to the specified account.
- `EACCES` - The *Mode* parameter is `S_SU`, and the current process is not permitted to use the `su` command to access the specified user.
- `EACCES` - Access to the account is not permitted in the specified *Mode*.
- `EINVAL` - The *Mode* parameter is not one of `S_LOGIN`, `S_SU`, `S_DAEMON`, `S_RLOGIN`.

**Related Information**

The `ckuserID` subroutine, `getpcred` subroutine, `getpenv` subroutine, `setpcrd` subroutine, `setpenv` subroutine.

The `login` command, `rlogin` command, `su` command, `telnet` command.
ckuserID Subroutine

Purpose
Authenticates the user.

Note: This subroutine is obsolete and is provided for backwards compatibility. Use the authenticate subroutine, instead.

Library
Security Library (libc.a)

Syntax
#include <login.h>
int ckuserID (User, Mode);
int Mode;
char *User;

Description
The ckuserID subroutine authenticates the account specified by the User parameter. The mode of the authentication is given by the Mode parameter. The login and su commands continue to use the ckuserID subroutine to process the /etc/security/user auth1 and auth2 authentication methods.

The ckuserID subroutine depends on the authenticate subroutine to process the SYSTEM attribute in the /etc/security/user file. If authentication is successful, the passwdexpired subroutine is called.

Errors caused by grammar or load modules during a call to the authenticate subroutine are displayed to the user if the user was authenticated. These errors are audited with the USER_Login audit event if the user failed authentication.

Parameters

User Specifies the name of the user to be authenticated.
Mode Specifies the mode of authentication. This parameter is a bit mask and may contain one or more of the following values, which are defined in the login.h file:

S_PRIMARY
The primary authentication methods defined for the User parameter are checked. All primary authentication checks must be passed.

S_SECONDARY
The secondary authentication methods defined for the User parameter are checked. Secondary authentication checks are not required to be successful.

Primary and secondary authentication methods for each user are set in the /etc/security/user file by defining the auth1 and auth2 attributes. If no primary methods are defined for a user, the SYSTEM attribute is assumed. If no secondary methods are defined, there is no default.
Security

Files Accessed:

<table>
<thead>
<tr>
<th>Mode</th>
<th>File</th>
</tr>
</thead>
<tbody>
<tr>
<td>r</td>
<td>/etc/passwd</td>
</tr>
<tr>
<td>r</td>
<td>/etc/security/passwd</td>
</tr>
<tr>
<td>r</td>
<td>/etc/security/user</td>
</tr>
<tr>
<td>r</td>
<td>/etc/security/login.cfg</td>
</tr>
</tbody>
</table>

Return Values

If the account is valid for the specified usage, the ckuserID subroutine returns a value of 0. Otherwise, a value of -1 is returned and the errno global variable is set to indicate the error.

Error Codes

The ckuserID subroutine fails if one or more of the following are true:

- **ESAD**: Security authentication failed for the user.
- **EINVAL**: The Mode parameter is neither S_PRIMARY nor SSECONDARY or the Mode parameter is both S_PRIMARY and SSECONDARY.

Related Information

The authenticate ("authenticate Subroutine" on page 113) subroutine, ckuseracct ("ckuseracct Subroutine" on page 164) subroutine, getpcred ("getpcred Subroutine" on page 398) subroutine, getpenv ("getpenv Subroutine" on page 400) subroutine, passwdexpired ("passwdexpired Subroutine" on page 963) subroutine, setpcred subroutine, setpenv subroutine.

The login command, su command.

List of Security and Auditing Subroutines and Subroutines Overview in AIX 5L Version 5.3 General Programming Concepts: Writing and Debugging Programs.

class, _class, finite, isnan, or unordered Subroutines

Purpose

Determines classifications of floating-point numbers.

Libraries

IEEE Math Library (libm.a)
or System V Math Library (libmsaa.a)

Syntax

```c
#include <math.h>
#include <float.h>

int class(x);
double x;
```

Base Operating System (BOS) Runtime Services (A-P) 167
int _class(x)
double x;
#include <math.h>
int finite(x)  
double x;
#include <math.h>
int isnan(x)   
double x;
#include <math.h>
int unordered(x, y)  
double x, y;

Description
The class subroutine, _class subroutine, finite subroutine, isnan subroutine, and unordered subroutine determine the classification of their floating-point value. The unordered subroutine determines if a floating-point comparison involving x and y would generate the IEEE floating-point unordered condition (such as whether x or y is a NaN).

The class subroutine returns an integer that represents the classification of the floating-point x parameter. Since class is a reversed key word in C++. The class subroutine can not be invoked in a C++ program. The _class subroutine is an interface for C++ program using the class subroutine. The interface and the return value for class and _class subroutines are identical. The values returned by the class subroutine are defined in the float.h header file. The return values are the following:

FP_PLUS_NORM                   Positive normalized, nonzero x
FP_MINUS_NORM                   Negative normalized, nonzero x
FP_PLUS_DENORM                  Positive denormalized, nonzero x
FP_MINUS_DENORM                 Negative denormalized, nonzero x
FP_PLUS_ZERO                    x = +0.0
FP_MINUS_ZERO                   x = -0.0
FP_PLUS_INF                     x = +INF
FP_MINUS_INF                    x = -INF
FP_NANS                         x = Signaling Not a Number (NaN)
FP_NANQ                         x = Quiet Not a Number (NaN)

Since class is a reserved keyword in C++, the class subroutine cannot be invoked in a C++ program. The _class subroutine is an interface for the C++ program using the class subroutine. The interface and the return values for class and _class subroutines are identical.

The finite subroutine returns a nonzero value if the x parameter is a finite number; that is, if x is not +-, INF, NaNQ, or NaN.

The isnan subroutine returns a nonzero value if the x parameter is an NaN or a NaNQ. Otherwise, it returns 0.

The unordered subroutine returns a nonzero value if a floating-point comparison between x and y would be unordered. Otherwise, it returns 0.

Note: Compile any routine that uses subroutines from the libm.a library with the -Im flag. To compile the class.c file, for example, enter:
cc class.c -Im
Parameters

\(x\)  Specifies some double-precision floating-point value.
\(y\)  Specifies some double-precision floating-point value.

Error Codes

The \texttt{finite}, \texttt{isnan}, and \texttt{unordered} subroutines neither return errors nor set bits in the floating-point exception status, even if a parameter is an NaN.

Related Information

List of Numerical Manipulation Services and Subroutines Overview in AIX 5L Version 5.3 General Programming Concepts: Writing and Debugging Programs.

clock Subroutine

Purpose

Reports central processing unit (CPU) time used.

Library

Standard C Library (\texttt{libc.a})

Syntax

```c
#include <time.h>
clock_t clock (void);
```

Description

The \texttt{clock} subroutine reports the amount of CPU time used. The reported time is the sum of the CPU time of the calling process and its terminated child processes for which it has executed \texttt{wait}, \texttt{system}, or \texttt{pclose} subroutines. To measure the amount of time used by a program, the \texttt{clock} subroutine should be called at the beginning of the program, and that return value should be subtracted from the return value of subsequent calls to the \texttt{clock} subroutine. To find the time in seconds, divide the value returned by the \texttt{clock} subroutine by the value of the macro \texttt{CLOCKS_PER_SEC}, which is defined in the \texttt{time.h} file.

Return Values

The \texttt{clock} subroutine returns the amount of CPU time used.

Related Information

The \texttt{getrusage}, \texttt{times} \texttt{("getrusage, getrusage64, times, or vtimes Subroutine" on page 423)} subroutine, \texttt{pclose} \texttt{("pclose Subroutine" on page 991)} subroutine, \texttt{system} subroutine, \texttt{vtimes} \texttt{("getrusage, getrusage64, times, or vtimes Subroutine" on page 423)} subroutine, \texttt{wait}, \texttt{waitpid}, \texttt{wait3} subroutine.

Subroutines Overview in AIX 5L Version 5.3 General Programming Concepts: Writing and Debugging Programs.

clock_getcpuclockid Subroutine

Purpose

Accesses a process CPU-time clock.
#include <time.h>
int clock_getcpuclockid(pid_t pid, clockid_t *clock_id);

### Description

The **clock_getcpuclockid** subroutine returns the clock ID of the CPU-time clock of the process specified by *pid*. If the process described by *pid* exists and the calling process has permission, the clock ID of this clock returns in *clock_id*.

If *pid* is zero, the **clock_getcpuclockid** subroutine returns the clock ID specified in *clock_id* of the CPU-time clock of the process making the call.

To obtain the CPU-time clock ID of other processes, the calling process should be root or have the same effective or real user ID as the process that owns the targeted CPU-time clock.

### Parameters

- **clock_id**
  Specifies the clock ID of the CPU-time clock.

- **pid**
  Specifies the process ID of the CPU-time clock.

### Return Values

Upon successful completion, the **clock_getcpuclockid** subroutine returns 0; otherwise, an error code is returned indicating the error.

### Error Codes

- **ENOTSUP**
  The function is not supported with checkpoint-restart processes.

- **EPERM**
  The requesting process does not have permission to access the CPU-time clock for the process.

- **ESRCH**
  No process can be found corresponding to the process specified by *pid*.

### Related Information

- **clock_getres, clock_gettime, and clock_settime Subroutine**

### Purpose

Clock and timer functions.

### Library

Standard C Library (**libc.a**)

### Syntax

```c
#include <time.h>

int clock_getres (clockid_t clock_id, struct timespec *res);

int clock_gettime (clockid_t clock_id, struct timespec *tp);
```
int clock_settime (clock_id, tp)
clockid_t clock_id;
const struct timespec *tp;

Description
The clock_getres subroutine returns the resolution of any clock. Clock resolutions are implementation-defined and cannot be set by a process. If the res parameter is not NULL, the resolution of the specified clock is stored in the location pointed to by the res parameter. If the res parameter is NULL, the clock resolution is not returned. If the time parameter of the clock_settime subroutine is not a multiple of the res parameter, the value is truncated to a multiple of the res parameter.

The clock_gettime subroutine returns the current value, tp, for the specified clock, clock_id.

The clock_settime subroutine sets the specified clock, clock_id, to the value specified by the tp parameter. Time values that are between two consecutive non-negative integer multiples of the resolution of the specified clock will be truncated down to the smaller multiple of the resolution.

A clock may be system-wide (visible to all processes) or per-process (measuring time that is meaningful only within a process). All implementations support a clock_id of CLOCK_REALTIME as defined in the time.h file. This clock represents the Realtime clock for the system. For this clock the values returned by the clock_gettime subroutine and specified by the clock_settime subroutine represent the amount of time (in seconds and nanoseconds) since the epoch.

If the value of the CLOCK_REALTIME clock is set through the clock_settime subroutine, the new value of the clock is used to determine the time of expiration for absolute time services based upon the CLOCK_REALTIME clock. This applies to the time at which armed absolute timers expire. If the absolute time requested at the invocation of such a time service is before the new value of the clock, the time service expires immediately as if the clock had reached the requested time normally.

Setting the value of the CLOCK_REALTIME clock through the clock_settime subroutine has no effect on threads that are blocked waiting for a relative time service based upon this clock, including the nanosleep subroutine; nor on the expiration of relative timers based upon this clock. Consequently, these time services expire when the requested relative interval elapses, independently of the new or old value of the clock.

A clock_id of CLOCK_MONOTONIC is defined in the time.h file. This clock represents the monotonic clock for the system. For this clock, the value returned by the clock_gettime subroutine represents the amount of time (in seconds and nanoseconds) since an unspecified point in the past. This point does not change after system start time (for example, this clock cannot have backward jumps). The value of the CLOCK_MONOTONIC clock cannot be set through the clock_settime subroutine. This subroutine fails if it is invoked with a clock_id parameter of CLOCK_MONOTONIC.

The calling process should have SYS_OPER authority to set the value of the CLOCK_REALTIME clock.

Process CPU-time clocks are supported by the system. For these clocks, the values returned by clock_gettime and specified by clock_settime represent the amount of execution time of the process associated with the clock. Clockid_t values for CPU-time clocks are obtained by calling clock_getcpuclid. A special clockid_t value, CLOCK_PROCESS_CPUTIME_ID, is defined in the time.h file. This value represents the CPU-time clock of the calling process when one of the clock_* or timer_* functions is called.

To get or set the value of a CPU-time clock, the calling process must have root permissions or have the same effective or real user ID as the process that owns the targeted CPU-time clock. The same rule applies to a process that tries to get the resolution of a CPU-time clock.
Thread CPU-time clocks are supported by the system. For these clocks, the values returned by `clock_gettime` and specified by `clock_settime` represent the amount of execution time of the thread associated with the clock. `Clockid_t` values for thread CPU-time clocks are obtained by calling the `pthread_getcpuclockid` subroutine. A special `clockid_t` value, `CLOCK_THREAD_CPUTIME_ID`, is defined in the `time.h` file. This value represents the thread CPU-time clock of the calling thread when one of the `clock_*()` or `timer_*` functions is called.

To get or set the value of a thread CPU-time clock, the calling thread must be a thread in the same process as the one that owns the targeted thread CPU-time clock. The same rule applies to a thread that tries to get the resolution of a thread CPU-time clock.

### Parameters

- **clock_id**: Specifies the clock.
- **res**: Stores the resolution of the specified clock.
- **tp**: Stores the current value of the specified clock.

### Return Values

If successful, 0 is returned. If unsuccessful, -1 is returned, and `errno` will be set to indicate the error.

### Error Codes

The `clock_getres`, `clock_gettime`, and `clock_settime` subroutines fail if:

- **EINVAL**: The `clock_id` parameter does not specify a known clock.
- **ENOTSUP**: The function is not supported with checkpoint-restart processes.

The `clock_settime` subroutine fails if:

- **EINVAL**: The `tp` parameter to the `clock_settime` subroutine is outside the range for the given clock ID.
- **EINVAL**: The `tp` parameter specified a nanosecond value less than zero or greater than or equal to 1000 million.
- **EINVAL**: The value of the `clock_id` argument is `CLOCK_MONOTONIC`.

The `clock_settime` subroutine might fail if:

- **EPERM**: The requesting process does not have the appropriate privilege to set the specified clock.

### Related Information

- [“clock_gettime Subroutine” on page 169](#)
- [“ctime, localtime, gmtime, mktime, difftime, asctime, or tzset Subroutine” on page 199](#)
- [“pthread_getcpuclockid Subroutine” on page 1231](#)
- [“nanosleep Subroutine” on page 887](#)


The `time command` in AIX 5L Version 5.3 Commands Reference, Volume 5.

---

**clock_nanosleep Subroutine**

### Purpose

Specifies clock for high resolution sleep.
Syntax
#include <time.h>
int clock_nanosleep(clockid_t clock_id, int flags,
                     const struct timespec *rqtp, struct timespec *rmtp);

Description
If the TIMER_ABSTIME flag is not set in the flags argument, the clock_nanosleep subroutine causes the current thread to be suspended from execution until either the time interval specified by the rqtp argument has elapsed, or a signal is delivered to the calling thread and its action is to invoke a signal-catching function, or the process is terminated. The clock_id argument specifies the clock used to measure the time interval.

If the TIMER_ABSTIME flag is set in the flags argument, the clock_nanosleep subroutine causes the current thread to be suspended from execution until either the time value of the clock specified by clock_id reaches the absolute time specified by the rqtp argument, or a signal is delivered to the calling thread and its action is to invoke a signal-catching function, or the process is terminated. If, at the time of the call, the time value specified by rqtp is less than or equal to the time value of the specified clock, then the clock_nanosleep subroutine returns immediately and the calling process shall not be suspended.

The suspension time caused by this function might be longer than requested either because the argument value is rounded up to an integer multiple of the sleep resolution, or because of the scheduling of other activity by the system. Except for the case of being interrupted by a signal, the suspension time for the relative clock_nanosleep subroutine (that is, with the TIMER_ABSTIME flag not set) shall not be less than the time interval specified by the rqtp argument, as measured by the corresponding clock. The suspension for the absolute clock_nanosleep subroutine (that is, with the TIMER_ABSTIME flag set) is in effect at least until the value of the corresponding clock reaches the absolute time specified by the rqtp argument, except for the case of being interrupted by a signal.

The clock_nanosleep subroutine has no effect on the action or blocking of any signal.

The subroutine fails if the clock_id argument refers to a process or a thread CPU-time clock.

Parameters

clock_id Specifies the clock used to measure the time.
flags Identifies the type of timeout. If TIMER_ABSTIME is set, the time value pointed to by rqtp is an absolute time value; otherwise, it is a time interval.
rmtp Points to the timespec structure used to return the remaining amount of time in an interval (the requested time minus the time actually slept) if the sleep is interrupted.
rqtp Points to the timespec structure that contains requested sleep time.

Return Values
The clock_nanosleep subroutine returns 0 when the requested time has elapsed.

The clock_nanosleep subroutine returns the corresponding error value when it has been interrupted by a signal. For the relative clock_nanosleep subroutine, when the rmtp argument is not NULL, the referenced timespec structure is updated to contain the amount of time remaining in the interval (the requested time minus the time actually slept) if the sleep is interrupted. If the rmtp argument is NULL, the remaining time is not returned. The absolute clock_nanosleep subroutine has no effect on the structure referenced by the rmtp argument.

Error Codes
EINTR The clock_nanosleep subroutine was interrupted by a signal.
EINVAL The `rqtp` parameter specified a nanosecond value less than 0 or greater than or equal to 1000 million; or the `TIMER_ABSTIME` flag was specified in the `flags` parameter and the `rqtp` parameter is outside the range for the clock specified by `clock_id`; or the `clock_id` parameter does not specify a known clock, or specifies the CPU-time clock of the calling thread.

ENOTSUP The `clock_id` argument specifies a clock for which the `clock_nanosleep` subroutine is not supported, such as a CPU-time clock.

ENOTSUP The subroutine is not supported with checkpoint-restarted processes.

Files

`timer.h`

Related Information

“clock_getres, clock_gettime, and clock_settime Subroutine” on page 170, “nanosleep Subroutine” on page 887, “pthread_cond_wait or pthread_cond_timedwait Subroutine” on page 1215, `sleep` subroutine.

The `timer.h` file.


clog, clogf, or clogl Subroutine

Purpose

Computes the complex natural logarithm.

Syntax

```c
#include <complex.h>

double complex clog (double complex z);

e float complex clogf (float complex z);

e long double complex clogl (long double complex z);
```

Description

The `clog`, `clogf`, and `clogl` subroutines compute the complex natural (base e) logarithm of `z`, with a branch cut along the negative real axis.

Parameters

- `z` Specifies the value to be computed.

Return Values

The `clog`, `clogf`, and `clogl` subroutines return the complex natural logarithm value, in the range of a strip mathematically unbounded along the real axis and in the interval `[-i pi, +i pi]` along the imaginary axis.

Related Information

“cexp, cexpf, or cexpl Subroutine” on page 141
close Subroutine

Purpose
Closes a file descriptor.

Syntax
#include <unistd.h>

int close (FileDescriptor);

Description
The close subroutine closes the file or shared memory object associated with the FileDescriptor parameter. If Network File System (NFS) is installed on your system, this file can reside on another node.

All file regions associated with the file specified by the FileDescriptor parameter that this process has previously locked with the lockf or fcntl subroutine are unlocked. This occurs even if the process still has the file open by another file descriptor.

If the FileDescriptor parameter resulted from an open ("open, openx, open64, creat, or creat64 Subroutine" on page 925) subroutine that specified O_DEFER, and this was the last file descriptor, all changes made to the file since the last fsync subroutine are discarded.

If the FileDescriptor parameter is associated with a mapped file, it is unmapped. The shmat subroutine provides more information about mapped files.

The close subroutine attempts to cancel outstanding asynchronous I/O requests on this file descriptor. If the asynchronous I/O requests cannot be canceled, the application is blocked until the requests have completed.

If the FileDescriptor parameter is associated with a shared memory object and the shared memory object remains referenced at the last close (that is, a process has it mapped), the entire contents of the memory object persists until the memory object becomes unreferenced. If this is the last close of a shared memory object and the close results in the memory object becoming unreferenced, and the memory object has been unlinked, the memory object is removed. The shm_open subroutine provides more information about shared memory objects.

The close subroutine is blocked until all subroutines which use the file descriptor return to usr space. For example, when a thread is calling close and another thread is calling select with the same file descriptor, the close subroutine does not return until the select call returns.

When all file descriptors associated with a pipe or FIFO special file have been closed, any data remaining in the pipe or FIFO is discarded. If the link count of the file is 0 when all file descriptors associated with the file have been closed, the space occupied by the file is freed, and the file is no longer accessible.

Note: If the FileDescriptor parameter refers to a device and the close subroutine actually results in a device close, and the device close routine returns an error, the error is returned to the application. However, the FileDescriptor parameter is considered closed and it may not be used in any subsequent calls.

All open file descriptors are closed when a process exits. In addition, file descriptors may be closed during the exec subroutine if the close-on-exec flag has been set for that file descriptor.
**Parameters**

*FileDescriptor*  
Specifies a valid open file descriptor.

**Return Values**

Upon successful completion, a value of 0 is returned. Otherwise, a value of -1 is returned and the `errno` global variable is set to identify the error. If the `close` subroutine is interrupted by a signal that is caught, it returns a value of -1, the `errno` global variable is set to `EINTR` and the state of the `FileDescriptor` parameter is closed.

**Error Codes**

The `close` subroutine is unsuccessful if the following is true:

- **EBADF**  
  The `FileDescriptor` parameter does not specify a valid open file descriptor.
- **EINTR**  
  Specifies that the `close` subroutine was interrupted by a signal.

The `close` subroutine may also be unsuccessful if the file being closed is NFS-mounted and the server is down under the following conditions:

- The file is on a hard mount.
- The file is locked in any manner.

The `close` subroutine may also be unsuccessful if NFS is installed and the following is true:

- **ETIMEDOUT**  
  The connection timed out.

**Related Information**

The `exec` ("exec: execl, execle, execclp, execv, execve, execvp, or execvp Subroutine" on page 235) subroutines, `fcntl` ("fcntl, dup, or dup2 Subroutine" on page 254) subroutine, `ioctl` ("ioctl, ioctlx, ioctl32, or ioctl32x Subroutine" on page 556) subroutine, `lockfx` ("lockfx, lockf, flock, or lockf64 Subroutine" on page 733) subroutine, `open`, `openx`, or `creat` ("open, openx, open64, creat, or creat64 Subroutine" on page 925) subroutine, `pipe` ("pipe Subroutine" on page 1014) subroutine, `socket` subroutine.

The Input and Output Handling in AIX 5L Version 5.3 General Programming Concepts: Writing and Debugging Programs.

---

**compare_and_swap Subroutine**

**Purpose**

Conditionally updates or returns a single word variable atomically.

**Library**

Standard C library (`libc.a`)

**Syntax**

```c
#include <sys/atomic_op.h>

boolean_t compare_and_swap (word_addr, old_val_addr, new_val)
atomic_p word_addr;
int *old_val_addr;
int new_val;
```
The compare_and_swap subroutine performs an atomic operation which compares the contents of a single word variable with a stored old value. If the values are equal, a new value is stored in the single word variable and TRUE is returned; otherwise, the old value is set to the current value of the single word variable and FALSE is returned.

The compare_and_swap subroutine is useful when a word value must be updated only if it has not been changed since it was last read.

Note: The word containing the single word variable must be aligned on a full word boundary.

Note: If compare_and_swap is used as a locking primitive, insert an isync at the start of any critical sections.

Parameters

word_addr Specifies the address of the single word variable.
old_val_addr Specifies the address of the old value to be checked against (and conditionally updated with) the value of the single word variable.
new_val Specifies the new value to be conditionally assigned to the single word variable.

Return Values

TRUE Indicates that the single word variable was equal to the old value, and has been set to the new value.
FALSE Indicates that the single word variable was not equal to the old value, and that its current value has been returned in the location where the old value was previously stored.

Related Information

The fetch_and_add subroutine, fetch_and_and subroutine, fetch_and_or subroutine.

c ompile, step, or advance Subroutine

Purpose

Compiles and matches regular-expression patterns.

Note: Commands use the regcomp, regexec, regfree, and regerror subroutines for the functions described in this article.

Library

Standard C Library (libc.a)

Syntax

#define INIT declarations
#define GETC( ) getc_code
#define PEEK( ) peekc_code
#define UNGETC(c) ungetc_code
#define RETURN(pointer) return_code
#define ERROR(val) error_code
#include <regexp.h>
#include <NLregexp.h>

char *compile (InString, ExpBuffer, EndBuffer, EndOfFile)
char * ExpBuffer;
char * InString; * ExpBuffer;
int EndOfFile;

int step (String, ExpBuffer)
const char * String, * ExpBuffer;
int advance (String, ExpBuffer)
const char * String, * ExpBuffer;

Description
The /usr/include/regexp.h file contains subroutines that perform regular-expression pattern matching. Programs that perform regular-expression pattern matching use this source file. Thus, only the regexp.h file needs to be changed to maintain regular expression compatibility between programs.

The interface to this file is complex. Programs that include this file define the following six macros before the #include <regexp.h> statement. These macros are used by the compile subroutine:

INIT
This macro is used for dependent declarations and initializations. It is placed right after the declaration and opening { (left brace) of the compile subroutine. The definition of the INIT buffer must end with a ; (semicolon). INIT is frequently used to set a register variable to point to the beginning of the regular expression so that this register variable can be used in the declarations for the GETC, PEEKC, and UNGETC macros. Otherwise, you can use INIT to declare external variables that GETC, PEEKC, and UNGETC require.

GETC()
This macro returns the value of the next character in the regular expression pattern. Successive calls to the GETC macro should return successive characters of the pattern.

PEEK()
This macro returns the next character in the regular expression. Successive calls to the PEEK macro should return the same character, which should also be the next character returned by the GETC macro.

UNGETC(c)
This macro causes the parameter c to be returned by the next call to the GETC and PEEKC macros. No more than one character of pushback is ever needed, and this character is guaranteed to be the last character read by the GETC macro. The return value of the UNGETC macro is always ignored.

RETURN(pointer)
This macro is used for normal exit of the compile subroutine. The pointer parameter points to the first character immediately following the compiled regular expression. This is useful for programs that have memory allocation to manage.
This macro is used for abnormal exit from the compile subroutine. It should never contain a return statement. The val parameter is an error number. The error values and their meanings are:

<table>
<thead>
<tr>
<th>Error</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>11</td>
<td>Interval end point too large</td>
</tr>
<tr>
<td>16</td>
<td>Bad number</td>
</tr>
<tr>
<td>25</td>
<td>\ digit out of range</td>
</tr>
<tr>
<td>36</td>
<td>Illegal or missing delimiter</td>
</tr>
<tr>
<td>41</td>
<td>No remembered search String</td>
</tr>
<tr>
<td>42</td>
<td>\ (?) imbalance</td>
</tr>
<tr>
<td>43</td>
<td>Too many (</td>
</tr>
<tr>
<td>44</td>
<td>More than two numbers given in ( )</td>
</tr>
<tr>
<td>45</td>
<td>) expected after \</td>
</tr>
<tr>
<td>46</td>
<td>First number exceeds second in ( )</td>
</tr>
<tr>
<td>49</td>
<td>[ ] imbalance</td>
</tr>
<tr>
<td>50</td>
<td>Regular expression overflow</td>
</tr>
<tr>
<td>70</td>
<td>Invalid endpoint in range</td>
</tr>
</tbody>
</table>

The compile subroutine compiles the regular expression for later use. The InString parameter is never used explicitly by the compile subroutine, but you can use it in your macros. For example, you can use the compile subroutine to pass the string containing the pattern as the InString parameter to compile and use the INIT macro to set a pointer to the beginning of this string. The example in the “Examples” on page 180 section uses this technique. If your macros do not use InString, then call compile with a value of ((char *) 0) for this parameter.

The ExpBuffer parameter points to a character array where the compiled regular expression is to be placed. The EndBuffer parameter points to the location that immediately follows the character array where the compiled regular expression is to be placed. If the compiled expression cannot fit in (EndBuffer-ExpBuffer) bytes, the call ERROR(50) is made.

The EndOfFile parameter is the character that marks the end of the regular expression. For example, in the ed command, this character is usually / (slash).

The regexp.h file defines other subroutines that perform actual regular-expression pattern matching. One of these is the step subroutine.

The String parameter of the step subroutine is a pointer to a null-terminated string of characters to be checked for a match.

The ExpBuffer parameter points to the compiled regular expression, obtained by a call to the compile subroutine.

The step subroutine returns the value 1 if the given string matches the pattern, and 0 if it does not match. If it matches, then step also sets two global character pointers: loc1, which points to the first character that matches the pattern, and loc2, which points to the character immediately following the last character that matches the pattern. Thus, if the regular expression matches the entire string, loc1 points to the first character of the String parameter and loc2 points to the null character at the end of the String parameter.
The step subroutine uses the global variable circf, which is set by the compile subroutine if the regular expression begins with a ^ (circumflex). If this variable is set, step only tries to match the regular expression to the beginning of the string. If you compile more than one regular expression before executing the first one, save the value of circf for each compiled expression and set circf to that saved value before each call to step.

Using the same parameters that were passed to it, the step subroutine calls a subroutine named advance. The step function increments through the String parameter and calls the advance subroutine until it returns a 1, indicating a match, or until the end of String is reached. To constrain the String parameter to the beginning of the string in all cases, call the advance subroutine directly instead of calling the step subroutine.

When the advance subroutine encounters an * (asterisk) or a \{ \} sequence in the regular expression, it advances its pointer to the string to be matched as far as possible and recursively calls itself, trying to match the rest of the string to the rest of the regular expression. As long as there is no match, the advance subroutine backs up along the string until it finds a match or reaches the point in the string that initially matched the * or \{ \}. You can stop this backing-up before the initial point in the string is reached. If the locs global character is equal to the point in the string sometime during the backing-up process, the advance subroutine breaks out of the loop that backs up and returns 0. This is used for global substitutions on the whole line so that expressions such as s/y*/g do not loop forever.

Note: In 64-bit mode, these interfaces are not supported: they fail with a return code of 0. In order to use the 64-bit version of this functionality, applications should migrate to the fnmatch, glob, regcomp, and regexec functions which provide full internationalized regular expression functionality compatible with ISO 9945-1:1996 (IEEE POSIX 1003.1) and with the UNIX98 specification.

Parameters

InString Specifies the string containing the pattern to be compiled. The InString parameter is not used explicitly by the compile subroutine, but it may be used in macros.

ExpBuffer Points to a character array where the compiled regular expression is to be placed.

EndBuffer Points to the location that immediately follows the character array where the compiled regular expression is to be placed.

EndOfFile Specifies the character that marks the end of the regular expression.

String Points to a null-terminated string of characters to be checked for a match.

Examples

The following is an example of the regular expression macros and calls:

#include <regexp.h>
... compile (patstr, expbuf, &expbuf[ESIZE], '\0');
      if (step (linebuf, expbuf))
        succeed();
      ...

Related Information

The regcmp or regex subroutine, regcomp subroutine, regerror subroutine, regexec subroutine, regfree subroutine.
confstr Subroutine

**Purpose**

Gets configurable variables.

**Library**

Standard C library (`libc.a`)

**Syntax**

```c
#include <unistd.h>

size_t confstr (int name, char * buf, size_t len);
```

**Description**

The `confstr` subroutine determines the current setting of certain system parameters, limits, or options that are defined by a string value. It is mainly used by applications to find the system default value for the `PATH` environment variable. Its use and purpose are similar to those of the `sysconf` subroutine, but it returns string values rather than numeric values.

If the `Len` parameter is not 0 and the `Name` parameter has a system-defined value, the `confstr` subroutine copies that value into a `Len`-byte buffer pointed to by the `Buf` parameter. If the string returns a value longer than the value specified by the `Len` parameter, including the terminating null byte, then the `confstr` subroutine truncates the string to `Len`-1 bytes and adds a terminating null byte to the result. The application can detect that the string was truncated by comparing the value returned by the `confstr` subroutine with the value specified by the `Len` parameter.

**Parameters**

- **Name**
  Specifies the system variable setting to be returned. Valid values for the `Name` parameter are defined in the `unistd.h` file.

- **Buf**
  Points to the buffer into which the `confstr` subroutine copies the value of the `Name` parameter.

- **Len**
  Specifies the size of the buffer storing the value of the `Name` parameter.

**Return Values**

If the value specified by the `Name` parameter is system-defined, the `confstr` subroutine returns the size of the buffer needed to hold the entire value. If this return value is greater than the value specified by the `Len` parameter, the string returned as the `Buf` parameter is truncated.

If the value of the `Len` parameter is set to 0 and the `Buf` parameter is a null value, the `confstr` subroutine returns the size of the buffer needed to hold the entire system-defined value, but does not copy the string value. If the value of the `Len` parameter is set to 0 but the `Buf` parameter is not a null value, the result is unspecified.
Error Codes
The **confstr** subroutine will fail if:

**EINVAL** The value of the name argument is invalid.

Example
To find out what size buffer is needed to store the string value of the Name parameter, enter:

```c
confstr(_CS_PATH, NULL, (size_t) 0)
```

The **confstr** subroutine returns the size of the buffer.

Files
- `/usr/include/limits.h` Contains system-defined limits.
- `/usr/include/unistd.h` Contains system-defined environment variables.

Related Information
- The **pathconf** subroutine.
- The **sysconf** subroutine.

Subroutines, Example Programs, and Libraries in *AIX 5L Version 5.3 General Programming Concepts: Writing and Debugging Programs*:

**conj, conjf, or conjl Subroutine**

**Purpose**
Computes the complex conjugate.

**Syntax**
```c
#include <complex.h>

double complex conj (z);
double complex z;

float complex conjf (z);
float complex z;

long double complex conjl (z);
long double complex z;
```

**Description**
The **conj**, **conjf**, or **conjl** subroutines compute the complex conjugate of z, by reversing the sign of its imaginary part.

**Parameters**
- **z** Specifies the value to be computed.

**Return Values**
The **conj**, **conjf**, or **conjl** subroutines return the complex conjugate value.
Related Information
The "carg, cargf, or cargl Subroutine" on page 131, "cimag, cimagf, or cimagl Subroutine" on page 163, "cproj, cprojf, or cprojl Subroutine" on page 189, "creal, crealf, or creall Subroutine" on page 190.

conv Subroutines

Purpose
Translates characters.

Library
Standard C Library (libc.a)

Syntax
#include <ctype.h>

int toupper (Character)
int Character;

int tolower (Character)
int Character;

int _toupper (Character)
int Character;

int _tolower (Character)
int Character;

int toascii (Character)
int Character;

int NCesc (Pointer, CharacterPointer)
NLchar *Pointer;
char *CharacterPointer;

int NCtoupper (Xcharacter)
int Xcharacter;

int NCltolower (Xcharacter)
int Xcharacter;

int _NCtoupper (Xcharacter)
int Xcharacter;

int _NCtolower (Xcharacter)
int Xcharacter;

int NCToNLchar (Xcharacter)
int Xcharacter;

int NCunes (CharacterPointer, Pointer)
char *CharacterPointer;
NLchar *Pointer;
int NCflatchr (Xcharacter)
int Xcharacter;

**Description**

The `toupper` and the `tolower` subroutines have as domain an `int`, which is representable as an unsigned `char` or the value of `EOF`: -1 through 255.

If the parameter of the `toupper` subroutine represents a lowercase letter and there is a corresponding uppercase letter (as defined by `LC_CTYPE`), the result is the corresponding uppercase letter. If the parameter of the `tolower` subroutine represents an uppercase letter, and there is a corresponding lowercase letter (as defined by `LC_CTYPE`), the result is the corresponding lowercase letter. All other values in the domain are returned unchanged. If case-conversion information is not defined in the current locale, these subroutines determine character case according to the "C" locale.

The `_toupper` and `_tolower` subroutines accomplish the same thing as the `toupper` and `tolower` subroutines, but they have restricted domains. The `_toupper` routine requires a lowercase letter as its parameter; its result is the corresponding uppercase letter. The `_tolower` routine requires an uppercase letter as its parameter; its result is the corresponding lowercase letter. Values outside the domain cause undefined results.

The `NCxxxxxx` subroutines translate all characters, including extended characters, as code points. The other subroutines translate traditional ASCII characters only. The `NCxxxxxx` subroutines are obsolete and should not be used if portability and future compatibility are a concern.

The value of the `Xcharacter` parameter is in the domain of any legal `NLchar` data type. It can also have a special value of -1, which represents the end of file (EOF).

If the parameter of the `NCtoupper` subroutine represents a lowercase letter according to the current collating sequence configuration, the result is the corresponding uppercase letter. If the parameter of the `NCtolower` subroutine represents an uppercase letter according to the current collating sequence configuration, the result is the corresponding lowercase letter. All other values in the domain are returned unchanged.

The `_NCtoupper` and `_NCtolower` routines are macros that perform the same function as the `NCtoupper` and `NCtolower` subroutines, but have restricted domains and are faster. The `_NCtoupper` macro requires a lowercase letter as its parameter; its result is the corresponding uppercase letter. The `_NCtolower` macro requires an uppercase letter as its parameter; its result is the corresponding lowercase letter. Values outside the domain cause undefined results.

The `NCtoNLchar` subroutine yields the value of its parameter with all bits turned off that are not part of an `NLchar` data type.

The `NCesc` subroutine converts the `NLchar` value of the `Pointer` parameter into one or more ASCII bytes stored in the character array pointed to by the `CharacterPointer` parameter. If the `NLchar` data type represents an extended character, it is converted into a printable ASCII escape sequence that uniquely identifies the extended character. `NCesc` returns the number of bytes it wrote. The display symbol table lists the escape sequence for each character.

The opposite conversion is performed by the `NCunesc` macro, which translates an ordinary ASCII byte or escape sequence starting at `CharacterPointer` into a single `NLchar` at `Pointer`. `NCunesc` returns the number of bytes it read.
The **NClatchr** subroutine converts its parameter value into the single ASCII byte that most closely resembles the parameter character in appearance. If no ASCII equivalent exists, it converts the parameter value to a ? (question mark).

**Note:** The **setlocale** subroutine may affect the conversion of the decimal point symbol and the thousands separator.

### Parameters

- **Character**
  - Specifies the character to be converted.
- **Xcharacter**
  - Specifies an **NLchar** value to be converted.
- **CharacterPointer**
  - Specifies a pointer to a single-byte character array.
- **Pointer**
  - Specifies a pointer to an escape sequence.

### Related Information

The Japanese **conv** ("Japanese conv Subroutines" on page 571) subroutines, **ctype** ("ctype, isalpha, isupper, islower, isdigit, isxdigit, isalnum, isspace, ispunct, isprint, isgraph, iscntrl, or isascii Subroutines" on page 208) subroutines, **getc**, **fgetc**, **getchar**, or **getw** ("getc, getchar, fgetc, or getw Subroutine" on page 343) subroutine, **getwc**, **fgetwc**, or **getwchar** ("getwc, fgetwc, or getwchar Subroutine" on page 472) subroutine, **setlocale** subroutine.

List of Character Manipulation Services and Subroutines, Example Programs, and Libraries in **AIX 5L Version 5.3 General Programming Concepts: Writing and Debugging Programs**.

**National Language Support Overview** in **AIX 5L Version 5.3 National Language Support Guide and Reference**.

---

**copysign, copysignf, or copysignl Subroutine**

### Purpose

Performs number manipulation.

### Syntax

```c
#include <math.h>

double copysign (x, y);

double x, double y;

float copysignf (x, y);

float x, float y;

long double copysignl (x, y);

long double x, long double y;
```

### Description

The **copysign**, **copysignf**, and **copysignl** subroutines produce a value with the magnitude of \( x \) and the sign of \( y \).

### Parameters

- \( x \)  
  - Specifies the magnitude.
- \( y \)  
  - Specifies the sign.
Return Values
Upon successful completion, the `copysign`, `copysignf` and `copysignl` subroutines return a value with a magnitude of \( x \) and a sign of \( y \).

Related Information
- `math.h` in AIX 5L Version 5.3 Files Reference.

coredump Subroutine

Purpose
Creates a core file without terminating the calling process.

Library
Standard C library (libc.a)

Syntax
```c
#include <core.h>

int coredump(
    struct coredumpinfo *coredumpinfop);
```

Description
The `coredump` subroutine creates a core file of the calling process without terminating the calling process. The created core file contains the memory image of the process, and this can be used with the `dbx` command for debugging purposes. In multithreaded processes, only one thread at a time should attempt to call this subroutine. Subsequent calls to `coredump` while a core dump (initiated by another thread) is in progress will fail.

Applications expected to use this facility need to be built with the `-bM:UR` binder flag, otherwise the routine will fail with an error code of `ENOTSUP`.

The `coredumpinfo` structure has the following fields:

<table>
<thead>
<tr>
<th>Member Type</th>
<th>Member Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>unsigned int</td>
<td>length</td>
<td>Length of the core file name</td>
</tr>
<tr>
<td>char *</td>
<td>name</td>
<td>Points to a character string that contains the name of the core file</td>
</tr>
<tr>
<td>int</td>
<td>reserved[8]</td>
<td>Reserved fields for future use</td>
</tr>
</tbody>
</table>

Parameters
- `coredumpinfop` Points to the `coredumpinfo` structure

If a NULL pointer is passed as an argument, the default file named core in the current directory is used.

Return Values
Upon successful completion, the `coredump` subroutine returns a value of 0. If the `coredump` subroutine is not successful, a value of -1 is returned and the `errno` global variable is set to indicate the error.
Error Codes

EINVAL
Invalid argument.

EACCESS
Search permission is denied on a component of the path prefix, the file exists and the
pwrite permission is denied, or the file does not exist and write permission is
denied for the parent directory of the file to be created.

EINVAL
A core dump is already in progress.

ENOMEM
Not enough memory.

EINPROGRESS
Routine not supported.

EFAULT
Invalid user address.

Related Information
The adb command, dbx command.

The core file format.

cosf, cosl, or cos Subroutine

Purpose
Computes the cosine.

Syntax
#include <math.h>

float cosf (x)
float x;

long double cosl (x)
long double x;

double cos (x)
double x;

Description
The cosf, cosl, and cos subroutines compute the cosine of the x, parameter (measured in radians).

An application wishing to check for error situations should set errno to zero and call
feclearexcept(FE_ALL_EXCEPT) before calling these subroutines. Upon return, if errno is nonzero or
fetestexcept(FE_INVALID | FE_DIVBYZERO | FE_OVERFLOW | FE_UNDERFLOW) is nonzero, an error
has occurred.

Parameters

x
Specifies the value to be computed.

Return Values
Upon successful completion, the cosf, cosl, and cos subroutines return the cosine of x.

If x is NaN, a NaN is returned.

If x is ±0, the value 1.0 is returned.

If x is ±Inf, a domain error occurs, and a NaN is returned.
cosh, coshf, or coshl Subroutine

Purpose
Computes the hyperbolic cosine.

Syntax

```c
#include <math.h>

float coshf (float x);

long double coshl (long double x);

double cosh (double x);
```

Description
The `coshf`, `coshl`, and `cosh` subroutines compute the hyperbolic cosine of the `x` parameter.

An application wishing to check for error situations should set `errno` to zero and call `feclearexcept(FE_ALL_EXCEPT)` before calling these functions. On return, if `errno` is nonzero or `fetestexcept(FE_INVALID | FE_DIVBYZERO | FE_OVERFLOW | FE_UNDERFLOW)` is nonzero, an error has occurred.

Parameters

- `x` Specifies the value to be computed.

Return Values
Upon successful completion, the `coshf`, `coshl`, and `cosh` subroutines return the hyperbolic cosine of `x`.

If the correct value would cause overflow, a range error occurs and the `coshf`, `coshl`, and `cosh` subroutines return the value of the macro `HUGE_VALF`, `HUGE_VALL`, and `HUGE_VAL`, respectively.

If `x` is NaN, a NaN is returned.

If `x` is ±0, the value 1.0 is returned.

If `x` is ±Inf, +Inf is returned.

Related Information
"acosh, acoshf, or acoshl Subroutine" on page 30, "feclearexcept Subroutine" on page 262, "fetestexcept Subroutine" on page 270, and "class, _class, finite, isnan, or unordered Subroutines" on page 167.
**cpow, cpowf, or cpowl Subroutine**

**Purpose**
Computes the complex power.

**Syntax**
```c
#include <complex.h>

double complex cpow (x, y)
double complex x;
double complex y;

float complex cpowf (x, y)
float complex x;
float complex y;

long double complex cpowl (x, y)
long double complex x;
long double complex y;
```

**Description**
The `cpow`, `cpowf`, and `cpowl` subroutines compute the complex power function $x^y$, with a branch cut for the first parameter along the negative real axis.

**Parameters**
- $x$ Specifies the base value.
- $y$ Specifies the power the base value is raised to.

**Return Values**
The `cpow`, `cpowf`, and `cpowl` subroutines return the complex power function value.

**Related Information**
“cabs, cabsf, or cabsl Subroutine” on page 129 and “csqrt, csqrtf, or csqrtl Subroutine” on page 194

**cproj, cprojf, or cprojl Subroutine**

**Purpose**
Computes the complex projection functions.

**Syntax**
```c
#include <complex.h>

double complex cproj (z)
double complex z;
```
float complex cprojf(z)
float complex z;

long double complex cprojl(z)
long double complex z;

Description
The cproj, cprojf, and cprojl subroutines compute a projection of z onto the Riemann sphere: z projects to z, except that all complex infinities (even those with one infinite part and one NaN part) project to positive infinity on the real axis. If z has an infinite part, cproj(z) shall be equivalent to:
INFINITY + I * copysign(0.0, cimag(z))

Parameters
z Specifies the value to be projected.

Return Values
The cproj, cprojf, and cprojl subroutines return the value of the projection onto the Riemann sphere.

Related Information
"carg, cargf, or cargl Subroutine" on page 131, "cimag, cimagf, or cimagl Subroutine" on page 163, "conj, conjf, or conjl Subroutine" on page 182, and "creal, crealf, or creall Subroutine."

creal, crealf, or creall Subroutine

Purpose
Computes the real part of a specified value.

Syntax
#include <complex.h>

double creal (z)
double complex z;

float crealf (z)
float complex z;

long double creall (z)
long double complex z;

Description
The creal, crealf, and creall subroutines compute the real part of the value specified by the z parameter.

Parameters
z Specifies the real to be computed.

Return Values
These subroutines return the real part value.
crypt, encrypt, or setkey Subroutine

Purpose
Encrypts or decrypts data.

Library
Standard C Library (libc.a)

Syntax
char *crypt (PW, Salt)
const char * PW, * Salt;

void encrypt (Block, EdFlag)
char Block[64];
int EdFlag;

void setkey (Key)
const char * Key;

Description
The crypt and encrypt subroutines encrypt or decrypt data. The crypt subroutine performs a one-way encryption of a fixed data array with the supplied PW parameter. The subroutine uses the Salt parameter to vary the encryption algorithm.

The encrypt subroutine encrypts or decrypts the data supplied in the Block parameter using the key supplied by an earlier call to the setkey subroutine. The data in the Block parameter on input must be an array of 64 characters. Each character must be an char 0 or char 1.

If you need to statically bind functions from libc.a for crypt do the following:
1. Create a file and add the following:
   #!
   ___setkey
   ___encrypt
   ___crypt
2. Perform the linking.
3. Add the following to the make file:
   -bi:YourFileName

   where YourFileName is the name of the file you created in step 1. It should look like the following:
   LDFLAGS=bnoautoimp -bi:/lib/syscalls.exp -bi:YourFileName -lc

These subroutines are provided for compatibility with UNIX® system implementations.

Parameters
Block Identifies a 64-character array containing the values (char) 0 and (char) 1. Upon return, this buffer contains the encrypted or decrypted data.
**EdFlag**

Determines whether the subroutine encrypts or decrypts the data. If this parameter is 0, the data is encrypted. If this is a nonzero value, the data is decrypted. If the /usr/lib/libdes.a file does not exist and the EdFlag parameter is set to nonzero, the encrypt subroutine returns the **ENOSYS** error code.

**Key**

Specifies an 64-element array of 0’s and 1’s cast as a **const char** data type. The Key parameter is used to encrypt or decrypt data.

**PW**

Specifies up to an 8-character string to be encrypted.

**Salt**

Specifies a 2-character string chosen from the following:

- **A-Z**  Uppercase alpha characters
- **a-z**  Lowercase alpha characters
- **0-9**  Numeric characters
- **.**  Period
- **/**  Slash

The Salt parameter is used to vary the hashing algorithm in one of 4096 different ways.

**Return Values**

The crypt subroutine returns a pointer to the encrypted password. The static area this pointer indicates may be overwritten by subsequent calls.

**Error Codes**

The encrypt subroutine returns the following:

**ENOSYS**

The encrypt subroutine was called with the EdFlag parameter which was set to a nonzero value. Also, the /usr/lib/libdes.a file does not exist.

**Related Information**

The newpass ("newpass Subroutine" on page 891) subroutine.

The login command, passwd command, su command.

List of Security and Auditing Subroutines and Subroutines Overview in AIX 5L Version 5.3 General Programming Concepts: Writing and Debugging Programs.

---

**csid Subroutine**

**Purpose**

Returns the character set ID (charsetID) of a multibyte character.

**Library**

Standard C Library (libc.a)

**Syntax**

```c
#include <stdlib.h>

int csid (const char *String);
```
Description

The csid subroutine returns the charsetID of the multibyte character pointed to by the String parameter. No validation of the character is performed. The parameter must point to a value in the character range of the current code set defined in the current locale.

Parameters

String Specifies the character to be tested.

Return Values

Successful completion returns an integer value representing the charsetID of the character. This integer can be a number from 0 through \( n \), where \( n \) is the maximum character set defined in the CHARSETID field of the charmap. See "Understanding the Character Set Description (charmap) Source File" in Operating system and device management for more information.

Related Information

The mbstowcs subroutine, wcsid subroutine.


Subroutines, Example Programs, and Libraries in AIX 5L Version 5.3 General Programming Concepts: Writing and Debugging Programs.

---

csin, csinf, or csinl Subroutine

Purpose

Computes the complex sine.

Syntax

```c
#include <complex.h>

double complex csin (z)
double complex z;

float complex csinf (z)
float complex z;

long double complex csinl (z)
long double complex z;
```

Description

The csin, csinf, and csinl subroutines compute the complex sine of the value specified by the z parameter.

Parameters

\( z \) Specifies the value to be computed.

Return Values

The csin, csinf, and csinl subroutines return the complex sine value.
csinh, csinhf, or csinhl Subroutine

Purpose
Computes the complex hyperbolic sine.

Syntax
#include <complex.h>

double complex csinh (z)
double complex z;

float complex csinhf (z)
float complex z;

long double complex csinhl (z)
long double complex z;

Description
The csinh, csinhf, and csinhl subroutines compute the complex hyperbolic sine of the value specified by the z parameter.

Parameters
z Specifies the value to be computed.

Return Values
The csinh, csinhf, and csinhl subroutines return the complex hyperbolic sine value.

Related Information
"casinh, casinfh, or casinlh Subroutine" on page 132

csqrt, csqrtf, or csqrtl Subroutine

Purpose
Computes complex square roots.

Syntax
#include <complex.h>

double complex csqrt (z)
double complex z;

float complex csqrtf (z)
float complex z;

long double complex csqrtl (z)
long double complex z;
Description
The csqrt, csqrtf, and csqrtl subroutines compute the complex square root of the value specified by the z parameter, with a branch cut along the negative real axis.

Parameters

z Specifies the value to be computed.

Return Values
The csqrt, csqrtf, and csqrtl subroutines return the complex square root value, in the range of the right half-plane (including the imaginary axis).

Related Information
“cabs, cabsf, or cabsl Subroutine” on page 129, “cpow, cpowf, or cpowl Subroutine” on page 189

CT_HOOKx and CT_GEN macros

Purpose
Record a trace event into Component Trace, LMT or system trace buffers.

Syntax
The following set of macros is provided to record a trace entry:

```
#include <sys/ras_trace.h>

CT_HOOK0(ras_block_t cb, int level, int mem_dest, long hkwd);
CT_HOOK1(ras_block_t cb, int level, int mem_dest, long hkwd, long d1);
CT_HOOK2(ras_block_t cb, int level, int mem_dest, long hkwd, long d1, long d2);
CT_HOOK3(ras_block_t cb, int level, int mem_dest, long hkwd, long d1, long d2, long d3);
CT_HOOK4(ras_block_t cb, int level, int mem_dest, long hkwd, long d1, long d2, long d3, long d4);
CT_HOOK5(ras_block_t cb, int level, int mem_dest, long hkwd, long d1, long d2, long d3, long d4, long d5);
CT_GEN (ras_block_t cb, int level, long hkwd, long data, long len, void *buf);
```

Description
The CT_HOOKx macros allow you to record a trace hook. The "x" is the number of data words you want in this trace event.

The CT_GEN macro is used to record a generic trace hook.

All traces are timestamped.

Parameters

ras_block_t cb The cb parameter is the RAS control block that refers to the component that this trace entry belongs to.
int level

The *level* parameter allows filtering of different trace entries. The higher this level is, the more this trace will be considered as debug or detail information. In other words, this trace entry will appear only if the level of the trace entry is less than or equal to the level of trace chosen for memory or system trace mode.

Ten levels of trace are available (CT_LEVEL_0 to CT_LEVEL_9, corresponding to value 0 to 9) with four special levels:

- minimal (CT_LVL_MINIMAL (=CT_LEVEL_1))
- normal (CT_LVL_NORMAL (=CT_LEVEL_3))
- detail (CT_LVL_DETAIL (=CT_LEVEL_7))
- default (CT_LVL_DEFAULT (= CT_LVL_MINIMAL (in 5.3 ML 5))

When you are porting an existing driver or subsystem from the existing system trace to component trace, trace existing entries at CT_LVL_DEFAULT.

int mem_dest

For CT_HOOKx macros, the *mem_dest* parameter indicates the memory destination for this trace entry. It is an ORed value with the following possible settings:

- MT_RARE: the trace entry is saved in the rare buffer of lightweight memory trace if the level condition of the memory trace mode for this control block is satisfied, meaning that the current level of trace for the memory trace mode is greater than or equal to the level of this trace entry.
- MT_COMMON: the trace entry is saved in the common buffer of the lightweight memory trace if the level condition of the memory trace mode for this control block is satisfied.
- MT_PRIV: the trace entry is saved in the private memory buffer of the component if the level condition of the memory trace mode for this control block is satisfied.
- MT_SYSTEM: the trace entry is saved in the existing system trace if the level condition of the system trace mode for this control block is satisfied, if the system trace is running, and if the hook meets any additional criteria specified as part of the system trace. For example, if MT_SYSTEM is not set, the trace entry is not saved in the existing system trace.

Only one of the MT_RARE, MT_COMMON and MT_PRIV values should be used, but you can combine ORed with MT_SYSTEM. Otherwise, the trace entry will be duplicated in several memory buffers.

The *mem_dest* parameter is not needed for the CT_GEN macro because lightweight memory trace cannot accommodate generic entries. CT_GEN checks the memory trace and system trace levels to determine whether the generic entry should enter the private memory buffer and system trace buffers respectively.

The *hkwd*, *d1*, *d2*, *d3*, *d4*, *d5*, *len* and *buf* parameters are the same as those used for the existing TRCHKx or TRCGEN macros. The TRCHKx refers to the TRCHKL*n*T macros where *n* is from 0 to 5. For example, TRCHKL1T (hkwd, d1). The TRCGEN macros refer to the TRCGEN and TRCGENT macros.

For the hookword, OR the hookID with a subhookID if needed. For the CT_HOOKx macro, the subhook is ORed into the hookword. For the CT_GEN macro, the subhook is the *d1* parameter.

Related Information

- **[Trace Facility]** in *AIX 5L Version 5.3 Kernel Extensions and Device Support Programming Concepts*.

- The trcgenk and trcgenkt kernel services.

- The trhook, trhook64, utrhook and utrhook64 subroutine.

- The ras_register and ras_unregister exported kernel services.
CT_TRCON macro

Purpose
Return information on whether any trace is active at a certain level for a component.

Syntax
#include <sys/ras_trace.h>
CT_TRCON(cb, level)

Description
The CT_TRCON macro allows you to ascertain whether any type of trace (Component Trace, lightweight memory trace or system trace) will record events for the component specified at the trace detail level specified.

Parameters
ras_block_t cb
The cb parameter is the RAS control block pointer that refers to the component that this trace entry belongs to.
int level
Specifies the trace detail level.

Related Information
Component Trace Facility in AIX 5L Version 5.3 Kernel Extensions and Device Support Programming Concepts.
The “CT_HOOKx and CT_GEN macros” on page 195.
The ras_register/ras_unregister exported kernel services.
The ras_control exported kernel services.

ctan, ctanf, or ctanl Subroutine

Purpose
Computes complex tangents.

Syntax
#include <complex.h>

double complex ctan (z)
double complex z;

float complex ctanf (z)
float complex z;

long double complex ctanl (z)
long double complex z;

Description
The ctan, ctanf, and ctanl subroutines compute the complex tangent of the value specified by the z parameter.
Parameters

z  Specifies the value to be computed.

Return Values
The `ctanh`, `ctanhf`, and `ctanhl` subroutines return the complex hyperbolic tangent value.

Related Information
“ctanh, ctanhf, or ctanhl Subroutine” on page 133

```
#include <complex.h>

double complex ctanh (z)
double complex z;

float complex ctanhf (z)
float complex z;

long double complex ctanhl (z)
long double complex z;
```

Description
The `ctanh`, `ctanhf`, and `ctanhl` subroutines compute the complex hyperbolic tangent of z.

Parameters

z  Specifies the value to be computed.

Return Values
The `ctanh`, `ctanhf`, and `ctanhl` subroutines return the complex hyperbolic tangent value.

Related Information
“ctanh, ctanhf, or ctanhl Subroutine” on page 133

ctermd Subroutine

Purpose
Generates the path name of the controlling terminal.

Library
Standard C Library (`libc.a`)
Syntax

```c
#include <stdio.h>
char *ctermid (String)
char *String;
```

Description

The `ctermid` subroutine generates the path name of the controlling terminal for the current process and stores it in a string.

**Note:** File access permissions depend on user access. Access to a file whose path name the `ctermid` subroutine has returned is not guaranteed.

The difference between the `ctermid` and `ttyname` subroutines is that the `ttyname` subroutine must be handed a file descriptor and returns the actual name of the terminal associated with that file descriptor. The `ctermid` subroutine returns a string (the `/dev/tty` file) that refers to the terminal if used as a file name. Thus, the `ttyname` subroutine is useful only if the process already has at least one file open to a terminal.

Parameters

- **String**
  - If the `String` parameter is a null pointer, the string is stored in an internal static area and the address is returned. The next call to the `ctermid` subroutine overwrites the contents of the internal static area.
  - If the `String` parameter is not a null pointer, it points to a character array of at least `L_ctermid` elements as defined in the `stdio.h` file. The path name is placed in this array and the value of the `String` parameter is returned.

Related Information

The `isatty` or `ttyname` subroutine.

Input and Output Handling Programmer's Overview in AIX 5L Version 5.3 General Programming Concepts: Writing and Debugging Programs.

### ctime, localtime, gmtime, mktime, difftime, asctime, or tzset Subroutine

**Purpose**

Converts the formats of date and time representations.

**Library**

Standard C Library (`libc.a`)

**Syntax**

```c
#include <time.h>
char *ctime (Clock)
const time_t *Clock;
struct tm *localtime (Clock)
const time_t *Clock;
struct tm *gmtime (Clock)
const time_t *Clock;
```
time_t mktime(Timeptr)
struct tm *Timeptr;

double difftime(Time1, Time0)
time_t Time0, Time1;

char *asctime(Tm)
const struct tm *Tm;

void tzset()
extern long int timezone;
extern int daylight;
extern char *tzname[];

Description

Attention: Do not use the tzset subroutine when linking with both libc.a and libbsd.a. The tzset subroutine sets the global external variable called timezone, which conflicts with the timezone subroutine in libbsd.a. This name collision may cause unpredictable results.

Attention: Do not use the ctime, localtime, gmtime, or asctime subroutine in a multithreaded environment. See the multithread alternatives in the ctime_r subroutine article.

The ctime subroutine converts a time value pointed to by the Clock parameter, which represents the time in seconds since 00:00:00 Coordinated Universal Time (UTC), January 1, 1970, into a 26-character string in the following form:
Sun Sept 16 01:03:52 1973

The width of each field is always the same as shown here.

The ctime subroutine adjusts for the time zone and daylight saving time, if it is in effect.

The localtime subroutine converts the long integer pointed to by the Clock parameter, which contains the time in seconds since 00:00:00 UTC, 1 January 1970, into a tm structure. The localtime subroutine adjusts for the time zone and for daylight-saving time, if it is in effect. Use the time-zone information as though localtime called tzset.

The gmtime subroutine converts the long integer pointed to by the Clock parameter into a tm structure containing the Coordinated Universal Time (UTC), which is the time standard the operating system uses.

Note: UTC is the international time standard intended to replace GMT.

The tm structure is defined in the time.h file, and it contains the following members:

int tm_sec;       /* Seconds (0 - 59) */
int tm_min;       /* Minutes (0 - 59) */
int tm_hour;      /* Hours (0 - 23) */
int tm_mday;      /* Day of month (1 - 31) */
int tm_mon;       /* Month of year (0 - 11) */
int tm_year;      /* Year - 1900 */
int tm_wday;      /* Day of week (Sunday = 0) */
int tm_yday;      /* Day of year (0 - 365) */
int tm_isdst;     /* Nonzero = Daylight saving time */

The mktime subroutine is the reverse function of the localtime subroutine. The mktime subroutine converts the tm structure into the time in seconds since 00:00:00 UTC, 1 January 1970. The tm_wday and tm_yday fields are ignored, and the other components of the tm structure are not restricted to the ranges specified in the /usr/include/time.h file. The value of the tm_isdst field determines the following actions of the mktime subroutine:
Initially presumes that Daylight Savings Time (DST) is not in effect.

>0 Initially presumes that DST is in effect.

-1 Actively determines whether DST is in effect from the specified time and the local time zone. Local time zone information is set by the tzset subroutine.

Upon successful completion, the mktime subroutine sets the values of the tm_wday and tm_yday fields appropriately. Other fields are set to represent the specified time since January 1, 1970. However, the values are forced to the ranges specified in the /usr/include/time.h file. The final value of the tm_mday field is not set until the values of the tm_mon and tm_year fields are determined.

Note: The mktime subroutine cannot convert time values before 00:00:00 UTC, January 1, 1970 and after 03:14:07 UTC, January 19, 2038.

The difftime subroutine computes the difference between two calendar times: the Time1 and Time0 parameters.

The asctime subroutine converts a tm structure to a 26-character string of the same format as ctime.

If the TZ environment variable is defined, then its value overrides the default time zone, which is the U.S. Eastern time zone. The environment facility contains the format of the time zone information specified by TZ. TZ is usually set when the system is started with the value that is defined in either the /etc/environment or /etc/profile files. However, it can also be set by the user as a regular environment variable for performing alternate time zone conversions.

The tzset subroutine sets the timezone, daylight, and tzname external variables to reflect the setting of TZ. The tzset subroutine is called by ctime and localtime, and it can also be called explicitly by an application program.

The timezone external variable contains the difference, in seconds, between UTC and local standard time. For example, the value of timezone is 5 * 60 * 60 for U.S. Eastern Standard Time.

The daylight external variable is nonzero when a daylight-saving time conversion should be applied. By default, this conversion follows the standard U.S. conventions; other conventions can be specified. The default conversion algorithm adjusts for the peculiarities of U.S. daylight saving time in 1974 and 1975.

The tzname external variable contains the name of the standard time zone (tzname[0]) and of the time zone when Daylight Savings Time is in effect (tzname[1]). For example:

    char *tzname[2] = {"EST", "EDT"};

The time.h file contains declarations of all these subroutines and externals and the tm structure.

Parameters

Clock Specifies the pointer to the time value in seconds.
Timeptr Specifies the pointer to a tm structure.
Time1 Specifies the pointer to a time_t structure.
Time0 Specifies the pointer to a time_t structure.
Tm Specifies the pointer to a tm structure.

Return Values

Attention: The return values point to static data that is overwritten by each call.

The tzset subroutine returns no value.
The `mktime` subroutine returns the specified time in seconds encoded as a value of type `time_t`. If the time cannot be represented, the function returns the value `(time_t)-1`.

The `localtime` and `gmtime` subroutines return a pointer to the `struct tm`.

The `ctime` and `asctime` subroutines return a pointer to a 26-character string.

The `difftime` subroutine returns the difference expressed in seconds as a value of type `double`.

**Related Information**

The `getenv` subroutine, `gettimeofday`, `settimeofday`, `resttime`, or `time` subroutine, `strftime` subroutine.

Time data manipulation services in *Operating system and device management*.


Subroutines, Example Programs, and Libraries in *AIX 5L Version 5.3 General Programming Concepts: Writing and Debugging Programs*.

---

**ctime64, localtime64, gmtime64, mktime64, difftime64, or asctime64 Subroutine**

**Purpose**

Converts the formats of date and time representations.

**Library**

Standard C Library (`libc.a`)

**Syntax**

```c
#include <time.h>

char *ctime64 (Clock)
const time64_t *Clock;

struct tm *localtime64 (Clock)
const time64_t *Clock;

struct tm *gmtime64 (Clock)
const time64_t *Clock;

time64_t mktime64 (Timeptr)
struct tm *Timeptr;

double difftime64 (Time1, Time0)
time64_t Time0, Time1;

char *asctime64 (Tm)
const struct tm *Tm;
```
Description

Attention: Do not use the ctime, localtime, gmtime, or asctime subroutine in a multithreaded environment. See "ctime64_r, localtime64_r, gmtime64_r, or asctime64_r Subroutine" on page 204 for multithread alternatives.

The ctime64 subroutine converts a time value pointed to by the Clock parameter, which represents the time in seconds since 00:00:00 Coordinated Universal Time (UTC), January 1, 1970, into a 26-character string in the following form:
Sun Sept 16 01:03:52 1973

The width of each field is always the same as shown here.

The ctime64 subroutine adjusts for the time zone and daylight saving time, if it is in effect.

The localtime64 subroutine converts the 64 bit long pointed to by the Clock parameter, which contains the time in seconds since 00:00:00 UTC, 1 January 1970, into a tm structure. The localtime64 subroutine adjusts for the time zone and for daylight saving time, if it is in effect. Use the time-zone information as though localtime64 called tzset.

The gmtime64 subroutine converts the 64 bit long pointed to by the Clock parameter into a tm structure containing the Coordinated Universal Time (UTC), which is the time standard that the operating system uses.

Note: UTC is the international time standard intended to replace GMT.

The mktime64 subroutine is the reverse function of the localtime64 subroutine. The mktime64 subroutine converts the tm structure into the time in seconds since 00:00:00 UTC, 1 January 1970. The tm_wday and tm_yday fields are ignored, and the other components of the tm structure are not restricted to the ranges specified in the /usr/include/time.h file. The value of the tm_isdst field determines the following actions of the mktime64 subroutine:

<table>
<thead>
<tr>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Initially presumes that Daylight Savings Time (DST) is not in effect.</td>
</tr>
<tr>
<td>&gt;0</td>
<td>Initially presumes that DST is in effect.</td>
</tr>
<tr>
<td>-1</td>
<td>Actively determines whether DST is in effect from the specified time and the local time zone. Local time zone information is set by the tzset subroutine.</td>
</tr>
</tbody>
</table>

Upon successful completion, the mktime64 subroutine sets the values of the tm_wday and tm_yday fields appropriately. Other fields are set to represent the specified time since January 1, 1970. However, the values are forced to the ranges specified in the /usr/include/time.h file. The final value of the tm_mday field is not set until the values of the tm_mon and tm_year fields are determined.

Note: The mktime64 subroutine cannot convert time values before 00:00:00 UTC, January 1, 1970 and after 23:59:59 UTC, December 31, 9999.

Note: The difftime64 subroutine computes the difference between two calendar times: the Time1 and Time0 parameters.

Note: The asctime64 subroutine converts a tm structure to a 26-character string of the same format as ctime64.

Parameters

Clock Specifies the pointer to the time value in seconds.
Timeptr Specifies the pointer to a tm structure.
Specifies the pointer to a `time64_t` structure.

Specifies the pointer to a `time64_t` structure.

Specifies the pointer to a `tm` structure.

Return Values

Attention: The return values point to static data that is overwritten by each call.

The `mktime64` subroutine returns the specified time in seconds encoded as a value of type `time64_t`. If the time cannot be represented, the function returns the value `(time64_t)-1`.

The `localtime64` and `gmtime64` subroutines return a pointer to the `tm` struct.

The `ctime64` and `asctime64` subroutines return a pointer to a 26-character string.

The `difftime64` subroutine returns the difference expressed in seconds as a value of type long double.

Related Information

- "ctime64_r, localtime64_r, gmtime64_r, or asctime64_r Subroutine" on page 360
- "getenv Subroutine" on page 441
- "strftime subroutine.

Time data manipulation services in Operating system and device management.


Subroutines, Example Programs, and Libraries in AIX 5L Version 5.3 General Programming Concepts: Writing and Debugging Programs.

ctime64_r, localtime64_r, gmtime64_r, or asctime64_r Subroutine

**Purpose**

Converts the formats of date and time representations.

**Library**

Thread-Safe C Library (`libc_r.a`)

**Syntax**

```c
#include <time.h>

char *ctime64_r(const time64_t *Timer, BufferPointer);
char *BufferPointer;

struct tm *localtime64_r(const time64_t *Timer, CurrentTime);
struct tm *CurrentTime;

struct tm *gmtime64_r(const time64_t *Timer, XTime);
struct tm *XTime;
```
char *asctime64_r (TimePointer, BufferPointer)
const struct tm * TimePointer;
char * BufferPointer;

Description
The ctime64_r subroutine converts a time value pointed to by the Timer parameter, which represents the time in seconds since 00:00:00 Coordinated Universal Time (UTC), January 1, 1970, into the character array pointed to by the BufferPointer parameter. The character array should have a length of at least 26 characters so the converted time value fits without truncation. The converted time value string takes the form of the following example:
Sun Sept 16 01:03:52 1973
The width of each field is always the same as shown here. Thus, ctime will only return dates up to December 31, 9999.

The ctime64_r subroutine adjusts for the time zone and daylight saving time, if it is in effect.

The localtime64_r subroutine converts the time64_t structure pointed to by the Timer parameter, which contains the time in seconds since 00:00:00 UTC, January 1, 1970, into the tm structure pointed to by the CurrentTime parameter. The localtime64_r subroutine adjusts for the time zone and for daylight saving time, if it is in effect.

The gmtime64_r subroutine converts the time64_t structure pointed to by the Timer parameter into the tm structure pointed to by the XTime parameter.

The tm structure is defined in the time.h header file. The time.h file contains declarations of these subroutines, externals, and the tm structure.

Theasctime64_r subroutine converts the tm structure pointed to by the TimePointer parameter into a 26-character string in the same format as the ctime64_r subroutine. The results are placed into the character array, BufferPointer. The BufferPointer parameter points to the resulting character array, which takes the form of the following example:
Sun Sept 16 01:03:52 1973

Programs using this subroutine must link to the libpthread.a library.

Parameters
Timer Points to a time64_t structure, which contains the number of seconds since 00:00:00 UTC, January 1, 1970.
BufferPointer Points to a character array at least 26 characters long.
CurrentTime Points to a tm structure. The result of the localtime64_r subroutine is placed here.
XTime Points to a tm structure used for the results of the gmtime64_r subroutine.
TimePointer Points to a tm structure used as input to the asctime64_r subroutine.

Return Values
The localtime64_r and gmtime64_r subroutines return a pointer to the tm structure. The asctime64_r subroutine returns NULL if either TimePointer or BufferPointer is NULL.

The ctime64_r and asctime64_r subroutines return a pointer to a 26-character string. The ctime64_r subroutine returns NULL if the BufferPointer is NULL.

The difftime64 subroutine returns the difference expressed in seconds as a value of type long double.
Files

/usr/include/time.h

Defines time macros, data types, and structures.

Related Information

“ctime64, localtime64, gmtime64, mktime64, difftime64, or asctime64 Subroutine” on page 202

_subroutines, Example Programs, and Libraries_in AIX 5L Version 5.3 General Programming Concepts: Writing and Debugging Programs.

ctime_r, localtime_r, gmtime_r, or asctime_r Subroutine

Purpose

Converts the formats of date and time representations.

Library

Thread-Safe C Library (libc_r.a)

Syntax

```
#include <time.h>

char *ctime_r(Timer, BufferPointer)
const time_t *Timer,
char *BufferPointer;

struct tm *localtime_r(Timer, CurrentTime)
const time_t *Timer;
struct tm *CurrentTime;

struct tm *gmtime_r(Timer, XTime)
const time_t *Timer;
struct tm *XTime;

char *asctime_r(TimePointer, BufferPointer)
const struct tm *TimePointer;
char *BufferPointer;
```

Description

The ctime_r subroutine converts a time value pointed to by the Timer parameter, which represents the time in seconds since 00:00:00 Coordinated Universal Time (UTC), January 1, 1970, into the character array pointed to by the BufferPointer parameter. The character array should have a length of at least 26 characters so the converted time value fits without truncation. The converted time value string takes the form of the following example:

Sun Sep 16 01:03:52 1973

The width of each field is always the same as shown here.

The ctime_r subroutine adjusts for the time zone and daylight saving time, if it is in effect.
The `localtime_r` subroutine converts the `time_t` structure pointed to by the `Timer` parameter, which contains the time in seconds since 00:00:00 UTC, January 1, 1970, into the `tm` structure pointed to by the `CurrentTime` parameter. The `localtime_r` subroutine adjusts for the time zone and for daylight saving time, if it is in effect.

The `gmtime_r` subroutine converts the `time_t` structure pointed to by the `Timer` parameter into the `tm` structure pointed to by the `XTime` parameter.

The `tm` structure is defined in the `time.h` header file. The `time.h` file contains declarations of these subroutines, externals, and the `tm` structure.

The `asctime_r` subroutine converts the `tm` structure pointed to by the `TimePointer` parameter into a 26-character string in the same format as the `ctime_r` subroutine. The results are placed into the character array, `BufferPointer`. The `BufferPointer` parameter points to the resulting character array, which takes the form of the following example:

```
Sun Sep 16 01:03:52 1973
```

Programs using this subroutine must link to the `libpthreads.a` library.

### Parameters

- **Timer**: Points to a `time_t` structure, which contains the number of seconds since 00:00:00 UTC, January 1, 1970.
- **BufferPointer**: Points to a character array at least 26 characters long.
- **CurrentTime**: Points to a `tm` structure. The result of the `localtime_r` subroutine is placed here.
- **XTime**: Points to a `tm` structure used for the results of the `gmtime_r` subroutine.
- **TimePointer**: Points to a `tm` structure used as input to the `asctime_r` subroutine.

### Return Values

The `localtime_r` and `gmtime_r` subroutines return a pointer to the `tm` structure. The `asctime_r` subroutine returns NULL if either `TimePointer` or `BufferPointer` are NULL.

The `ctime_r` and `asctime_r` subroutines return a pointer to a 26-character string. The `ctime_r` subroutine returns NULL if the `BufferPointer` is NULL.

### Files

`/usr/include/time.h`

Defines time macros, data types, and structures.

### Related Information

The `ctime`, `localtime`, `gmtime`, `mktime`, `difftime`, `asctime`, or `tzset` subroutine ("ctime, localtime, gmtime, mktime, difftime, asctime, or tzset Subroutine" on page 199) subroutine.

Subroutines, Example Programs, and Libraries and List of Multi-threaded Programming Subroutines in AIX 5L Version 5.3 General Programming Concepts: Writing and Debugging Programs.

**Purpose**
Classifies characters.

**Library**
Standard Character Library (libc.a)

**Syntax**
```
#include <ctype.h>

int isalpha (Character);
int isupper (Character);
int islower (Character);
int isdigit (Character);
int isxdigit (Character);
int isalnum (Character);
int isspace (Character);
int ispunct (Character);
int isprint (Character);
int isgraph (Character);
int iscntrl (Character);
int isascii (Character);
```

**Description**
The **ctype** subroutines classify character-coded integer values specified in a table. Each of these subroutines returns a nonzero value for True and 0 for False.

**Note:** The **ctype** subroutines should only be used on character data that can be represented by a single byte value (0 through 255). Attempting to use the **ctype** subroutines on multi-byte locale data may give inconsistent results. Wide character classification routines (such as **iswprint**, **iswlower**, etc.) should be used with dealing with multi-byte character data.

**Locale Dependent Character Tests**
The following subroutines return nonzero (True) based upon the character class definitions for the current locale.

**isalnum**
Returns nonzero for any character for which the **isalpha** or **isdigit** subroutine would return nonzero. The **isalnum** subroutine tests whether the character is of the **alpha** or **digit** class.
isalpha
Returns nonzero for any character for which the isupper or islower subroutines would return
nonzero. The isalpha subroutine also returns nonzero for any character defined as an alphabetic
character in the current locale, or for a character for which none of the iscntrl, isdigit, ispunct,
or isspace subroutines would return nonzero. The isalpha subroutine tests whether the
character is of the alpha class.
isupper
Returns nonzero for any uppercase letter [A through Z]. The isupper subroutine also returns
nonzero for any character defined to be uppercase in the current locale. The isupper subroutine
tests whether the character is of the upper class.
islower
Returns nonzero for any lowercase letter [a through z]. The islower subroutine also returns
nonzero for any character defined to be lowercase in the current locale. The islower subroutine
tests whether the character is of the lower class.
isspace
Returns nonzero for any white-space character (space, form feed, new line, carriage return,
horizontal tab or vertical tab). The isspace subroutine tests whether the character is of the
space class.
ispunct
Returns nonzero for any character for which the isprint subroutine returns nonzero, except the
space character and any character for which the isalnum subroutine would return nonzero. The
ispunct subroutine also returns nonzero for any locale-defined character specified as a
punctuation character. The ispunct subroutine tests whether the character is of the punct class.
isprint
Returns nonzero for any printing character. Returns nonzero for any locale-defined character that
is designated a printing character. This routine tests whether the character is of the print class.
isgraph
Returns nonzero for any character for which the isprint character returns nonzero, except the
space character. The isgraph subroutine tests whether the character is of the graph class.
iscntrl
Returns nonzero for any character for which the isprint subroutine returns a value of False (0)
and any character that is designated a control character in the current locale. For the C locale,
control characters are the ASCII delete character (0177 or 0x7F), or an ordinary control character
(less than 040 or 0x20). The iscntrl subroutine tests whether the character is of the cntrl class.

Locale Independent Character Tests
The following subroutines return nonzero for the same characters, regardless of the locale:

isdigit
Character is a digit in the range [0 through 9].
isxdigit
Character is a hexadecimal digit in the range [0 through 9], [A through F], or [a through f].
isascii
Character is an ASCII character whose value is in the range 0 through 0177 (0 through 0x7F),
inclusive.

Parameter

Character Indicates the character to be tested (integer value).

Return Codes
The ctype subroutines return nonzero (True) if the character specified by the Character parameter is a
member of the selected character class; otherwise, a 0 (False) is returned.

Related Information
The setlocale subroutine.

List of Character Manipulation Services and Subroutines, Example Programs, and Libraries in AIX 5L
Version 5.3 General Programming Concepts: Writing and Debugging Programs.

National Language Support Overview in AIX 5L Version 5.3 National Language Support Guide and
Reference.
cuserid Subroutine

Purpose
Gets the alphanumeric user name associated with the current process.

Library
Standard C Library (libc.a)

Use the libc_r.a library to access the thread-safe version of this subroutine.

Syntax

```c
#include <stdio.h>

char *cuserid (Name)
char *Name;
```

Description
The cuserid subroutine gets the alphanumeric user name associated with the current process. This subroutine generates a character string representing the name of a process's owner.

**Note:** The cuserid subroutine duplicates functionality available with the getpwuid and getuid subroutines. Present applications should use the getpwuid and getuid subroutines.

If the Name parameter is a null pointer, then a character string of size L_cuserid is dynamically allocated with malloc, and the character string representing the name of the process owner is stored in this area. The cuserid subroutine then returns the address of this area. Multithreaded application programs should use this functionality to obtain thread specific data, and then continue to use this pointer in subsequent calls to the cuserid subroutine. In any case, the application program must deallocate any dynamically allocated space with the free subroutine when the data is no longer needed.

If the Name parameter is not a null pointer, the character string is stored into the array pointed to by the Name parameter. This array must contain at least the number of characters specified by the constant L_cuserid. This constant is defined in the stdio.h file.

If the user name cannot be found, the cuserid subroutine returns a null pointer; if the Name parameter is not a null pointer, a null character ("\0") is stored in Name [0].

Parameter

Name Points to a character string representing a user name.

Related Information
The endpwent ("getpwent, getpwuid, getpwnam, putpwent, setpwent, or endpwent Subroutine" on page 417) subroutine, getlogin ("getlogin Subroutine" on page 389), getpwent ("getpwent, getpwuid, getpwnam, putpwent, setpwent, or endpwent Subroutine" on page 417), getpwnam ("getpwent, getpwuid, getpwnam, putpwent, setpwent, or endpwent Subroutine" on page 417), getpwuid ("getpwent, getpwuid, getpwnam, putpwent, setpwent, or endpwent Subroutine" on page 417), or putpwent ("getpwent, getpwuid, getpwnam, putpwent, setpwent, or endpwent Subroutine" on page 417) subroutine.

Input and Output Handling Programmer's Overview in AIX 5L Version 5.3 General Programming Concepts: Writing and Debugging Programs.
defssys Subroutine

Purpose
Initializes the SRCsubsys structure with default values.

Library
System Resource Controller Library (libsrc.a)

Syntax
#include <sys/srcobj.h>
#include <spc.h>

void defssys(SRCSubsystem)
struct SRCsubsys *SRCSubsystem;

Description
The defssys subroutine initializes the SRCsubsys structure of the /usr/include/sys/srcobj.h file with the following default values:

<table>
<thead>
<tr>
<th>Field</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>display</td>
<td>SRCYES</td>
</tr>
<tr>
<td>multi</td>
<td>SRCNO</td>
</tr>
<tr>
<td>contact</td>
<td>SRCSOCKET</td>
</tr>
<tr>
<td>waittime</td>
<td>TIMELIMIT</td>
</tr>
<tr>
<td>priority</td>
<td>20</td>
</tr>
<tr>
<td>action</td>
<td>ONCE</td>
</tr>
<tr>
<td>standerr</td>
<td>/dev/console</td>
</tr>
<tr>
<td>standin</td>
<td>/dev/console</td>
</tr>
<tr>
<td>standout</td>
<td>/dev/console</td>
</tr>
</tbody>
</table>

All other numeric fields are set to 0, and all other alphabetic fields are set to an empty string.

This function must be called to initialize the SRCsubsys structure before an application program uses this structure to add records to the subsystem object class.

Parameters
SRCSubsystem Points to the SRCsubsys structure.

Related Information
The addssys subroutine.

Defining Your Subsystem to the SRC, List of SRC Subroutines, System Resource Controller (SRC) Overview for Programmers

delssys Subroutine

Purpose
Removes the subsystem objects associated with the SubsystemName parameter.
Library
System Resource Controller Library (libsrc.a)

Syntax
#include <sys/srcobj.h>
#include <spc.h>

int delssys (SubsystemName);
char *SubsystemName;

Description
The delssys subroutine removes the subsystem objects associated with the specified subsystem. This
removes all objects associated with that subsystem from the following object classes:
• Subsystem
• Subserver Type
• Notify

The program running with this subroutine must be running with the group system.

Parameter
SubsystemName Specifies the name of the subsystem.

Return Values
Upon successful completion, the delssys subroutine returns a positive value. If no record is found, a value
of 0 is returned. Otherwise, -1 is returned and the odmerrno variable is set to indicate the error. See
"Appendix B. ODM Error Codes (Appendix B, “ODM Error Codes,” on page 1325)" for a description of
possible odmerrno values.

Security
Privilege Control:

SET_PROC_AUDIT kernel privilege

Files Accessed:

Mode File
644 /etc/objrepos/SRCsubsys
644 /etc/objrepos/SRCsubsvr
644 /etc/objrepos/SRCnotify

Auditing Events:

Event Information
SRC_Delssys Lists in an audit log the name of the subsystem being removed.

Files

/etc/objrepos/SRCsubsys SRC Subsystem Configuration object class.
/etc/objrepos/SRCsubsvr SRC Subsystem Configuration object class.
Related Information

The addssys subroutine, chssys subroutine.

The chssys command, mkssys command, rmssys command.

List of SRC Subroutines and System Resource Controller (SRC) Overview for Programmers in AIX 5L Version 5.3 General Programming Concepts: Writing and Debugging Programs.

dirname Subroutine

Purpose

Report the parent directory name of a file path name.

Library

Standard C Library (libc.a)

Syntax

#include <libgen.h>

cchar *dirname (path)
cchar *path

Description

Given a pointer to a character string that contains a file system path name, the dirname subroutine returns a pointer to a string that is the parent directory of that file. Trailing "/" characters in the path are not counted as part of the path.

If path is a null pointer or points to an empty string, a pointer to a static constant "." is returned.

The dirname and basename subroutines together yield a complete path name. dirname (path) is the directory where basename (path) is found.

Parameters

path Character string containing a file system path name.

Return Values

The dirname subroutine returns a pointer to a string that is the parent directory of path. If path or "path is a null pointer or points to an empty string, a pointer to a string "." is returned. The dirname subroutine may modify the string pointed to by path and may return a pointer to static storage that may then be overwritten by sequent calls to the dirname subroutine.
Examples
A simple file name and the strings "." and ".." all have "." as their return value.

<table>
<thead>
<tr>
<th>Input string</th>
<th>Output string</th>
</tr>
</thead>
<tbody>
<tr>
<td>/usr/lib</td>
<td>/usr</td>
</tr>
<tr>
<td>/usr/</td>
<td>/</td>
</tr>
<tr>
<td>usr</td>
<td>.</td>
</tr>
<tr>
<td>/</td>
<td>/</td>
</tr>
<tr>
<td>.</td>
<td>.</td>
</tr>
<tr>
<td>..</td>
<td>.</td>
</tr>
</tbody>
</table>

The following code reads a path name, changes directory to the appropriate directory, and opens the file.

```c
char path [MAXPATHEN], *pathcopy;
int fd;
fgets (path, MAXPATHEN, stdin);
pathcopy = strdup (path);
chdir (dirname (pathcopy));
fd = open (basename (path), 0_RDONLY);
```

Related Information
The basename "basename Subroutine" on page 117 or chdir "chdir Subroutine" on page 147 subroutine.

disclaim Subroutine

Purpose
Disclaims the content of a memory address range.

Syntax
```c
#include <sys/shm.h>

int disclaim ( Address, Length, Flag)
char *Address;
unsigned int Length, Flag;
```

Description
The disclaim subroutine marks an area of memory having content that is no longer needed. The system then stops paging the memory area. This subroutine cannot be used on memory that is mapped to a file by the shmat subroutine.

Parameters
- **Address**: Points to the beginning of the memory area.
- **Length**: Specifies the length of the memory area in bytes.
- **Flag**: Must be the value ZERO_MEM, which indicates that each memory location in the address range should be set to 0.

Return Values
When successful, the disclaim subroutine returns a value of 0.
**Error Codes**

If the `disclaim` subroutine is unsuccessful, it returns a value of -1 and sets the `errno` global variable to indicate the error. The `disclaim` subroutine is unsuccessful if one or more of the following are true:

- **EFAULT** The calling process does not have write access to the area of memory that begins at the `Address` parameter and extends for the number of bytes specified by the `Length` parameter.
- **EINVAL** The value of the `Flag` parameter is not valid.
- **EINVAL** The memory area is mapped to a file.

---

**dlclose Subroutine**

**Purpose**
Closes and unloads a module loaded by the `dlopen` subroutine.

**Syntax**
```c
#include <dlfcn.h>
int dlclose(Data);
void *Data;
```

**Description**
The `dlclose` subroutine is used to remove access to a module loaded with the `dlopen` subroutine. In addition, access to dependent modules of the module being unloaded is removed as well.

Modules being unloaded with the `dlclose` subroutine will not be removed from the process’s address space if they are still required by other modules. Nevertheless, subsequent uses of `Data` are invalid, and further uses of symbols that were exported by the module being unloaded result in undefined behavior.

**Parameters**
- **Data** A loaded module reference returned from a previous call to `dlopen`.

**Return Values**
Upon successful completion, 0 (zero) is returned. Otherwise, `errno` is set to `EINVAL`, and the return value is also `EINVAL`. Even if the `dlclose` subroutine succeeds, the specified module may still be part of the process’s address space if the module is still needed by other modules.

**Error Codes**
- **EINVAL** The `Data` parameter does not refer to a module opened by `dlopen` that is still open. The parameter may be corrupt or the module may have been unloaded by a previous call to `dlclose`.

**Related Information**
The `dlerror` ("dlerror Subroutine" on page 216) subroutine, `dlopen` ("dlopen Subroutine" on page 216) subroutine, `load` ("Load Subroutine" on page 721) subroutine, `loadquery` ("loadquery Subroutine" on page 726) subroutine, `unload` subroutine, `loadbind` ("loadbind Subroutine" on page 725) subroutine.

The `ld` command.

The Shared Libraries and Shared Memory Overview and Subroutines Overview in AIX 5L Version 5.3 General Programming Concepts: Writing and Debugging Programs.
dlerror Subroutine

Purpose
Returns a pointer to information about the last dlopen, dlsym, or dlclose error.

Syntax
#include <dlfcn.h>
char *dlerror(void);

Description
The dlerror subroutine is used to obtain information about the last error that occurred in a dynamic loading routine (that is, dlopen, dlsym, or dlclose). The returned value is a pointer to a null-terminated string without a final newline. Once a call is made to this function, subsequent calls without any intervening dynamic loading errors will return NULL.

Applications can avoid calling the dlerror subroutine, in many cases, by examining errno after a failed call to a dynamic loading routine. If errno is ENOEXEC, the dlerror subroutine will return additional information. In all other cases, dlerror will return the string corresponding to the value of errno.

The dlerror function may invoke loadquery to ascertain reasons for a failure. If a call is made to load or unload between calls to dlopen and dlerror, incorrect information may be returned.

Return Values
A pointer to a static buffer is returned; a NULL value is returned if there has been no error since the last call to dlerror. Applications should not write to this buffer; they should make a copy of the buffer if they wish to preserve the buffer's contents.

Related Information
The load ("load Subroutine" on page 721) subroutine, loadbind ("loadbind Subroutine" on page 725) subroutine, loadquery ("loadquery Subroutine" on page 726) subroutine, unload subroutine, dlopen ("dlopen Subroutine") subroutine, dlclose ("dlclose Subroutine" on page 215) subroutine.

The ld command.

The Shared Libraries and Shared Memory Overview and Subroutines Overview in AIX 5L Version 5.3 General Programming Concepts: Writing and Debugging Programs.

dlopen Subroutine

Purpose
Dynamically load a module into the calling process.

Syntax
#include <dlfcn.h>
void *dlopen (FilePath, Flags);
const char *FilePath;
int Flags;
Description
The **dlopen** subroutine loads the module specified by *FilePath* into the executing process's address space. Dependents of the module are automatically loaded as well. If the module is already loaded, it is not loaded again, but a new, unique value will be returned by the **dlopen** subroutine.

The value returned by **dlopen** may be used in subsequent calls to **dlsym** and **dlclose**. If an error occurs during the operation, **dlopen** returns **NULL**.

If the main application was linked with the `-brtl` option, then the runtime linker is invoked by **dlopen**. If the module being loaded was linked with runtime linking enabled, both intra-module and inter-module references are overridden by any symbols available in the main application. If runtime linking was enabled, but the module was not built enabled, then all inter-module references will be overridden, but some intra-module references will not be overridden.

If the module being opened with **dlopen** or any of its dependents is being loaded for the first time, initialization routines for these newly-loaded routines are called (after runtime linking, if applicable) before **dlopen** returns. Initialization routines are the functions specified with the `-binitfini`: linker option when the module was built. (Refer to the **ld** command for more information about this option.)

Notes:
1. The initialization functions need not have any special names, and multiple functions per module are allowed.
2. If the module being loaded has read-other permission, the module is loaded into the global shared library segment. Modules loaded into the global shared library segment are not unloaded even if they are no longer being used. Use the **slibclean** command to remove unused modules from the global shared library segment.

The **LIBPATH** or **LD_LIBRARY_PATH** environment variables can be used to specify a list of directories in which **dlopen** searches for the named module. The running application also contains a set of library search paths that were specified when the application was linked; these paths are searched after any paths found in **LIBPATH** or **LD_LIBRARY_PATH**.

*FilePath* Specifies the name of a file containing the loadable module. This parameter can be contain an absolute path, a relative path, or no path component. If *FilePath* contains a slash character, *FilePath* is used directly, and no directories are searched.

If the *FilePath* parameter is `/unix`, **dlopen** returns a value that can be used to look up symbols in the current kernel image, including those symbols found in any kernel extension that was available at the time the process began execution.

If the value of *FilePath* is **NULL**, a value for the main application is returned. This allows dynamically loaded objects to look up symbols in the main executable, or for an application to examine symbols available within itself.

Flags
Specifies variations of the behavior of **dlopen**. Either **RTLD_NOW** or **RTLD_LAZY** must always be specified. Other flags may be OR'ed with **RTLD_NOW** or **RTLD_LAZY**.

**RTLD_NOW** Load all dependents of the module being loaded and resolve all symbols.

**RTLD_LAZY** Specifies the same behavior as **RTLD_NOW**. In a future release of the operating system, the behavior of the **RTLD_LAZY** may change so that loading of dependent modules is deferred of resolution of some symbols is deferred.

**RTLD_GLOBAL** Allows symbols in the module being loaded to be visible when resolving symbols used by other **dlopen** calls. These symbols will also be visible when the main application is opened with **dlopen**(NULL, **mode**).
Prevent symbols in the module being loaded from being used when resolving symbols used by other `dlopen` calls. Symbols in the module being loaded can only be accessed by calling `dlsym` subroutine. If neither `RTLD_GLOBAL` nor `RTLD_LOCAL` is specified, the default is `RTLD_LOCAL`. If both flags are specified, `RTLD_LOCAL` is ignored.

The `dlopen` subroutine can be used to load a module that is a member of an archive. The `L_LOADMEMBER` flag is used when the `load` subroutine is called. The module name `FilePath` names the archive and archive member according to the rules outlined in the `load` subroutine.

Prevents deferred imports in the module being loaded from being automatically resolved by subsequent loads. The `L_NOAUTODEFER` flag is used when the `load` subroutine is called.

Ordinarily, modules built for use by the `dlopen` and `dlsym` subroutines will not contain deferred imports. However, deferred imports can be still used. A module opened with `dlopen` may provide definitions for deferred imports in the main application, for modules loaded with the `load` subroutine (if the `L_NOAUTODEFER` flag was not used), and for other modules loaded with the `dlopen` subroutine (if the `RTLD_NOAUTODEFER` flag was not used).

**Return Values**

Upon successful completion, `dlopen` returns a value that can be used in calls to the `dlsym` and `dlclose` subroutines. The value is not valid for use with the `loadbind` and `unload` subroutines.

If the `dlopen` call fails, NULL (a value of 0) is returned and the global variable `errno` is set. If `errno` contains the value `ENOEXEC`, further information is available via the `dlerror` function.

**Error Codes**

See the `load` subroutine for a list of possible `errno` values and their meanings.

**Related Information**

The `dlclose` subroutine, `dlsym` subroutine, `dlerror` subroutine, `load` subroutine, `loadbind` subroutine, `loadquery` subroutine, `unload` subroutine.

The `ld` command.

**dlsym Subroutine**

**Purpose**

Looks up the location of a symbol in a module that is loaded with `dlsym`.

**Syntax**

```c
#include <dlfcn.h>
void *dlsym(Handle, Symbol);
void *Handle;
const char *Symbol;
```
Description

The dlsym subroutine looks up a named symbol exported from a module loaded by a previous call to the dlopen subroutine. Only exported symbols are found by dlsym. See the ld command to see how to export symbols from a module.

**Handle**
Specifies a value returned by a previous call to dlopen or one of the special handles RTLD_DEFAULT, RTLD_NEXT or RTLD_MYSELF.

**Symbol**
Specifies the name of a symbol exported from the referenced module in the form of a NULL-terminated string or the special symbol name RTLD_ENTRY.

**Note:** C++ symbol names should be passed to dlsym in mangled form; dlsym does not perform any name demangling on behalf of the calling application.

In case of the special handle RTLD_DEFAULT, dlsym searches for the named symbol starting with the first module loaded. It then proceeds through the list of initial loaded modules and any global modules obtained with dlopen until a match is found. This search follows the default model employed to relocate all modules within the process.

In case of the special handle RTLD_NEXT, dlsym searches for the named symbol in the modules that were loaded following the module from which the dlsym call is being made.

In case of the special handle RTLD_MYSELF, dlsym searches for the named symbol in the modules that were loaded starting with the module from which the dlsym call is being made.

In case of the special symbol name RTLD_ENTRY, dlsym returns the module’s entry point. The entry point, if present, is the value of the module’s loader section symbol marked as entry point.

In case of RTLD_DEFAULT, RTLD_NEXT, and RTLD_MYSELF, if the modules being searched have been loaded from dlopen calls, dlsym searches the module only if the caller is part of the same dlopen dependency hierarchy, or if the module was given global search access. See dlopen for a discussion of the RTLD_GLOBAL mode.

A search for the named symbol is based upon breadth-first ordering of the module and its dependants. If the module was constructed using the -G or -brtl linker option, the module’s dependants will include all modules named on the ld command line, in the original order. The dependants of a module that was not linked with the -G or -brtl linker option will be listed in an unspecified order.

**Return Values**

If the named symbol is found, its address is returned. If the named symbol is not found, NULL is returned and errno is set to 0. If Handle or Symbol is invalid, NULL is returned and errno is set to EINVAL.

If the first definition found is an export of an imported symbol, this definition will satisfy the search. The address of the imported symbol is returned. If the first definition is a deferred import, the definition is ignored and the search continues.

If the named symbol refers to a BSS symbol ( uninitialized data structure), the search continues until an initialized instance of the symbol is found or the module and all of its dependants have been searched. If an initialized instance is found, its address is returned; otherwise, the address of the first uninitialized instance is returned.
**Error Codes**

EINVAL

If the Handle parameter does not refer to a module opened by `dlopen` that is still loaded or if the Symbol parameter points to an invalid address, the `dlsym` subroutine returns NULL and `errno` is set to EINVAL.

**Related Information**

The `dlclose` subroutine, `dlerror` subroutine, `dlopen` subroutine, `load` subroutine, `loadbind` subroutine, `loadquery` subroutine, `unload` subroutine.

The ld command.

**drand48, erand48, jrand48, Icong48, Irand48, mrand48, nrand48, seed48, or srand48 Subroutine**

**Purpose**

Generate uniformly distributed pseudo-random number sequences.

**Library**

Standard C Library (libc.a)

**Syntax**

```
#include <stdlib.h>
double drand48 (void)

double erand48 (xsubi)
unsigned short int xsubi[3];
long int jrand48 (xsubi)
unsigned short int xsubi[3];

void Icong48 (Parameter)
unsigned short int Parameter[7];
long int Irand48 (void)
long int mrand48 (void)
long int nrand48 (xsubi)
unsigned short int xsubi[3];

unsigned short int *seed48 (Seed16v)
unsigned short int Seed16v[3];

void srand48 (SeedValue)
long int SeedValue;
```

**Description**

**Attention:** Do not use the `drand48`, `erand48`, `jrand48`, `Icong48`, `Irand48`, `mrand48`, `nrand48`, `seed48`, or `srand48` subroutine in a multithreaded environment.

This family of subroutines generates pseudo-random numbers using the linear congruential algorithm and 48-bit integer arithmetic.
The \texttt{drand48} subroutine and the \texttt{erand48} subroutine return positive double-precision floating-point values uniformly distributed over the interval \([0.0, 1.0)\).

The \texttt{lrand48} subroutine and the \texttt{nrand48} subroutine return positive long integers uniformly distributed over the interval \([0,2^{31})\).

The \texttt{mrand48} subroutine and the \texttt{jrand48} subroutine return signed long integers uniformly distributed over the interval \([-2^{31}, 2^{31})\).

The \texttt{srand48} subroutine, \texttt{seed48} subroutine, and \texttt{lcong48} subroutine initialize the random-number generator. Programs must call one of them before calling the \texttt{drand48}, \texttt{lrand48} or \texttt{mrand48} subroutines. (Although it is not recommended, constant default initializer values are supplied if the \texttt{drand48}, \texttt{lrand48} or \texttt{mrand48} subroutines are called without first calling an initialization subroutine.) The \texttt{erand48}, \texttt{nrand48}, and \texttt{jrand48} subroutines do not require that an initialization subroutine be called first.

The previous value pointed to by the \texttt{seed48} subroutine is stored in a 48-bit internal buffer, and a pointer to the buffer is returned by the \texttt{seed48} subroutine. This pointer can be ignored if it is not needed, or it can be used to allow a program to restart from a given point at a later time. In this case, the pointer is accessed to retrieve and store the last value pointed to by the \texttt{seed48} subroutine, and this value is then used to reinitialize, by means of the \texttt{seed48} subroutine, when the program is restarted.

All the subroutines work by generating a sequence of 48-bit integer values, \(x[i]\), according to the linear congruential formula:
\[ x[n+1] = (ax[n] + c) \mod m, \ n \geq 0 \]

The parameter \(m = 2^{48}\); hence 48-bit integer arithmetic is performed. Unless the \texttt{lcong48} subroutine has been called, the multiplier value \(a\) and the addend value \(c\) are:
\[ a = 5DEECE66D \text{ base 16} = 273673163155 \text{ base 8} \]
\[ c = B \text{ base 16} = 13 \text{ base 8} \]

**Parameters**

- \texttt{xsubi} specifies an array of three shorts, which, when concatenated together, form a 48-bit integer.
- \texttt{SeedValue} specifies the initialization value to begin randomization. Changing this value changes the randomization pattern.
- \texttt{Seed16v} specifies another seed value; an array of three unsigned shorts that form a 48-bit seed value.
- \texttt{Parameter} specifies an array of seven shorts, which specifies the initial \texttt{xsubi} value, the multiplier value \(a\) and the add-in value \(c\).

**Return Values**

The value returned by the \texttt{drand48}, \texttt{erand48}, \texttt{jrand48}, \texttt{lrand48}, \texttt{nrand48}, and \texttt{mrand48} subroutines is computed by first generating the next 48-bit \(x[i]\) in the sequence. Then the appropriate number of bits, according to the type of data item to be returned, are copied from the high-order (most significant) bits of \(x[i]\) and transformed into the returned value.

The \texttt{drand48}, \texttt{lrand48}, and \texttt{mrand48} subroutines store the last 48-bit \(x[i]\) generated into an internal buffer; this is why they must be initialized prior to being invoked.

The \texttt{erand48}, \texttt{jrand48}, and \texttt{nrand48} subroutines require the calling program to provide storage for the successive \(x[i]\) values in the array pointed to by the \texttt{xsubi} parameter. This is why these routines do not have to be initialized; the calling program places the desired initial value of \(x[i]\) into the array and pass it as a parameter.
By using different parameters, the `erand48`, `jrand48`, and `nrand48` subroutines allow separate modules of a large program to generate independent sequences of pseudo-random numbers. In other words, the sequence of numbers that one module generates does not depend upon how many times the subroutines are called by other modules.

The `lcong48` subroutine specifies the initial \( x[i] \) value, the multiplier value \( a \), and the addend value \( c \). The `Parameter` array elements `Parameter[0-2]` specify \( x[i] \), `Parameter[3-5]` specify the multiplier \( a \), and `Parameter[6]` specifies the 16-bit addend \( c \). After `lcong48` has been called, a subsequent call to either the `srand48` or `seed48` subroutine restores the standard \( a \) and \( c \) specified before.

The initializer subroutine `seed48` sets the value of \( x[i] \) to the 48-bit value specified in the array pointed to by the `Seed16v` parameter. In addition, `seed48` returns a pointer to a 48-bit internal buffer that contains the previous value of \( x[i] \) that is used only by `seed48`. The returned pointer allows you to restart the pseudo-random sequence at a given point. Use the pointer to copy the previous \( x[i] \) value into a temporary array. Then call `seed48` with a pointer to this array to resume processing where the original sequence stopped.

The initializer subroutine `srand48` sets the high-order 32 bits of \( x[i] \) to the 32 bits contained in its parameter. The low order 16 bits of \( x[i] \) are set to the arbitrary value 330E16.

**Related Information**
The `rand`, `srand` subroutine, `random`, `srandom`, `initstate`, or `setstate` subroutine.

**Subroutines Overview** in AIX 5L Version 5.3 General Programming Concepts: Writing and Debugging Programs.

---

### drem Subroutine

**Purpose**
Computes the IEEE Remainder as defined in the IEEE Floating-Point Standard.

**Libraries**
IEEE Math Library (`libm.a`

or System V Math Library (`libmsaa.a`)

**Syntax**
```c
#include <math.h>

double drem (double x, double y);
```

**Description**
The `drem` subroutine calculates the remainder \( r \) equal to \( x \) minus \( n \) to the \( x \) power multiplied by \( y \) \((r = x - n \cdot y)\), where the \( n \) parameter is the integer nearest the exact value of \( x \) divided by \( y \) \((x/y)\). If \(|n - x/y| = 1/2\), then the \( n \) parameter is an even value. Therefore, the remainder is computed exactly, and the absolute value of \( r \) \(|r|\) is less than or equal to the absolute value of \( y \) divided by 2 \((|y|/2)\).

The IEEE Remainder differs from the `fmod` subroutine in that the IEEE Remainder always returns an \( r \) parameter such that \(|r|\) is less than or equal to \(|y|/2\), while FMOD returns an \( r \) such that \(|r|\) is less than or equal to \(|y|\). The IEEE Remainder is useful for argument reduction for transcendental functions.

**Note:** Compile any routine that uses subroutines from the `libm.a` library with the `-lm` flag. For example: compile the `drem.c` file:


```cc
c drem.c -lm
```

**Note:** For new development, the **remainder** subroutine is the preferred interface.

## Parameters

- **x** Specifies double-precision floating-point value.
- **y** Specifies a double-precision floating-point value.

## Return Values

The `drem` subroutine returns a NaNQ value for `(x, 0)` and `(+/INF, y)`.

## Related Information

The `floor`, `ceil`, `nearest`, `trunc`, `rint`, `itrunc`, `fmod`, `fabs`, or `uitruns` ("floor, floorf, floorl, nearest, trunc, itrunc, or uitrunc Subroutine" on page 274) subroutine.

** Related Information **

Subroutines Overview in *AIX 5L Version 5.3 General Programming Concepts: Writing and Debugging Programs.*

### _end, _etext, or _edata Identifier

**Purpose**

Define the first addresses following the program, initialized data, and all data.

**Syntax**

```c
extern _end;
extern _etext;
extern _edata;
```

**Description**

The external names `_end`, `_etext`, and `_edata` are defined by the loader for all programs. They are not subroutines but identifiers associated with the following addresses:

- **_etext** The first address following the program text.
- **_edata** The first address following the initialized data region.
- **_end** The first address following the data region that is not initialized. The name `end` (with no underscore) defines the same address as does `_end` (with underscore).

The break value of the program is the first location beyond the data. When a program begins running, this location coincides with `end`. However, many factors can change the break value, including:

- The `brk` or `sbrk` subroutine
- The `malloc` subroutine
- The standard I/O subroutines
- The `-p` flag with the `cc` command

Therefore, use the `brk` or `sbrk(0)` subroutine, not the `end` address, to determine the break value of the program.
ecvt, fcvt, or gcvt Subroutine

Purpose
Converts a floating-point number to a string.

Library
Standard C Library (libc.a)

Syntax
```
#include <stdlib.h>

char *ecvt (double Value, int NumberOfDigits, int *DecimalPointer, int *Sign);
char *fcvt (double Value, int NumberOfDigits, int *DecimalPointer, int *Sign);
char *gcvt (double Value, int NumberOfDigits, char *Buffer);
```

Description
The ecvt, fcvt, and gcvt subroutines convert floating-point numbers to strings.

The ecvt subroutine converts the Value parameter to a null-terminated string and returns a pointer to it. The NumberOfDigits parameter specifies the number of digits in the string. The low-order digit is rounded according to the current rounding mode. The ecvt subroutine sets the integer pointed to by the DecimalPointer parameter to the position of the decimal point relative to the beginning of the string. (A negative number means the decimal point is to the left of the digits given in the string.) The decimal point itself is not included in the string. The ecvt subroutine also sets the integer pointed to by the Sign parameter to a nonzero value if the Value parameter is negative and sets a value of 0 otherwise.

The fcvt subroutine operates identically to the ecvt subroutine, except that the correct digit is rounded for C or FORTRAN F-format output of the number of digits specified by the NumberOfDigits parameter.

Note: In the F-format, the NumberOfDigits parameter is the number of digits desired after the decimal point. Large numbers produce a long string of digits before the decimal point, and then NumberOfDigits digits after the decimal point. Generally, the gcvt and ecvt subroutines are more useful for large numbers.

The gcvt subroutine converts the Value parameter to a null-terminated string, stores it in the array pointed to by the Buffer parameter, and then returns the Buffer parameter. The gcvt subroutine attempts to produce a string of the NumberOfDigits parameter significant digits in FORTRAN F-format. If this is not possible, the E-format is used. The gcvt subroutine suppresses trailing zeros. The string is ready for
printing, complete with minus sign, decimal point, or exponent, as appropriate. The radix character is
determined by the current locale (see setlocale subroutine). If the setlocale subroutine has not been
called successfully, the default locale, POSIX, is used. The default locale specifies a . (period) as the radix
character. The LC_NUMERIC category determines the value of the radix character within the current
locale.

The ecvt, fcvt, and gcvt subroutines represent the following special values that are specified in
ANSI/IEEE standards 754-1985 and 854-1987 for floating-point arithmetic:

<table>
<thead>
<tr>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Quiet NaN</td>
<td>Indicates a quiet not-a-number (NaNQ)</td>
</tr>
<tr>
<td>Signalling NaN</td>
<td>Indicates a signaling NaN</td>
</tr>
<tr>
<td>Infinity</td>
<td>Indicates an INF value</td>
</tr>
</tbody>
</table>

The sign associated with each of these values is stored in the Sign parameter.

**Note:** A value of 0 can be positive or negative. In the IEEE floating-point, zeros also have signs and set
the Sign parameter appropriately.

**Attention:** All three subroutines store the strings in a static area of memory whose contents are
overwritten each time one of the subroutines is called.

**Parameters**

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Value</td>
<td>Specifies some double-precision floating-point value.</td>
</tr>
<tr>
<td>NumberOfDigits</td>
<td>Specifies the number of digits in the string.</td>
</tr>
<tr>
<td>DecimalPointer</td>
<td>Specifies the position of the decimal point relative to the beginning of the string.</td>
</tr>
</tbody>
</table>
| Sign               | Specifies that the sign associated with the return value is placed in the Sign parameter. In
                  | IEEE floating-point, since 0 can be signed, the Sign parameter is set appropriately for
                  | signed 0.                                           |
| Buffer             | Specifies a character array for the string.      |

**Related Information**

The atof, strtol, atof, or strtof (“atof atoff Subroutine” on page 96) subroutine, fp_read_rnd, or
fp_swap_rnd (“fp_read_rnd or fp_swap_rnd Subroutine” on page 299) subroutine, printf (“printf, fprintf,
sprintf, snprintf, wsprintf, vprintf, vfprintf, vsprintf, or vwsprintf Subroutine” on page 1148) subroutine,
scanf subroutine.

---

**EnableCriticalSections, BeginCriticalSection, and EndCriticalSection Subroutine**

**Purpose**

Enables a thread to be exempted from timeslicing and signal suspension, and protects critical sections.

**Library**

Standard C Library (libc.a)
Syntax

```c
#include <sys/thread_ctl.h>

int EnableCriticalSections(void);
void BeginCriticalSection(void);
void EndCriticalSection(void);
```

Description

When called, the `EnableCriticalSections` subroutine enables the thread to be exempted from timeslicing and signal suspension. Once that is done, the thread can call the `BeginCriticalSection` and `EndCriticalSection` subroutines to protect critical sections. Calling the `BeginCriticalSection` and `EndCriticalSection` subroutines with exemption disabled has no effect. The subroutines are safe for use by multithreaded applications.

Once the service is enabled, the thread can protect critical sections by calling the `BeginCriticalSection` and `EndCriticalSection` subroutines. Calling the `BeginCriticalSection` subroutine will exempt the thread from timeslicing and suspension. Calling the `EndCriticalSection` subroutine will clear exemption for the thread.

The `BeginCriticalSection` subroutine will not make a system call. The `EndCriticalSection` subroutine might make a system call if the thread was granted a benefit during the critical section. The purpose of the system call would be to notify the kernel that any posted but undelivered stop signals can be delivered, and any postponed timeslice can now be completed.

Return Values

The `EnableCriticalSections` subroutine returns a zero.

**erf, erff, or erfl Subroutine**

**Purpose**

Computes the error and complementary error functions.

**Libraries**

IEEE Math Library (`libm.a`) or System V Math Library (`libmsaa.a`)

**Syntax**

```c
#include <math.h>

double erf (double x);
float erff (float x);
long double erfl (long double x);
```

**Description**

The `erf`, `erff`, and `erfl` subroutines return the error function of the `x` parameter, defined for the `erf` subroutine as the following:

\[
erf(x) = \frac{2}{\sqrt{\pi}} \int_{0}^{x} e^{-t^2} \, dt
\]

\[
erfc(x) = 1.0 - erf(x)
\]
**Note:** Compile any routine that uses subroutines from the `libm.a` library with the `-lm` flag. To compile the `erf.c` file, for example, enter:

```
cc erf.c -lm
```

An application wishing to check for error situations should set `errno` to zero and call `feclearexcept(FE_ALL_EXCEPT)` before calling these functions. Upon return, if `errno` is nonzero or `fetestexcept(FE_INVALID | FE_DIVBYZERO | FE_OVERFLOW | FE_UNDERFLOW)` is nonzero, an error has occurred.

**Parameters**

`x`  
Specifies a double-precision floating-point value.

**Return Values**

Upon successful completion, the `erf`, `erff`, and `erfl` subroutines return the value of the error function.

If `x` is NaN, a NaN is returned.

If `x` is ±0, ±0 is returned.

If `x` is ±Inf, ±1 is returned.

If `x` is subnormal, a range error may occur, and `2 * x/sqrt(pi)` should be returned.

**Related Information**

"erfc, erfcf, or erfcl Subroutine," "exp, expf, or expl Subroutine" on page 244, "feclearexcept Subroutine" on page 262, "fetestexcept Subroutine" on page 270, and "class, _class, finite, isnan, or unordered Subroutines" on page 167.

The `sqrt`, `sqrtf`, or `sqrtl` Subroutine in *AIX 5L Version 5.3 Technical Reference: Base Operating System and Extensions Volume 2*.

Subroutines Overview in *AIX 5L Version 5.3 General Programming Concepts: Writing and Debugging Programs*.

128-Bit long double Floating-Point Format in *AIX 5L Version 5.3 General Programming Concepts: Writing and Debugging Programs*.

`math.h` in *AIX 5L Version 5.3 Files Reference*.

**erfc, erfcf, or erfcl Subroutine**

**Purpose**

Computes the complementary error function.

**Syntax**

```c
#include <math.h>

float erf cf (x)
float x;
long double erf cl (x)
```
long double x;

double erfc (x)

double x;

Description
The erfc, erfcl, and erfc subroutines compute the complementary error function 1.0 - erf(x).

An application wishing to check for error situations should set errno to zero and call
feclearexcept(FE_ALL_EXCEPT) before calling these functions. Upon return, if errno is nonzero or
fetestexcept(FE_INVALID | FE_DIVBYZERO | FE_OVERFLOW | FE_UNDERFLOW) is nonzero, an error
has occurred.

Parameters
x Specifies the value to be computed.

Return Values
Upon successful completion, the erfc, erfcl, and erfc subroutines return the value of the complementary
error function.

If the correct value would cause underflow and is not representable, a range error may occur. Either 0.0 (if
representable), or an implementation-defined value is returned.

If x is NaN, a NaN is returned.

If x is ±0, +1 is returned.

If x is -Inf, +2 is returned.

If x is +Inf, +0 is returned.

If the correct value would cause underflow and is representable, a range error may occur and the correct
value is returned.

Related Information
“feclearexcept Subroutine” on page 262, “fetestexcept Subroutine” on page 270, and “class, _class, finite,
isnan, or unordered Subroutines” on page 167.

math.h in AIX 5L Version 5.3 Files Reference.

errlog Subroutine

Purpose
Logs an application error to the system error log.

Library
Run-Time Services Library (librts.a)
Syntax

```
#include <sys/errids.h>
int errlog (ErrorStructure, Length)
void *ErrorStructure;
unsigned int Length;
```

Description

The `errlog` subroutine writes an error log entry to the `/dev/error` file. The `errlog` subroutine is used by application programs.

The transfer from the `err_rec` structure to the error log is by a `write` subroutine to the `/dev/error` special file.

The `errdemon` process reads from the `/dev/error` file and writes the error log entry to the system error log. The timestamp, machine ID, node ID, and Software Vital Product Data associated with the resource name (if any) are added to the entry before going to the log.

Parameters

`ErrorStructure` Points to an error record structure containing an error record. Valid error record structures are typed in the `/usr/include/sys/err_rec.h` file. The two error record structures available are `err_rec` and `err_rec0`. The `err_rec` structure is used when the `detail_data` field is required. When the `detail_data` field is not required, the `err_rec0` structure is used.

```c
struct err_rec0 {
    unsigned int error_id;
    char resource_name[ERR_NAMESIZE];
};
struct err_rec {
    unsigned int error_id;
    char resource_name[ERR_NAMESIZE];
    char detail_data[1];
};
```

The fields of the structures `err_rec` and `err_rec0` are:

- `error_id` Specifies an index for the system error template database, and is assigned by the `errupdate` command when adding an error template. Use the `errupdate` command with the `-h` flag to get a `#define` statement for this 8-digit hexadecimal index.

- `resource_name` Specifies the name of the resource that has detected the error. For software errors, this is the name of a software component or an executable program. For hardware errors, this is the name of a device or system component. It does not indicate that the component is faulty or needs replacement instead, it is used to determine the appropriate diagnostic modules to be used to analyze the error.

- `detail_data` Specifies an array from 0 to `ERR_REC_MAX` bytes of user-supplied data. This data may be displayed by the `errpt` command in hexadecimal, alphanumeric, or binary form, according to the `data_encoding` fields in the error log template for this `error_id` field.

`Length` Specifies the length in bytes of the `err_rec` structure, which is equal to the size of the `error_id` and `resource_name` fields plus the length in bytes of the `detail_data` field.
Return Values

0   The entry was logged successfully.
-1   The entry was not logged.

Files

/dev/error Provides standard device driver interfaces required by the error log component.
/usr/include/sys/errids.h Contains definitions for error IDs.
/usr/include/sys/err_rec.h Contains structures defined as arguments to the errsave kernel service and the errlog subroutine.
/var/adm/ras/errlog Maintains the system error log.

Related Information

The errclear errdead errinstall errlogger errmsg errpt errstop and errupdate commands.
The errlog_open errlog_close errlog_find_first errlog_find_next errlog_find_sequence errlog_set_direction and errlog_write subroutines.
The /dev/error special file.
The errdemon daemon.
The errsave kernel service.

Error Logging Overview in Messages Guide and Reference.

errlog_close Subroutine

Purpose
Closes an open error log file.

Syntax
library liberrlog.a

#include <sys/errlog.h>

int errlog_close(handle)
errlog_handle_t handle;

Description
The error log specified by the handle argument is closed. The handle must have been returned from a previous errlog_open call.

Return Values
Upon successful completion, the errlog_close subroutine returns 0.

If an error occurs, the errlog_close subroutine returns LE_ERR_INVARG.
Related Information

The `errlog_open`, `errlog_find_first`, `errlog_find_next`, `errlog_find_sequence`, `errlog_set_direction`, `errlog_write`, and `errlog` subroutines.

**errlog_find_first, errlog_find_next, and errlog_find_sequence Subroutines**

**Purpose**
Retrieves an error log entry using supplied criteria.

**Syntax**

```c
#include <sys/errlog.h>

int errlog_find_first(handle, filter, result)
    errlog_handle_t handle;
    errlog_match_t *filter;
    errlog_entry_t *result;

int errlog_find_next(handle, result)
    errlog_handle_t handle;
    errlog_entry_t *result;

int errlog_find_sequence(handle, sequence, result)
    errlog_handle_t handle;
    int sequence;
    errlog_entry_t *result;
```

**Description**

The `errlog_find_first` subroutine finds the first occurrence of the search argument specified by filter using the direction specified by the `errlog_set_direction` subroutine. The reverse direction is used if none was specified. In other words, by default, entries are searched starting with the most recent entry.

The `errlog_match_t` structure, pointed to by the filter parameter, defines a test expression or set of expressions to be applied to each errlog entry.

If the value passed in the filter parameter is null, the `errlog_find_first` subroutine returns the first entry in the log, and the `errlog_find_next` subroutine can then be used to return subsequent entries. To read all log entries in the desired direction, open the log, then issue `errlog_find_next` calls.

To define a basic expression, `em_field` must be set to the field in the errlog entry to be tested, `em_op` must be set to the relational operator to be applied to that field, and either `em_intvalue` or `em_strvalue` must be set to the value to test against. Basic expressions may be combined by attaching them to `em_left` and `em_right` of another `errlog_match_t` structure and setting `em_op` of that structure to a binary or unary operator. These complex expressions may then be combined with other basic or complex expressions in the same fashion to build a tree that can define a filter of arbitrary complexity.

The `errlog_find_next` subroutine finds the next error log entry matching the criteria specified by a previous `errlog_find_first` call. The search continues in the direction specified by the `errlog_set_direction` subroutine or the reverse direction by default.

The `errlog_find_sequence` subroutine returns the entry matching the specified error log sequence number, found in the `el_sequence` field of the `errlog_entry` structure.
Parameters
The handle contains the handle returned by a prior call to errlog_open.

The filter parameter points to an `errlog_match_t` element defining the search argument, or the first of an argument tree.

The sequence parameter contains the sequence number of the entry to be retrieved.

The result parameter must point to the area to contain the returned error log entry.

Return Values
Upon successful completion, the `errlog_find_first`, `errlog_find_next`, and `errlog_find_sequence` subroutines return 0, and the memory referenced by result contains the found entry.

The following errors may be returned:

- `LE_ERR_INVARG` A parameter error was detected.
- `LE_ERR_NOMEM` Memory could not be allocated.
- `LE_ERR_IO` An i/o error occurred.
- `LE_ERR_DONE` No more entries were found.

Examples
The code below demonstrates how to search for all errlog entries in a date range and with a class of H (hardware) or S (software).

```c
{ extern int begintime, endtime;
  errlog_match_t beginstamp, endstamp, andstamp;
  errlog_match_t hardclass, softclass, orclass;
  errlog_match_t andtop;
  int ret;
  errlog_entry_t result;

  /* * Select begin and end times */
  */
  beginstamp.em_op = LE_OP_GT;
  beginstamp.em_field = LE_MATCH_TIMESTAMP;
  beginstamp.em_intvalue = begintime;

  endstamp.em_op = LE_OP_LT;
  endstamp.em_field = LE_MATCH_TIMESTAMP;
  endstamp.em_intvalue = endtime;

  andstamp.em_op = LE_OP_AND;
  andstamp.em_left = &beginstamp;
  andstamp.em_right = &endstamp;

  /* * Select the classes we're interested in. */
  */
  hardclass.em_op = LE_OP_EQ;
  hardclass.em_field = LE_MATCH_CLASS;
  hardclass.em_strvalue = "H";

  softclass.em_op = LE_OP_EQ;
  softclass.em_field = LE_MATCH_CLASS;
  softclass.em_strvalue = "S";

  orclass.em_op = LE_OP_OR;
  orclass.em_field = LE_MATCH_CLASS;
  orclass.em_strvalue = "H", "S";
```

232 Technical Reference, Volume 1: Base Operating System and Extensions
The `errlog_find_first` function will return the first entry matching filter. Successive calls to the `errlog_find_next` function will return successive entries that match the filter specified in the most recent call to the `errlog_find_first` function. When no more matching entries are found, the `errlog_find_first` and `errlog_find_next` functions will return the value `LE_ERR_DONE`.

**Related Information**

The `errlog_open`, `errlog_close`, `errlog_set_direction`, `errlog_write` and `errlog` subroutines.

---

### errlog_open Subroutine

**Purpose**

Opens an error log and returns a handle for use with other `liberrlog.a` functions.

**Syntax**

```c
#include <fcntl.h>
#include <sys/errlog.h>

int errlog_open(char *path, int mode, unsigned int magic, errlog_handle_t *handle)
```

**Description**

The error log specified by the path argument will be opened using mode. The handle pointed to by the handle parameter must be used with subsequent operations.

**Parameters**

The path parameter specifies the path to the log file to be opened. If path is NULL, the default errlog file will be opened. The valid values for mode are the same as they are for the open system subroutine. They can be found in the `fcntl.h` files.

The `magic` argument takes the `LE_MAGIC` value, indicating which version of the `errlog_entry_t` structure this application was compiled with.

**Return Values**

Upon successful completion, the `errlog_open` subroutine returns a 0 and sets the memory pointed to by handle to a handle used by subsequent `liberrlog` operations.

Upon error, the `errlog_open` subroutine returns one of the following:

- **LE_ERR_INVARG**: A parameter error was detected.
- **LE_ERR_NOFILE**: The log file does not exist.
LE_ERR_NOMEM  Memory could not be allocated.
LE_ERR_IO     An i/o error occurred.
LE_ERR_INVFILE The file is not a valid error log.

Related Information
The errlog_close, errlog_find_first, errlog_find_next, errlog_find_sequence, errlog_set_direction,
errlog_write, and errlog subroutines.

The /usr/include/fcntl.h include files found in AIX 5L Version 5.3 Files Reference.

errlog_set_direction Subroutine

Purpose
Sets the direction for the error log find functions.

Syntax
library liberrlog.a
#include <sys/errlog.h>

int errlog_set_direction(handle, direction)
  errlog_handle_t handle;
  int direction;

Description
The errlog_find_next and errlog_find_sequence subroutines search the error log starting with the most
recent log entry and going backward in time, by default. The errlog_set_direction subroutine is used to
alter this direction.

Parameters
The handle parameter must contain a handle returned by a previous errlog_open call.

The direction parameter must be LE_FORWARD or LE_REVERSE. LE_REVERSE is the default if the
errlog_set_direction subroutine is not used.

Return Values
Upon successful completion, the errlog_set_direction subroutine returns 0.

If a parameter is invalid, the errlog_set_direction subroutine returns LE_ERR_INVARG.

Related Information
The errlog_open, errlog_close, errlog_find_first, errlog_find_next, errlog_find_sequence,
errlog_write, and errlog subroutines.

errlog_write Subroutine

Purpose
Changes the previously read error log entry.
Syntax

#include <sys/errlog.h>

int errlog_write(handle, entry)
errlog_handle_t handle;
errlog_entry_t *entry;

Description

The `errlog_write` subroutine is used to update the most recently read log entry. Neither the length nor the sequence number of the entry may be changed. The entry is simply updated in place.

If the `errlog_write` subroutine is used in a multi-threaded application, the program should obtain a lock around the read/write pair to avoid conflict.

Parameters

The handle parameter must contain a handle returned by a previous `errlog_open` call.

The entry parameter must point to an entry returned by the previous error log find function.

Return Values

Upon successful completion, the `errlog_write` subroutine returns 0.

If a parameter is invalid, the `errlog_write` subroutine returns `LE_ERR_INVARG`.

The `errlog_write` subroutine may also return one of the following:

- `LE_ERR_INVFILE` The data on file is invalid.
- `LE_ERR_IO` An i/o error occurred.
- `LE_ERR_NOWRITE` The entry to be written didn’t match the entry being updated.

Related Information

The `errlog_open`, `errlog_close`, `errlog_find_first`, `errlog_find_next`, `errlog_find_sequence`, `errlog_set_direction`, and `errlog` subroutines.

The `/usr/include/sys/errlog.h` include file.

exec: execl, execle, execvp, execv, execve, execvp, or exec Subroutine

Purpose

Executes a file.

Library

Standard C Library (`libc.a`)

Syntax

#include <unistd.h>

extern char **environ;
int execl   
  (Path, 
   Argument0 [, Argument1 ...], 0) 
const char *Path, *Argument0, *Argument 
1, ...;

int execle (Path, 
   Argument0 [, Argument1 ...], 0, 
   EnvironmentPointer) 
const 
char *Path, *Argument0, *Argument 
ent 
1, ...; 
char *const EnvironmentPointer[ ];

int execlp (File, 
   Argument0 [, Argument1 
   , ...], 0) 
const char *File, *Argument0, *Argument 
1, ...;

int execv (Path, 
   ArgumentV) 
const char *Path; 
char *const ArgumentV[ ];

int execve (Path, 
   ArgumentV, 
   EnvironmentPointer) 
const char *Path; 
char 
*const ArgumentV[ ], *EnvironmentPointer 
[ ];

int execvp (File, 
   ArgumentV) 
const char *File; 
char *const ArgumentV[ ];

int exect (Path, 
   ArgumentV, 
   EnvironmentPointer) 
Description
The **exec** subroutine, in all its forms, executes a new program in the calling process. The **exec** subroutine does not create a new process, but overlays the current program with a new one, which is called the *new-process image*. The new-process image file can be one of three file types:

- An executable binary file in XCOFF file format.
- An executable text file that contains a shell procedure (only the **execpl** and **execvp** subroutines allow this type of new-process image file).
- A file that names an executable binary file or shell procedure to be run.

The new-process image inherits the following attributes from the calling process image: session membership, supplementary group IDs, process signal mask, and pending signals.

The last of the types mentioned is recognized by a header with the following syntax:

```
#!/ Path [String]
```

The `#!` is the file *magic number*, which identifies the file type. The path name of the file to be executed is specified by the *Path* parameter. The *String* parameter is an optional character string that contains no tab or space characters. If specified, this string is passed to the new process as an argument in front of the name of the new-process image file. The header must be terminated with a new-line character. When called, the new process passes the *Path* parameter as **ArgumentV[0]**. If a *String* parameter is specified in the new process image file, the **exec** subroutine sets **ArgumentV[0]** to the *String* and *Path* parameter values concatenated together. The rest of the arguments passed are the same as those passed to the **exec** subroutine.

The **exec** subroutine attempts to cancel outstanding asynchronous I/O requests by this process. If the asynchronous I/O requests cannot be canceled, the application is blocked until the requests have completed.

The **exec** subroutine is similar to the **load** (*load Subroutine* on page 721) subroutine, except that the **exec** subroutine does not have an explicit library path parameter. Instead, the **exec** subroutine uses either the LIBPATH or LD_LIBRARY_PATH environment variable. The LIBPATH variable, when set, is used in favor of LD_LIBRARY_PATH; otherwise, LD_LIBRARY_PATH is used. These library path variables are ignored when the program that the **exec** subroutine is run on has more privilege than the calling program (for example, an *suid* program).

The **exec** subroutine is included for compatibility with older programs being traced with the **ptrace** command. The program being executed is forced into hardware single-step mode.

**Note:** **exec** is not supported in 64-bit mode.

**Note:** Currently, a Graphics Library program cannot be overlaid with another Graphics Library program. The overlaying program can be a nongraphics program. For additional information, see the /usr/lpp/GL/README file.

Parameters

**Path**

Specifies a pointer to the path name of the new-process image file. If Network File System (NFS) is installed on your system, this path can cross into another node. Data is copied into local virtual memory before proceeding.
Specifies a pointer to the name of the new-process image file. Unless the File parameter is a full path name, the path prefix for the file is obtained by searching the directories named in the PATH environment variable. The initial environment is supplied by the shell.

**Note:** The execlp subroutine and the execvp subroutine take File parameters, but the rest of the exec subroutines take Path parameters. (For information about the environment, see the environment miscellaneous facility and the sh command.)

**Argument0 [ Argument1, ...]**
Point to null-terminated character strings. The strings constitute the argument list available to the new process. By convention, at least the Argument0 parameter must be present, and it must point to a string that is the same as the Path parameter or its last component.

**ArgumentV**
Specifies an array of pointers to null-terminated character strings. These strings constitute the argument list available to the new process. By convention, the ArgumentV parameter must have at least one element, and it must point to a string that is the same as the Path parameter or its last component. The last element of the ArgumentV parameter is a null pointer.

**EnvironmentPointer**
An array of pointers to null-terminated character strings. These strings constitute the environment for the new process. The last element of the EnvironmentPointer parameter is a null pointer.

When a C program is run, it receives the following parameters:

```c
main (ArgumentCount, ArgumentV, EnvironmentPointer)
int ArgumentCount;
char *ArgumentV[ ], *EnvironmentPointer[ ];
```

In this example, the ArgumentCount parameter is the argument count, and the ArgumentV parameter is an array of character pointers to the arguments themselves. By convention, the value of the ArgumentCount parameter is at least 1, and the ArgumentV[0] parameter points to a string containing the name of the new-process image file.

The main routine of a C language program automatically begins with a runtime start-off routine. This routine sets the environ global variable so that it points to the environment array passed to the program in EnvironmentPointer. You can access this global variable by including the following declaration in your program:

```c
extern char **environ;
```

The execl, execv, execlp, and execvp subroutines use the environ global variable to pass the calling process current environment to the new process.

File descriptors open in the calling process remain open, except for those whose close-on-exec flag is set. For those file descriptors that remain open, the file pointer is unchanged. (For information about file control, see the fcntl.h file.)

The state-of-conversion descriptors and message-catalog descriptors in the new process image are undefined. For the new process, an equivalent of the setlocale subroutine, specifying the LC_ALL value for its category and the "C" value for its locale, is run at startup.

If the new program requires shared libraries, the exec subroutine finds, opens, and loads each of them into the new-process address space. The referenced counts for shared libraries in use by the issuer of the exec are decremented. Shared libraries are searched for in the directories listed in the LIBPATH environment variable. If any of these files is remote, the data is copied into local virtual memory.
The `exec` subroutines reset all caught signals to the default action. Signals that cause the default action continue to do so after the `exec` subroutines. Ignored signals remain ignored, the signal mask remains the same, and the signal stack state is reset. (For information about signals, see the `sigaction` subroutine.)

If the `SetUserID` mode bit of the new-process image file is set, the `exec` subroutine sets the effective user ID of the new process to the owner ID of the new-process image file. Similarly, if the `SetGroupID` mode bit of the new-process image file is set, the effective group ID of the new process is set to the group ID of the new-process image file. The real user ID and real group ID of the new process remain the same as those of the calling process. (For information about the `SetID` modes, see the `chmod` subroutine.)

At the end of the `exec` operation the saved user ID and saved group ID of the process are always set to the effective user ID and effective group ID, respectively, of the process.

When one or both of the set ID mode bits is set and the file to be executed is a remote file, the file user and group IDs go through outbound translation at the server. Then they are transmitted to the client node where they are translated according to the inbound translation table. These translated IDs become the user and group IDs of the new process.

**Note:** `setuid` and `setgid` bids on shell scripts do not affect user or group IDs of the process finally executed.

Profiling is disabled for the new process.

The new process inherits the following attributes from the calling process:

- Nice value (see the `getpriority` subroutine, `setpriority` subroutine, `nice` subroutine)
- Process ID
- Parent process ID
- Process group ID
- `semadj` values (see the `semop` subroutine)
- tty group ID (see the `exit`, `atexit`, or `_exit` subroutine, `sigaction` subroutine)
- `trace` flag (see request 0 of the `ptrace` subroutine)
- Time left until an alarm clock signal (see the `incinterval` subroutine, `setitimer` subroutine, and `alarm` subroutine)
- Current directory
- Root directory
- File-mode creation mask (see the `umask` subroutine)
- File size limit (see the `ulimit` subroutine)
- Resource limits (see the `getrlimit` subroutine, `setrlimit` subroutine, and `vlimit` subroutine)
- `tms_utime`, `tms_stime`, `tms_cutime`, and `tms_ctime` fields of the `tms` structure (see the `times` subroutine)
- Login user ID

Upon successful completion, the `exec` subroutines mark for update the `st_atime` field of the file.

**Examples**

1. To run a command and pass it a parameter, enter:

   ```c
   exec1p("ls", "ls", "-al", 0);
   ```

   The `exec1p` subroutine searches each of the directories listed in the `PATH` environment variable for the `ls` command, and then it overlays the current process image with this command. The `exec1p` subroutine is not returned, unless the `ls` command cannot be executed.
Note: This example does not run the shell command processor, so operations interpreted by the shell, such as using wildcard characters in file names, are not valid.

2. To run the shell to interpret a command, enter:

```c
execl("/usr/bin/sh", "sh", "-c", "ls -l *.c", 0);
```

This runs the `sh` command with the `-c` flag, which indicates that the following parameter is the command to be interpreted. This example uses the `exec` subroutine instead of the `execvp` subroutine because the full path name `/usr/bin/sh` is specified, making a path search unnecessary.

Running a shell command in a child process is generally more useful than simply using the `exec` subroutine, as shown in this example. The simplest way to do this is to use the `system` subroutine.

3. The following is an example of a new-process file that names a program to be run:

```bash
#!/usr/bin/awk -f
{ for (i = NF; i > 0; --i) print $i }
```

If this file is named `reverse`, entering the following command on the command line:

```
reverse chapter1 chapter2
```

This command runs the following command:

```
/usr/bin/awk -f reverse chapter1 chapter2
```

Note: The `exec` subroutines use only the first line of the new-process image file and ignore the rest of it. Also, the `awk` command interprets the text that follows a `#` (pound sign) as a comment.

### Return Values

Upon successful completion, the `exec` subroutines do not return because the calling process image is overlaid by the new-process image. If the `exec` subroutines return to the calling process, the value of -1 is returned and the `errno` global variable is set to identify the error.

### Error Codes

If the `exec` subroutine is unsuccessful, it returns one or more of the following error codes:

- **EACCES**: The new-process image file is not an ordinary file.
- **EACCES**: The mode of the new-process image file denies execution permission.
- **ENOEXEC**: The `exec` subroutine is neither an `execvp` subroutine nor an `execvp` subroutine. The new-process image file has the appropriate access permission, but the magic number in its header is not valid.
- **ENOEXEC**: The new-process image file has a valid magic number in its header, but the header is damaged or is incorrect for the machine on which the file is to be run.
- **ETXTBSY**: The new-process image file is a pure procedure (shared text) file that is currently open for writing by some process.
- **ENOMEM**: The new process requires more memory than is allowed by the system-imposed maximum, the `MAXMEM` compiler option.
- **E2BIG**: The number of bytes in the new-process argument list is greater than the system-imposed limit. This limit is a system configurable value that can be set by superusers or system group users using SMIT. Refer to [Kernel Tunable Parameters](#) for details.
- **EFAULT**: The `Path`, `ArgumentV`, or `EnvironmentPointer` parameter points outside of the process address space.
- **EPERM**: The `SetUserID` or `SetGroupID` mode bit is set on the process image file. The translation tables at the server or client do not allow translation of this user or group ID.

If the `exec` subroutine is unsuccessful because of a condition requiring path name resolution, it returns one or more of the following error codes:
EACCES  Search permission is denied on a component of the path prefix. Access could be denied due to a secure mount.

EFAULT  The Path parameter points outside of the allocated address space of the process.

EIO  An input/output (I/O) error occurred during the operation.

ELOOP  Too many symbolic links were encountered in translating the Path parameter.

ENAMETOOLONG  A component of a path name exceeded 255 characters and the process has the disallow truncation attribute (see the ulimit subroutine), or an entire path name exceeded 1023 characters.

ENOENT  A component of the path prefix does not exist.

ENOENT  A symbolic link was named, but the file to which it refers does not exist.

ENOENT  The path name is null.

ENOTDIR  A component of the path prefix is not a directory.

ESTALE  The root or current directory of the process is located in a virtual file system that has been unmounted.

In addition, some errors can occur when using the new-process file after the old process image has been overwritten. These errors include problems in setting up new data and stack registers, problems in mapping a shared library, or problems in reading the new-process file. Because returning to the calling process is not possible, the system sends the SIGKILL signal to the process when one of these errors occurs.

If an error occurred while mapping a shared library, an error message describing the reason for error is written to standard error before the signal SIGKILL is sent to the process. If a shared library cannot be mapped, the subroutine returns one of the following error codes:

ENOENT  One or more components of the path name of the shared library file do not exist.

ENOTDIR  A component of the path prefix of the shared library file is not a directory.

ENAMETOOLONG  A component of a path name prefix of a shared library file exceeded 255 characters, or an entire path name exceeded 1023 characters.

EACCES  Search permission is denied for a directory listed in the path prefix of the shared library file.

EACCES  The shared library file mode denies execution permission.

ENOTEXEC  The shared library file has the appropriate access permission, but a magic number in its header is not valid.

ETXTBSY  The shared library file is currently open for writing by some other process.

ENOMEM  The shared library requires more memory than is allowed by the system-imposed maximum.

ESTALE  The process root or current directory is located in a virtual file system that has been unmounted.

EPROCLIM  If WLM is running, the limit on the number of processes, threads, or logins in the class may have been met.

If NFS is installed on the system, the exec subroutine can also fail if the following is true:

ETIMEDOUT  The connection timed out.

Related Information

The alarm (“getinterval, incinterval, absinterval, resinc, resabs, alarm, ualarm, gettimer or settimer subroutine” on page 382) or incinterval (“getinterval, incinterval, absinterval, resinc, resabs, alarm, ualarm, gettimer or settimer subroutine” on page 382) subroutine, chmod (“chmod or fchmod Subroutine” on page 148) or fchmod (“chmod or fchmod Subroutine” on page 148) subroutine, exit (“exit, atexit, unatexit, exit, or Exit Subroutine” on page 242) subroutine, fcntl (“fcntl, dup, or dup2 Subroutine” on page 254) subroutine, fork (“fork, fork, or vfork Subroutine” on page 287) subroutine, getusage (“getusage, getusage64, times, or vtimes Subroutine” on page 423) or times (“getusage, getusage64 subroutine” on page 423) subroutine.
exit, atexit, unatexit, _exit, or _Exit Subroutine

Purpose
Terminates a process.

Library
Standard C Library (libc.a)

Syntax
```c
#include <stdlib.h>

void exit (Status)
int Status;

void _exit (Status)
int Status;
void _Exit (Status)
int Status;
#include <sys/limits.h>

int atexit (Function)
void (*Function) (void);
int unatexit (Function)
void (*Function) (void);
```

Description
The `exit` subroutine terminates the calling process after calling the standard I/O library `_cleanup` function to flush any buffered output. Also, it calls any functions registered previously for the process by the `atexit` subroutine. The `atexit` subroutine registers functions called at normal process termination for cleanup processing. Normal termination occurs as a result of either a call to the `exit` subroutine or a `return` statement in the `main` function.
Each function a call to the **atexit** subroutine registers must return. This action ensures that all registered functions are called.

Finally, the **exit** subroutine calls the **_exit** subroutine, which completes process termination and does not return. The **_exit** subroutine terminates the calling process and causes the following to occur:

The **_Exit** subroutine is functionally equivalent to the **_exit** subroutine. The **_Exit** subroutine does not call functions registered with **atexit** or any registered signal handlers. The way the subroutine is implemented determines whether open streams are flushed or closed, and whether temporary files are removed. The calling process is terminated with the consequences described below.

- All of the file descriptors, directory streams, conversion descriptors, and message catalog descriptors open in the calling process are closed.
- If the parent process of the calling process is executing a **wait** or **waitpid**, and has not set its **SA_NOCLDWAIT** flag nor **SIGCHLD** to **SIG_IGN**, it is notified of the calling process’ termination and the low order eight bits (that is, bits 0377) of **status** are made available to it. If the parent is not waiting, the child’s status is made available to it when the parent subsequently executes **wait** or **waitpid**.
- If the parent process of the calling process is not executing a **wait** or **waitpid**, and has neither set its **SA_NOCLDWAIT** flag nor set **SIGCHLD** to **SIG_IGN**, the calling process is transformed into a zombie process. A zombie process is an inactive process that is deleted at some later time when its parent process executes **wait** or **waitpid**.
- Termination of a process does not directly terminate its children. The sending of a **SIGHUP** signal indirectly terminates children in some circumstances. This can be accomplished in one of two ways. If the implementation supports the **SIGCHLD** signal, a **SIGCHLD** is sent to the parent process. If the parent process has set its **SA_NOCLDWAIT** flag or set **SIGCHLD** to **SIG_IGN**, the calling process is transformed into a zombie process. A zombie process is an inactive process that is deleted at some later time when its parent process executes **wait** or **waitpid**.
- The parent process ID of all of the calling process’ existing child processes and zombie processes are set to the process ID of an implementation defined system process.
- Each attached shared memory segment is detached and the value of **shm_nattch** (see **shmget**) in the data structure associated with its shared memory ID is decremented by 1.
- For each semaphore for which the calling process has set a **semadj** value (see **semop**), that value is added to the **semval** of the specified semaphore.
- If the process is a controlling process, the **SIGHUP** signal is sent to each process in the foreground process group of the controlling terminal belonging to the calling process.
- If the process is a controlling process, the controlling terminal associated with the session is disassociated from the session, allowing it to be acquired by a new controlling process.
- If the exit of the process causes a process group to become orphaned, and if any member of the newly orphaned process group is stopped, a **SIGHUP** signal followed by a **SIGCONT** signal is sent to each process in the newly orphaned process group.
- All open named semaphores in the calling process are closed as if by appropriate calls to **sem_close**.
- Memory mappings that were created in the process are unmapped before the process is destroyed.
- Any blocks of typed memory that were mapped in the calling process are unmapped, as if the **munmap** subroutine was implicitly called to unmap them.
- All open message queue descriptors in the calling process are closed.
- Any outstanding cancelable asynchronous I/O operations may be canceled. Those asynchronous I/O operations that are not canceled complete as if the **_Exit** subroutine had not yet occurred, but any associated signal notifications are suppressed.
- The **_Exit** subroutine may block awaiting such I/O completion. The implementation defines whether any I/O is canceled, and which I/O may be canceled upon **_Exit**.
- Threads terminated by a call to **_Exit** do not invoke their cancelation cleanup handlers or per thread data destructors.
• If the calling process is a trace controller process, any trace streams that were created by the calling process are shut down.

The unatexit() subroutine is used to unregister functions that were previously registered by the atexit() subroutine. If the referenced function is found, it is removed from the list of functions that are called at normal program termination.

Parameters

Status Indicates the status of the process. May be set to 0, EXIT_SUCCESS, EXIT_FAILURE, or any other value, though only the least significant 8 bits are available to a waiting parent process.

Function Specifies a function to be called at normal process termination for cleanup processing. You may specify a number of functions to the limit set by the AEXIT_MAX function, which is defined in the sys/limits.h file. A pushdown stack of functions is kept so that the last function registered is the first function called.

Return Values

Upon successful completion, the atexit subroutine returns a value of 0. Otherwise, a nonzero value is returned. The exit and _exit subroutines do not return a value.

The unatexit() subroutine returns a value of 0 if the function referenced by Function is found and removed from the atexit list. Otherwise, a non-zero value is returned.

Related Information

“acct Subroutine” on page 7, “lockfx, lockf, flock, or lockf64 Subroutine” on page 733, “lockfx, lockf, flock, or lockf64 Subroutine” on page 733, and “getrusage, getrusage64, times, or vtimes Subroutine” on page 423.

longjmp Subroutine, semop Subroutine, shmget Subroutine, sigaction, sigvec, or signal Subroutine, and wait, waitpid, or wait3 Subroutine in AIX 5L Version 5.3 Technical Reference: Base Operating System and Extensions Volume 2.


unistd.h in AIX 5L Version 5.3 Files Reference.

exp, expf, or expl Subroutine

Purpose

Computes exponential, logarithm, and power functions.

Libraries

IEEE Math Library (libm.a)
or System V Math Library (libmsaa.a)

Syntax

#include <math.h>

double exp (x)

double x;

exp, expf, or expl Subroutine

Purpose

Computes exponential, logarithm, and power functions.

Libraries

IEEE Math Library (libm.a)
or System V Math Library (libmsaa.a)

Syntax

#include <math.h>

double exp (x)

double x;
float expf (x)
float x;
long double expl (x)
long double x;

Description
These subroutines are used to compute exponential, logarithm, and power functions.

The exp, expf, and expl subroutines returns \( e^x \).

An application wishing to check for error situations should set the errno global variable to zero and call 
\texttt{fetastexcet}(FE\_ALL\_EXCEPT) before calling these subroutines. Upon return, if errno is nonzero or 
\texttt{fetestexcet}(FE\_INVALID \textbar FE\_DIVBYZERO \textbar FE\_OVERFLOW \textbar FE\_UNDERFLOW) is nonzero, an error 
has occurred.

Parameters
\( x \) Specifies some double-precision floating-point value.
\( y \) Specifies some double-precision floating-point value.

Return Values
Upon successful completion, the exp, expf, and expl subroutines return the exponential value of \( x \).

If the correct value would cause overflow, a range error occurs and the exp, expf, and expl subroutine 
returns the value of the macro \texttt{HUGE\_VAL}, \texttt{HUGE\_VALF} and \texttt{HUGE\_VALL}, respectively.

If the correct value would cause underflow, and is not representable, a range error may occur, and either 
0.0 (if supported), or an implementation-defined value is returned.

If \( x \) is NaN, a NaN is returned.

If \( x \) is ±0, 1 is returned.

If \( x \) is -Inf, +0 is returned.

If \( x \) is +Inf, \( x \) is returned.

If the correct value would cause underflow, and is representable, a range error may occur and the correct 
value is returned.

Error Codes
When using the \texttt{libm.a} library:

\( \texttt{exp} \) If the correct value would overflow, the \texttt{exp} subroutine returns a \texttt{HUGE\_VAL} value and the \texttt{errno} 
global variable is set to a \texttt{ERANGE} value.

When using \texttt{libmsaa.a(-lmsaa)}:

\( \texttt{exp} \) If the correct value would overflow, the \texttt{exp} subroutine returns a \texttt{HUGE\_VAL} value. If the correct 
value would underflow, the \texttt{exp} subroutine returns 0. In both cases \texttt{errno} is set to \texttt{ERANGE}.

\( \texttt{expl} \) If the correct value would overflow, the \texttt{expl} subroutine returns a \texttt{HUGE\_VAL} value. If the correct 
value would underflow, the \texttt{expl} subroutine returns 0. In both cases \texttt{errno} is set to \texttt{ERANGE}.
If the correct value overflows, the `expi` subroutine returns a `HUGE_VAL` value and `errno` is set to `ERANGE`.

These error-handling procedures may be changed with the `matherr` subroutine when using the `libmsaa.a` library.

Related Information

- “fustect Subroutine” on page 262, “fetestexcept Subroutine” on page 270, and “class, _class, finite, isnan, or unordered Subroutines” on page 167.

The `matherr` (matherr Subroutine” on page 780 subroutine, `sinh`, `cosh`, or `tanh` subroutine.

Subroutines Overview in AIX 5L Version 5.3 General Programming Concepts: Writing and Debugging Programs.

128-Bit long double Floating-Point Format in AIX 5L Version 5.3 General Programming Concepts: Writing and Debugging Programs.

`math.h` in AIX 5L Version 5.3 Files Reference.

**exp2, exp2f, or exp2l Subroutine**

**Purpose**
Computes the base 2 exponential.

**Syntax**

```c
#include <math.h>

double exp2 (x)

double x;

float exp2f (x)

float x;

long double exp2l (x)

long double x;
```

**Description**
The `exp2`, `exp2f`, and `exp2l` subroutines compute the base 2 exponential of the `x` parameter.

An application wishing to check for error situations should set the `errno` global variable to zero and call `feclearexcept (FE_ALL_EXCEPT)` before calling these subroutines. On return, if `errno` is nonzero or `fetestexcept (FE_INVALID | FE_DIVBYZERO | FE_OVERFLOW | FE_UNDERFLOW)` is nonzero, an error has occurred.

**Parameters**

- `x` Specifies the base 2 exponential to be computed.

**Return Values**
Upon successful completion, the `exp2`, `exp2f`, or `exp2l` subroutine returns $2^x$. 
If the correct value causes overflow, a range error occurs and the \texttt{exp2}, \texttt{exp2f}, and \texttt{exp2l} subroutines return the value of the macro (\texttt{HUGE\_VAL}, \texttt{HUGE\_VALF}, and \texttt{HUGE\_VALL}, respectively).

If the correct value causes underflow and is not representable, a range error occurs, and 0.0 is returned.

If \( x \) is \( \text{NaN} \), \( \text{NaN} \) is returned.

If \( x \) is \( \pm 0 \), 1 is returned.

If \( x \) is \( -\infty \), 0 is returned.

If \( x \) is \( +\infty \), \( x \) is returned.

If the correct value would cause underflow, and is representable, a range error may occur and the correct value is returned.

\textbf{Related Information}

\texttt{math.h} in AIX 5L Version 5.3 Files Reference.

\texttt{expm1}, \texttt{expm1f}, or \texttt{expm1l} Subroutine

\textbf{Purpose}
Computes exponential functions.

\textbf{Syntax}

\begin{verbatim}
#include <math.h>

float expm1f (x)
float x;

long double expm1l (x)
long double x;

double expm1l (x)
double x;
\end{verbatim}

\textbf{Description}
The \texttt{expm1f}, \texttt{expm1l}, and \texttt{expm1} subroutines compute \( e^x - 1.0 \).

An application wishing to check for error situations should set the \texttt{errno} global variable to zero and call \texttt{fclearexcept(FE\_ALL\_EXCEPT)} before calling these functions. Upon return, if \texttt{errno} is nonzero or \texttt{fetestexcept(FE\_INVALID | FE\_DIVBYZERO | FE\_OVERFLOW | FE\_UNDERFLOW)} is nonzero, an error has occurred.

\textbf{Parameters}

\texttt{x} \hspace{1cm} \text{Specifies the value to be computed.}

\textbf{Return Values}

Upon successful completion, the \texttt{expm1f}, \texttt{expm1l}, and \texttt{expm1} subroutines return \( e^x - 1.0 \).

If the correct value would cause overflow, a range error occurs and the \texttt{expm1f}, \texttt{expm1l}, and \texttt{expm1} subroutines return the value of the macro \texttt{HUGE\_VALF}, \texttt{HUGE\_VALL}, and \texttt{HUGE\_VAL}, respectively.
If \( x \) is NaN, a NaN is returned.

If \( x \) is ±0, ±0 is returned.

If \( x \) is -Inf, -1 is returned.

If \( x \) is +Inf, \( x \) is returned.

If \( x \) is subnormal, a range error may occur and \( x \) is returned.

Related Information
- “exp, expf, or expl Subroutine” on page 244
- “fclearexcept Subroutine” on page 262
- “fetestexcept Subroutine” on page 270
- “ilogbf, ilogbl, or ilogb Subroutine” on page 529
- “log, logf, or logl Subroutine” on page 740

math.h in AIX 5L Version 5.3 Files Reference.

fabsf, fabsl, or fabs Subroutine

Purpose
Determines the absolute value.

Syntax
```c
#include <math.h>

float fabsf (x)
float x;

long double fabsl (x)
long double x;

double fabs (x)
double x;
```

Description
The \texttt{fabsf}, \texttt{fabsl}, and \texttt{fabs} subroutines compute the absolute value of the \( x \) parameter, \(|x|\).

Parameters
\( x \)
Specifies the value to be computed.

Return Values
Upon successful completion, the \texttt{fabsf}, \texttt{fabsl}, and \texttt{fabs} subroutines return the absolute value of \( x \).

If \( x \) is NaN, a NaN is returned.

If \( x \) is ±0, ±0 is returned.

If \( x \) is ±Inf, ±Inf is returned.
Related Information
The “class, _class, finite, isnan, or unordered Subroutines” on page 167.

math.h in AIX 5L Version 5.3 Files Reference.

fattach Subroutine

Purpose
Attaches a STREAMS-based file descriptor to a file.

Library
Standard C Library (libc.a)

Syntax
#include <stropts.h>
int fattach(int fildes, const char *path);

Description
The fattach subroutine attaches a STREAMS-based file descriptor to a file, effectively associating a pathname with fildes. The fildes argument must be a valid open file descriptor associated with a STREAMS file. The path argument points to a pathname of an existing file. The process must have appropriate privileges, or must be the owner of the file named by path and have write permission. A successful call to fattach subroutine causes all pathnames that name the file named by path to name the STREAMS file associated with fildes, until the STEAMS file is detached from the file. A STREAMS file can be attached to more than one file and can have several pathnames associated with it.

The attributes of the named STREAMS file are initialized as follows: the permissions, user ID, group ID, and times are set to those of the file named by path, the number of links is set to 1, and the size and device identifier are set to those of the STREAMS file associated with fildes. If any attributes of the named STREAMS file are subsequently changed (for example, by chmod subroutine), neither the attributes of the underlying file nor the attributes of the STREAMS file to which fildes refers are affected.

File descriptors referring to the underlying file, opened prior to an fattach subroutine, continue to refer to the underlying file.

Parameters

fildes A file descriptor identifying an open STREAMS-based object.
path An existing pathname which will be associated with fildes.

Return Value

0 Successful completion.
-1 Not successful and errno set to one of the following.

Errno Value

EACCES Search permission is denied for a component of the path prefix, or the process is the owner of path but does not have write permission on the file named by path.
EBADF The file referred to by fildes is not an open file descriptor.
ENOENT A component of path does not name an existing file or path is an empty string.
ENOTDIR A component of the path prefix is not a directory.
EPERM The effective user ID of the process is not the owner of the file named by path and the process does not have appropriate privilege.
EBUSY The file named by path is currently a mount point or has a STREAMS file attached to it.
ENAMETOOLONG The size of path exceeds \{PATH_MAX\}, or a component of path is longer than \{NAME_MAX\}.
ELOOP Too many symbolic links were encountered in resolving path.
EINVAL The fildes argument does not refer to a STREAMS file.
ENOMEM Insufficient storage space is available.

Related Specifics
The fdetach ("fdetach Subroutine" on page 260) subroutine, isastream subroutine.

fchdir Subroutine

Purpose
Directory pointed to by the file descriptor becomes the current working directory.

Library
Standard C Library (libc.a)

Syntax
#include <unistd.h>

int fchdir (int Fildes)

Description
The fchdir subroutine causes the directory specified by the Fildes parameter to become the current working directory.

Parameter
Fildes A file descriptor identifying an open directory obtained from a call to the open subroutine.

Return Values
0 Successful completion
-1 Not successful and errno set to one of the following.

Error Codes
EACCES Search access if denied.
EBADF The file referred to by Fildes is not an open file descriptor.
ENOTDIR The open file descriptor does not refer to a directory.

Related Information
The chdir ("chdir Subroutine" on page 147) subroutine, chroot ("chroot Subroutine" on page 160) subroutine, open ("open, openx, open64, creat, or creat64 Subroutine" on page 925) subroutine.
fclear or fclear64 Subroutine

Purpose
Makes a hole in a file.

Library
Standard C Library (libc.a)

Syntax
off_t fclear (FileDescriptor, NumberOfBytes)
int FileDescriptor;
off_t NumberOfBytes;

off64_t fclear64 (FileDescriptor, NumberOfBytes)
int FileDescriptor;
off64_t NumberOfBytes;

Description
The fclear and fclear64 subroutines zero the number of bytes specified by the NumberOfBytes parameter starting at the current file pointer for the file specified in the FileDescriptor parameter. If Network File System (NFS) is installed on your system, this file can reside on another node.

The fclear subroutine can only clear up to OFF_MAX bytes of the file while fclear64 can clear up to the maximum file size.

The fclear and fclear64 subroutines cannot be applied to a file that a process has opened with the O_DEFER mode.

Successful completion of the fclear and fclear64 subroutines clear the SetUserID bit (S_ISUID) of the file if any of the following are true:
• The calling process does not have root user authority.
• The effective user ID of the calling process does not match the user ID of the file.
• The file is executable by the group (S_IXGRP) or others (S_IXOTH).

This subroutine also clears the SetGroupID bit (S_ISGID) if:
• The file does not match the effective group ID or one of the supplementary group IDs of the process, OR
• The file is executable by the owner (S_IXUSR) or others (S_IXOTH).

Note: Clearing of the SetUserID and SetGroupID bits can occur even if the subroutine fails because the data in the file was modified before the error was detected.

In the large file enabled programming environment, fclear is redefined to be fclear64.

Parameters
FileDescriptor Indicates the file specified by the FileDescriptor parameter must be open for writing. The FileDescriptor is a small positive integer used instead of the file name to identify a file. This function differs from the logically equivalent write operation in that it returns full blocks of binary zeros to the file system, constructing holes in the file.
‘NumberOfBytes’ indicates the number of bytes that the seek pointer is advanced. If you use the ‘fclear’ and ‘fclear64’ subroutines past the end of a file, the rest of the file is cleared and the seek pointer is advanced by ‘NumberOfBytes’. The file size is updated to include this new hole, which leaves the current file position at the byte immediately beyond the new end-of-file pointer.

Return Values
Upon successful completion, a value of ‘NumberOfBytes’ is returned. Otherwise, a value of -1 is returned and the ‘errno’ global variable is set to indicate the error.

Error Codes
The ‘fclear’ and ‘fclear64’ subroutines fail if one or more of the following are true:

- **EIO** I/O error.
- **EBADF** The ‘FileDescriptor’ value is not a valid file descriptor open for writing.
- **EINVAL** The file is not a regular file.
- **EMFILE** The file is mapped **O_DEFER** by one or more processes.
- **EAGAIN** The write operation in the ‘fclear’ subroutine failed due to an enforced write lock on the file.

- **EFBIG** The current offset plus ‘NumberOfBytes’ is exceeds the offset maximum established in the open file description associated with ‘FileDescriptor’.

- **ETIMEDOUT** The connection timed out.

If NFS is installed on the system the ‘fclear’ and ‘fclear64’ subroutines can also fail if the following is true:

**Related Information**
The ‘open’, ‘openx’, or ‘creat’ subroutine, ‘truncate’ or ‘ftruncate’ subroutines.

Files, Directories, and File Systems for Programmers in **AIX 5L Version 5.3 General Programming Concepts: Writing and Debugging Programs**.

---

**fclose** or **fflush** Subroutine

**Purpose**
Closes or flushes a stream.

**Library**
Standard C Library (**libc.a**)

**Syntax**

```c
#include <stdio.h>
```
int fclose (Stream)
FILE *Stream;

int fflush (Stream)
FILE *Stream;

Description
The fclose subroutine writes buffered data to the stream specified by the Stream parameter, and then closes the stream. The fclose subroutine is automatically called for all open files when the exit subroutine is invoked.

The fflush subroutine writes any buffered data for the stream specified by the Stream parameter and leaves the stream open. The fflush subroutine marks the st_ctime and st_mtime fields of the underlying file for update.

If the Stream parameter is a null pointer, the fflush subroutine performs this flushing action on all streams for which the behavior is defined.

Parameters
Stream Specifies the output stream.

Return Values
Upon successful completion, the fclose and fflush subroutines return a value of 0. Otherwise, a value of EOF is returned.

Error Codes
If the fclose and fflush subroutines are unsuccessful, the following errors are returned through the errno global variable:

EAGAIN The O_NONBLOCK or O_NDELAY flag is set for the file descriptor underlying the Stream parameter and the process would be delayed in the write operation.
EBADF The file descriptor underlying Stream is not valid.
EFBIG An attempt was made to write a file that exceeds the process’ file size limit or the maximum file size. See the ulimit subroutine.
EFBIG The file is a regular file and an attempt was made to write at or beyond the offset maximum associated with the corresponding stream.
EINTR The fflush subroutine was interrupted by a signal.
EIO The process is a member of a background process group attempting to write to its controlling terminal, the TOSTOP signal is set, the process is neither ignoring nor blocking the SIGTTOU signal and the process group of the process is orphaned. This error may also be returned under implementation-dependent conditions.
ENOSPC No free space remained on the device containing the file.
EPIPE An attempt is made to write to a pipe or FIFO that is not open for reading by any process. A SIGPIPE signal is sent to the process.
ENXIO A request was made of a non-existent device, or the request was outside the capabilities of the device

Related Information
The close subroutine, exit, atexit, or _exit subroutine, fopen, freopen, or fdopen subroutine.
fcntl, dup, or dup2 Subroutine

Purpose
Controls open file descriptors.

Library
Standard C Library (libc.a)

Syntax
#include <fcntl.h>

int fcntl(int FileDescriptor, int Command, int Argument);

#include <unistd.h>

int dup2(int Old, int New);

int dup(int FileDescriptor);

Description
The fcntl subroutine performs controlling operations on the open file specified by the FileDescriptor parameter. If Network File System (NFS) is installed on your system, the open file can reside on another node. The fcntl subroutine is used to:

- Duplicate open file descriptors.
- Set and get the file-descriptor flags.
- Set and get the file-status flags.
- Manage record locks.
- Manage asynchronous I/O ownership.
- Close multiple files.

The fcntl subroutine can provide the same functions as the dup and dup2 subroutines.

If FileDescriptor refers to a terminal device or socket, then asynchronous I/O facilities can be used. These facilities are normally enabled by using the ioctl subroutine with the FIOASYNC, FIOSETOWN, and FIOGETOWN commands. However, a BSD-compatible mechanism is also available if the application is linked with the libbsd.a library.

When the FileDescriptor parameter refers to a shared memory object, the fcntl subroutine manages only the F_DUPFD, F_DUP2FD, F_GETFD, F_SETFD, F_GETFL, and F_CLOSEM commands.

When using the libbsd.a library, asynchronous I/O is enabled by using the F_SETFL command with the FASYNC flag set in the Argument parameter. The F_GETOWN and F_SETOWN commands get the current asynchronous I/O owner and set the asynchronous I/O owner.
All applications containing the \texttt{fcntl} subroutine must be complied with \_\texttt{BSD} set to a specific value. Acceptable values are 43 and 44. In addition, all socket applications must include the BSD libbsd.a library.

**General Record Locking Information**

A lock is either an \textit{enforced} or \textit{advisory lock} and either a \textit{read} or a \textit{write lock}.

\textbf{Attention:} Buffered I/O does not work properly when used with file locking. Do not use the standard I/O package routines on files that are going to be locked.

For a lock to be an enforced lock, the Enforced Locking attribute of the file must be set; for example, the \texttt{S\_ENFMT} bit must be set, but the \texttt{S\_IXGRP}, \texttt{S\_IXUSR}, and \texttt{S\_IXOTH} bits must be clear. Otherwise, the lock is an advisory lock. A given file can have advisory or enforced locks, but not both. The description of the \texttt{sys/mode.h} file includes a description of file attributes.

When a process holds an enforced lock on a section of a file, no other process can access that section of the file with the \texttt{read} or \texttt{write} subroutine. In addition, the \texttt{open} (\texttt{open, openx, open64, creat, or creat64 Subroutine} on page 925) and \texttt{ftruncate} subroutines cannot truncate the locked section of the file, and the \texttt{fclear} (\texttt{fclear or fclear64 Subroutine} on page 251) subroutine cannot modify the locked section of the file. If another process attempts to read or modify the locked section of the file, the process either sleeps until the section is unlocked or returns with an error indication.

When a process holds an advisory lock on a section of a file, no other process can lock that section of the file (or an overlapping section) with the \texttt{fcntl} subroutine. (No other subroutines are affected.) As a result, processes must voluntarily call the \texttt{fcntl} subroutine in order to make advisory locks effective.

When a process holds a read lock on a section of a file, other processes can also set read locks on that section or on subsets of it. Read locks are also called \textit{shared} locks.

A read lock prevents any other process from setting a write lock on any part of the protected area. If the read lock is also an enforced lock, no other process can modify the protected area.

The file descriptor on which a read lock is being placed must have been opened with read access.

When a process holds a write lock on a section of a file, no other process can set a read lock or a write lock on that section. Write locks are also called \textit{exclusive} locks. Only one write lock and no read locks can exist for a specific section of a file at any time.

If the lock is also an enforced lock, no other process can read or modify the protected area.

The following general rules about file locking apply:

- Changing or unlocking part of a file in the middle of a locked section leaves two smaller sections locked at each end of the originally locked section.
- If the calling process holds a lock on a file, that lock can be replaced by later calls to the \texttt{fcntl} subroutine.
- All locks associated with a file for a given process are removed when the process closes any file descriptor for that file.
- Locks are not inherited by a child process after a \texttt{fork} (\texttt{fork, f\_fork, or vfork Subroutine} on page 287) subroutine is run.

\textbf{Note:} Deadlocks due to file locks in a distributed system are not always detected. When such deadlocks can possibly occur, the programs requesting the locks should set time-out timers.

Locks can start and extend beyond the current end of a file but cannot be negative relative to the beginning of the file. A lock can be set to extend to the end of the file by setting the \_\_len field to 0. If
such a lock also has the l_start and l_whence fields set to 0, the whole file is locked. The l_len, l_start, and l_whence locking fields are part of the flock structure.

**Note:** The following description applies to AIX 4.3 and later releases.

When an application locks a region of a file using the 32 bit locking interface (F_SETLK), and the last byte of the lock range includes MAX_OFF (2 Gb - 1), then the lock range for the unlock request will be extended to include MAX_END (2 ^ ^ 63 - 1).

**Parameters**

**FileDescriptor**

Specifies an open file descriptor obtained from a successful call to the open subroutine, fcntl subroutine, pipe subroutine, or shm_open subroutine. File descriptors are small positive integers used (instead of file names) to identify files or a shared memory object.

**Argument**

Specifies a variable whose value sets the function specified by the Command parameter. When dealing with file locks, the Argument parameter must be a pointer to the FLOCK structure.

**Command**

Specifies the operation performed by the fcntl subroutine. The fcntl subroutine can duplicate open file descriptors, set file-descriptor flags, set file descriptor locks, set process IDs and close open file descriptors.

**Duplicating File Descriptors**

- **F_DUPFD**
  - Returns a new file descriptor as follows:
    - Lowest-numbered available file descriptor greater than or equal to the Argument parameter
    - Same object references as the original file
    - Same file pointer as the original file (that is, both file descriptors share one file pointer if the object is a file)
    - Same access mode (read, write, or read-write)
    - Same file status flags (That is, both file descriptors share the same file status flags.)
    - The close-on-exec flag (FD_CLOEXEC bit) associated with the new file descriptor is cleared

**Setting File-Descriptor Flags**

- **F_GETFD**
  - Gets the close-on-exec flag (FD_CLOEXEC bit) that is associated with the file descriptor specified by the FileDescriptor parameter. The Argument parameter is ignored. File descriptor flags are associated with a single file descriptor, and do not affect others associated with the same file.

- **F_SETFD**
  - Assigns the value of the Argument parameter to the close-on-exec flag (FD_CLOEXEC bit) that is associated with the FileDescriptor parameter. If the FD_CLOEXEC flag value is 0, the file remains open across any calls to exec subroutines; otherwise, the file will close upon the successful execution of an exec subroutine.
**F_GETFL**  
Gets the file-status flags and file-access modes for the open file description associated with the file descriptor specified by the `FileDescriptor` parameter. The open file description is set at the time the file is opened and applies only to those file descriptors associated with that particular call to the file. This open file descriptor does not affect other file descriptors that refer to the same file with different open file descriptions.

The file-status flags have the following values:

- **O_APPEND**
  - Set append mode.

- **O_NONBLOCK**
  - No delay.

The file-access modes have the following values:

- **O_RDONLY**
  - Open for reading only.

- **O_RDWR**
  - Open for reading and writing.

- **O_WRONLY**
  - Open for writing only.

The file access flags can be extracted from the return value using the `O_ACCMODE` mask, which is defined in the `fcntl.h` file.

**F_SETFL**

Sets the file status flags from the corresponding bits specified by the `Argument` parameter. The file-status flags are set for the open file description associated with the file descriptor specified by the `FileDescriptor` parameter. The following flags may be set:

- **O_APPEND** or `FAPPEND`
- **O_NDELAY** or `FNDELAY`
- **O_NONBLOCK** or `FNONBLOCK`
- **O_SYNC** or `FSYNC`
- **FASYNC**

The `O_NDELAY` and `O_NONBLOCK` flags affect only operations against file descriptors derived from the same `open` subroutine. In BSD, these operations apply to all file descriptors that refer to the object.

### Setting File Locks

**F_GETLK**

Gets information on the first lock that blocks the lock described in the `flock` structure. The `Argument` parameter should be a pointer to a type `struct flock`, as defined in the `flock.h` file. The information retrieved by the `fcntl` subroutine overwrites the information in the `struct flock` pointed to by the `Argument` parameter. If no lock is found that would prevent this lock from being created, the structure is left unchanged, except for lock type (`l_type`) which is set to `F_UNLCK`.

**F_SETLK**

Sets or clears a file-segment lock according to the lock description pointed to by the `Argument` parameter. The `Argument` parameter should be a pointer to a type `struct flock`, which is defined in the `flock.h` file. The `F_SETLK` option is used to establish (or shared) locks (`F_RDLCK`), or write (or exclusive) locks (`F_WRLCK`), as well as to remove either type of lock (`F_UNLCK`). The lock types are defined by the `fcntl.h` file. If a shared or exclusive lock cannot be set, the `fcntl` subroutine returns immediately.

**F_SETLKW**

Performs the same function as the `F_SETLK` option unless a read or write lock is blocked by existing locks, in which case the process sleeps until the section of the file is free to be locked. If a signal that is to be caught is received while the `fcntl` subroutine is waiting for a region, the `fcntl` subroutine is interrupted, returns a -1, sets the `errno` global variable to `EINTR`. The lock operation is not done.
F_GETLK64 Gets information on the first lock that blocks the lock described in the flock64 structure. The Argument parameter should be a pointer to an object of the type struct flock64, as defined in the flock.h file. The information retrieved by the fcntl subroutine overwrites the information in the struct flock64 pointed to by the Argument parameter. If no lock is found that would prevent this lock from being created, the structure is left unchanged, except for lock type (l_type) which is set to F_UNLCK.

F_SETLK64 Sets or clears a file-segment lock according to the lock description pointed to by the Argument parameter. The Argument parameter should be a pointer to a type struct flock64, which is defined in the flock.h file. The F_SETLK option is used to establish read (or shared) locks (F_RDLCK), or write (or exclusive) locks (F_WRLCK), as well as to remove either type of lock (F_UNLCK). The lock types are defined by the fcntl.h file. If a shared or exclusive lock cannot be set, the fcntl subroutine returns immediately.

F_SETLKW64 Performs the same function as the F_SETLK option unless a read or write lock is blocked by existing locks, in which case the process sleeps until the section of the file is free to be locked. If a signal that is to be caught is received while the fcntl subroutine is waiting for a region, the fcntl subroutine is interrupted, returns a -1, sets the errno global variable to EINTR. The lock operation is not done.

Setting Process ID

F_GETOWN Gets the process ID or process group currently receiving SIGIO and SIGURG signals. Process groups are returned as negative values.

F_SETOWN Sets the process or process group to receive SIGIO and SIGURG signals. Process groups are specified by supplying a negative Argument value. Otherwise, the Argument parameter is interpreted as a process ID.

Closing File Descriptors

F_CLOSEM Closes all file descriptors from FileDescriptor up to the number specified by the OPEN_MAX value.

Old Specifies an open file descriptor.

New Specifies an open file descriptor that is returned by the dup2 subroutine.

Compatibility Interfaces

The lockfx Subroutine
The fcntl subroutine functions similar to the lockfx subroutine, when the Command parameter is F_SETLK, F_SETLKW, or F_GETLK, and when used in the following way:

fcntl (FileDescriptor, Command, Argument) is equivalent to:

lockfx (FileDescriptor, Command, Argument)

The dup and dup2 Subroutines
The fcntl subroutine functions similar to the dup and dup2 subroutines, when used in the following way:

dup (FileDescriptor) is equivalent to:

cntl (FileDescriptor, F_DUPFD, 0)
dup2 (Old, New) is equivalent to:
close (New);
fcntl(Old, F_DUPFD, New)

The dup and dup2 subroutines differ from the fcntl subroutine in the following ways:

- If the file descriptor specified by the New parameter is greater than or equal to OPEN_MAX, the dup2 subroutine returns a -1 and sets the errno variable to EBADF.
- If the file descriptor specified by the Old parameter is valid and equal to the file descriptor specified by the New parameter, the dup2 subroutine will return the file descriptor specified by the New parameter, without closing it.
- If the file descriptor specified by the Old parameter is not valid, the dup2 subroutine will be unsuccessful and will not close the file descriptor specified by the New parameter.
- The value returned by the dup and dup2 subroutines is equal to the New parameter upon successful completion; otherwise, the return value is -1.

Return Values

Upon successful completion, the value returned depends on the value of the Command parameter, as follows:

<table>
<thead>
<tr>
<th>Command</th>
<th>Return Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>F_DUPFD</td>
<td>A new file descriptor</td>
</tr>
<tr>
<td>F_GETFD</td>
<td>The value of the flag (only the FD_CLOEXEC bit is defined)</td>
</tr>
<tr>
<td>F_SETFD</td>
<td>A value other than -1</td>
</tr>
<tr>
<td>F_GETFL</td>
<td>The value of file flags</td>
</tr>
<tr>
<td>F_SETFL</td>
<td>A value other than -1</td>
</tr>
<tr>
<td>F_GETOWN</td>
<td>The value of descriptor owner</td>
</tr>
<tr>
<td>F_SETOWN</td>
<td>A value other than -1</td>
</tr>
<tr>
<td>F_GETLK</td>
<td>A value other than -1</td>
</tr>
<tr>
<td>F_SETLK</td>
<td>A value other than -1</td>
</tr>
<tr>
<td>F_SETLKW</td>
<td>A value other than -1</td>
</tr>
<tr>
<td>F_CLOSEM</td>
<td>A value other than -1</td>
</tr>
</tbody>
</table>

If the fcntl subroutine fails, a value of -1 is returned and the errno global variable is set to indicate the error.

Error Codes

The fcntl subroutine is unsuccessful if one or more of the following are true:

- **EACCES** The Command argument is F_SETLK; the type of lock is a shared or exclusive lock and the segment of a file to be locked is already exclusively-locked by another process, or the type is an exclusive lock and some portion of the segment of a file to be locked is already shared-locked or exclusive-locked by another process.
- **EBADF** The FileDescriptor parameter is not a valid open file descriptor.
- **EDEADLK** The Command argument is F_SETLKW; the lock is blocked by some lock from another process and putting the calling process to sleep, waiting for that lock to become free would cause a deadlock.
- **EMFILE** The Command parameter is F_DUPFD, and the maximum number of file descriptors are currently open (OPEN_MAX).
- **EINVAL** The Command parameter is F_DUPFD, and the Argument parameter is negative or greater than or equal to OPEN_MAX.
- **EINVAL** An illegal value was provided for the Command parameter.
- **EINVAL** An attempt was made to lock a fifo or pipe.
- **ESRCH** The value of the Command parameter is F_SETOWN, and the process ID specified as the Argument parameter is not in use.
The Command parameter was F_SETLKW and the process received a signal while waiting to acquire the lock.

EOVERFLOW The Command parameter was F_GETLK and the block lock could not be represented in the flock structure.

The dup and dup2 subroutines fail if one or both of the following are true:

EBADF The Old parameter specifies an invalid open file descriptor or the New parameter specifies a file descriptor that is out of range.

EMFILE The number of file descriptors exceeds the OPEN_MAX value or there is no file descriptor above the value of the New parameter.

If NFS is installed on the system, the fcntl subroutine can fail if the following is true:

ETIMEDOUT The connection timed out.

Related Information

The close subroutine, execl, execv, execle, execve, execlp, execvp, or exact subroutine, fork, f_fork, or vfork subroutine, ioctl or ioctlx subroutine, lockf, lockfx, lockf, flock, lockf64 Subroutine, open, openx, or creat subroutine.

Files, Directories, and File Systems for Programmers in AIX 5L Version 5.3 General Programming Concepts: Writing and Debugging Programs.

fdetach Subroutine

Purpose
Detaches STREAMS-based file from the file to which it was attached.

Library
Standard C Library (libc.a)

Syntax
#include <stropts.h>
int fdetach(const char *path);

Parameters
path Pathname of a file previous associated with a STREAMS-based object using the fattach subroutine.

Description
The fdetach subroutine detaches a STREAMS-based file from the file to which it was attached by a previous call to fattach subroutine. The path argument points to the pathname of the attached STREAMS file. The process must have appropriate privileges or be the owner of the file. A successful call to fdetach subroutine causes all pathnames that named the attached STREAMS file to again name the file to which the STREAMS file was attached. All subsequent operations on path will operate on the underlying file and not on the STREAMS file.
All open file descriptors established while the STREAMS file was attached to the file referenced by path will still refer to the STREAMS file after the fdetach subroutine has taken effect.

If there are no open file descriptors or other references to the STREAMS file, then a successful call to fdetach subroutine has the same effect as performing the last close subroutine on the attached file.

The umount command may be used to detach a file name if an | application exits before performing fdetach subroutine.

**Return Value**

<table>
<thead>
<tr>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Successful completion.</td>
</tr>
<tr>
<td>-1</td>
<td>Not successful and errno set to one of the following.</td>
</tr>
</tbody>
</table>

**Errno Value**

<table>
<thead>
<tr>
<th>Errno</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EACCES</td>
<td>Search permission is denied on a component of the path prefix.</td>
</tr>
<tr>
<td>EPERM</td>
<td>The effective user ID is not the owner of path and the process does not have appropriate privileges.</td>
</tr>
<tr>
<td>ENOTDIR</td>
<td>A component of the path prefix is not a directory.</td>
</tr>
<tr>
<td>ENOENT</td>
<td>A component of path parameter does not name an existing file or path is an empty string.</td>
</tr>
<tr>
<td>EINVAL</td>
<td>The path parameter names a file that is not currently attached.</td>
</tr>
<tr>
<td>ENAMETOOLONG</td>
<td>The size of path parameter exceeds (PATH_MAX), or a component of path is longer than (NAME_MAX).</td>
</tr>
<tr>
<td>ELOOP</td>
<td>Too many symbolic links were encountered in resolving the path parameter.</td>
</tr>
<tr>
<td>ENOMEM</td>
<td>Insufficient storage space is available.</td>
</tr>
</tbody>
</table>

**Related Information**
The fattach subroutine, isastream subroutine.

---

**fdim, fdimf, or fdiml Subroutine**

**Purpose**

Computes the positive difference between two floating-point numbers.

**Syntax**

```c
#include <math.h>

double fdim (, )
double x;
double y;

float fdimf (x, y)
float x;
float y;

long double fdiml (x, y)
long double x;
long double y;
```

**Description**

The fdim, fdimf, and fdiml subroutines determine the positive difference between their arguments. If x is greater than y, x - y is returned. If x is less than or equal to y, +0 is returned.
An application wishing to check for error situations should set the `errno` global variable to zero and call `feclearexcept(FE_ALL_EXCEPT)` before calling these subroutines. Upon return, if `errno` is nonzero or `fetestexcept(FE_INVALID | FE_DIVBYZERO | FE_OVERFLOW | FE_UNDERFLOW)` is nonzero, an error has occurred.

**Parameters**

- `x` Specifies the value to be computed.
- `y` Specifies the value to be computed.

**Return Values**

Upon successful completion, the `fdim`, `fdimf`, and `fdiml` subroutines return the positive difference value.

If `x-y` is positive and overflows, a range error occurs and the `fdim`, `fdimf`, and `fdiml` subroutines return the value of the macro `HUGE_VAL`, `HUGE_VALF`, and `HUGE_VALL`, respectively.

If `x-y` is positive and underflows, a range error may occur, and 0.0 is returned.

If `x` or `y` is NaN, a NaN is returned.

**Related Information**

- “feclearexcept Subroutine,” “fetestexcept Subroutine” on page 270, “fmax, fmaxf, or fmaxl Subroutine” on page 277, and “madd, msup, mult, mdiv, pow, gcdn, invert, rpow, msqrt, mcmp, move, min, omin, fmin, m_in, mout, omout, fmout, m_out, sdiv, or itom Subroutine” on page 776.

**feclearexcept Subroutine**

**Purpose**

Clears floating-point exceptions.

**Syntax**

```c
#include <fenv.h>

int feclearexcept (excepts);

int excepts;
```

**Description**

The `feclearexcept` subroutine attempts to clear the supported floating-point exceptions represented by the `excepts` parameter.

**Parameters**

- `excepts` Specifies the supported floating-point exception to be cleared.

**Return Values**

If the `excepts` parameter is zero or if all the specified exceptions were successfully cleared, the `feclearexcept` subroutine returns zero. Otherwise, it returns a nonzero value.
fegetenv or fesetenv Subroutine

Purpose
Gets and sets the current floating-point environment.

Syntax
```c
#include <fenv.h>

int fegetenv (envp)
  envp_t *envp;

int fesetenv (envp)
  const env_t *envp;
```

Description
The `fegetenv` subroutine stores the current floating-point environment in the object pointed to by the `envp` parameter.

The `fesetenv` subroutine attempts to establish the floating-point environment represented by the object pointed to by the `envp` parameter. The `envp` parameter points to an object set by a call to the `fegetenv` or `feholdexcept` subroutines, or equal a floating-point environment macro. The `fesetenv` subroutine does not raise floating-point exceptions. It only installs the state of the floating-point status flags represented through its argument.

Parameters
`envp` Points to an object set by a call to the `fegetenv` or `feholdexcept` subroutines, or equal a floating-point environment macro.

Return Values
If the representation was successfully stored, the `fegetenv` subroutine returns zero. Otherwise, it returns a nonzero value. If the environment was successfully established, the `fesetenv` subroutine returns zero. Otherwise, it returns a nonzero value.

Related Information
"feholdexcept Subroutine" on page 265 and "feupdateenv Subroutine" on page 271

fegetexceptflag or fesetexceptflag Subroutine

Purpose
Gets and sets floating-point status flags.

Syntax
```c
#include <fenv.h>

int fegetexceptflag (flagp, flags)
  feexcept_t *flagp;

int fesetexceptflag (flagp, flags)
  const feexcept_t *flagp;
```
int excepts;

int fesetexceptflag (flagp, excepts)
const fexcept_t *flagp;
int excepts;

Description
The fesetexceptflag subroutine attempts to store an implementation-defined representation of the states of the floating-point status flags indicated by the excepts parameter in the object pointed to by the flagp parameter.

The fesetexceptflag subroutine attempts to set the floating-point status flags indicated by the excepts parameter to the states stored in the object pointed to by the flagp parameter. The value pointed to by the flagp parameter shall have been set by a previous call to the fesetexceptflag subroutine whose second argument represented at least those floating-point exceptions represented by the excepts parameter. This subroutine does not raise floating-point exceptions. It only sets the state of the flags.

Parameters
flagp Points to the object that holds the implementation-defined representation of the states of the floating-point status flags.
excepts Points to an implementation-defined representation of the states of the floating-point status flags.

Return Values
If the representation was successfully stored, the fesetexceptflag parameter returns zero. Otherwise, it returns a nonzero value. If the excepts parameter is zero or if all the specified exceptions were successfully set, the fesetexceptflag subroutine returns zero. Otherwise, it returns a nonzero value.

Related Information
"feraiseexcept Subroutine" on page 268 and "fetestexcept Subroutine" on page 270.

fegetround or fesetround Subroutine

Purpose
Gets and sets the current rounding direction.

Syntax
#include <fenv.h>

int fegetround (void)
int fesetround (round)
int round;

Description
The fegetround subroutine gets the current rounding direction.

The fesetround subroutine establishes the rounding direction represented by the round parameter. If the round parameter is not equal to the value of a rounding direction macro, the rounding direction is not changed.
Parameters

round Specifies the rounding direction.

Return Values
The fgetround subroutine returns the value of the rounding direction macro representing the current rounding direction or a negative value if there is no such rounding direction macro or the current rounding direction is not determinable.

The fesetround subroutine returns a zero value if the requested rounding direction was established.

feholdexcept Subroutine

Purpose
Saves current floating-point environment.

Syntax

```c
#include <fenv.h>

int feholdexcept (envp)
  envp_t *envp;
```

Description
The feholdexcept subroutine saves the current floating-point environment in the object pointed to by envp, clears the floating-point status flags, and installs a non-stop (continue on floating-point exceptions) mode for all floating-point exceptions.

Parameters

envp Points to the current floating-point environment.

Return Values
The feholdexcept subroutine returns zero if non-stop floating-point exception handling was successfully installed.

Related Information
The “feupdateenv Subroutine” on page 271.

fence Subroutine

Purpose
Allows you to request and change the virtual shared disk fence map.

Syntax

```c
#include <vsd_ioctl.h>

int ioctl(FileDescriptor, Command, Argument)
  int FileDescriptor, Command;
  void *Argument;
```
Description

Use this subroutine to request and change the virtual shared disk fence map. The fence map, which controls whether virtual shared disks can send or satisfy requests from virtual shared disks at remote nodes, is defined as:

```c
struct vsd_FenceMap /* This is the argument to the VSD fence ioctl. */
{
    ulong flags;
    vsd_minorBitmap_t minornoBitmap; /* Bitmap of minor numbers to fence */
        /* Bitmap of minor numbers to fence (supports 10000 vsds) */
    vsd_Fence_Bitmap_t nodesBitmap; /* Nodes to (un)fence these vsds from */
        /* Nodes to (un)fence these vsds from (supports node numbers 1-2048) */
}
```

The flags `VSD_FENCE` and `VSD_UNFENCE` are mutually exclusive — an ioctl can either fence a set of virtual shared disks or unfence a set of virtual shared disks, but not both. The `minornoBitmap` denotes which virtual shared disks are to be fenced/unfenced from the nodes specified in the `nodesBitmap`.

Parameters

- **FileDescriptor** Specifies the open file descriptor for which the control operation is to be performed.
- **Command** Specifies the control function to be performed. The value of this parameter is always `GIOCFENCE`.
- **Argument** Specifies a pointer to a `vsd_fence_map` structure.

The `flags` field of the `vsd_fence_map` structure determines the type of operation that is performed. The flags could be set with one or more options using the OR operator. These options are as follows:

- **VSD_FENCE_FORCE** If this option is specified, a node can unfence itself.
- **VSD_FENCE_GET** Denotes a query request.
- **VSD_FENCE** Denotes a fence request.
- **VSD_UNFENCE** Denotes an unfence request.

Examples

The following example fences a virtual shared disk with a minor number of 7 from node 4 and 5, and unfences a virtual shared disk with a minor number of 5 from node 1:

```c
int fd;
vsd_FenceMap_t FenceMap;

/* Clear the FenceMap */
bzero(FenceMap, sizeof(vsd_FenceMap_t));

/* fence nodes 4,5 from minor 7 */
FenceMap.flags = VSD_FENCE;
MAP_SET(7, FenceMap.minornoBitmap);
MAP_SET(4, FenceMap.nodesBitmap);
MAP_SET(5, FenceMap.nodesBitmap);

/* Issue the fence request */
ioctl(fd, GIOCFENCE, &FenceMap);

/* Unfence node 1 from minor 5 */
bzero(FenceMap, sizeof(vsd_FenceMap_t));
FenceMap.flags = VSD_UNFENCE | VSD_FENCE_FORCE;
MAP_SET(5, FenceMap.minornoBitmap);
MAP_SET(1, FenceMap.nodesBitmap);

/* Issue the fence request */
ioctl(fd, GIOCFENCE, &FenceMap);
```
Return Values
If the request succeeds, the ioctl returns 0. In the case of an error, a value of -1 is returned with the global variable errno set to identify the error.

Error Values
The fence ioctl subroutine can return the following error codes:

**EACCES** Indicates that an unfence was requested from a fenced node without the VSD_FENCE_FORCE option.

**EINVAL** Indicates an invalid request (ambiguous flags or unidentified virtual shared disks).

**ENOCONNECT** Indicates that either the primary or the secondary node for a virtual shared disk to be fenced is not a member of the virtual shared disk group, or the virtual shared disk in question is in the **stopped** state.

**ENOTREADY** Indicates that the group is not active or the Recoverable virtual shared disk subsystem is not available.

**ENXIO** Indicates that the Virtual shared disk driver is being unloaded.

**feof, ferror, clearerr, or fileno Macro**

**Purpose**
Checks the status of a stream.

**Library**
Standard C Library (libc.a)

**Syntax**
```c
#include <stdio.h>

int feof (FILE *Stream);
int ferror (FILE *Stream);
void clearerr (FILE *Stream);
int fileno (FILE *Stream);
```

**Description**
The **feof** macro inquires about the end-of-file character (EOF). If EOF has previously been detected reading the input stream specified by the Stream parameter, a nonzero value is returned. Otherwise, a value of 0 is returned.

The **ferror** macro inquires about input or output errors. If an I/O error has previously occurred when reading from or writing to the stream specified by the Stream parameter, a nonzero value is returned. Otherwise, a value of 0 is returned.

The **clearerr** macro inquires about the status of a stream. The **clearerr** macro resets the error indicator and the EOF indicator to a value of 0 for the stream specified by the Stream parameter.
The `fileno` macro inquires about the status of a stream. The `fileno` macro returns the integer file descriptor associated with the stream pointed to by the `Stream` parameter. Otherwise a value of -1 is returned.

**Parameters**

STREAM

Specifies the input or output stream.

**Related Information**

The `fopen`, `freopen`, or `fdopen` ("fopen, fopen64, freopen, freopen64 or fdopen Subroutine" on page 284) subroutine, `open` ("open, openx, open64, creat, or creat64 Subroutine" on page 925) subroutine.

`Input and Output Handling` in AIX 5L Version 5.3 General Programming Concepts: Writing and Debugging Programs.

---

**feraiseexcept Subroutine**

**Purpose**

Raises the floating-point exception.

**Syntax**

```c
#include <fenv.h>

int feraiseexcept (excepts)
int excepts;
```

**Description**

The `feraiseexcept` subroutine attempts to raise the supported floating-point exceptions represented by the `excepts` parameter. The order in which these floating-point exceptions are raised is unspecified.

**Parameters**

`excepts` Points to the floating-point exceptions.

**Return Values**

If the argument is zero or if all the specified exceptions were successfully raised, the `feraiseexcept` subroutine returns a zero. Otherwise, it returns a nonzero value.

**Related Information**

"feclearexcept Subroutine" on page 262, "fegetexceptflag or fesetexceptflag Subroutine" on page 263, "fetestexcept Subroutine" on page 270.

---

**fetch_and_add Subroutine**

**Purpose**

Updates a single word variable atomically.

**Library**

Standard C library (libc.a)
#include <sys/atomic_op.h>

```c
int fetch_and_add (word_addr, value)
atomic_p word_addr;
int value;
```

**Description**

The `fetch_and_add` subroutine increments one word in a single atomic operation. This operation is useful when a counter variable is shared between several threads or processes. When updating such a counter variable, it is important to make sure that the fetch, update, and store operations occur atomically (are not interruptible). For example, consider the sequence of events which could occur if the operations were interruptible:

1. A process fetches the counter value and adds one to it.
2. A second process fetches the counter value, adds one, and stores it.
3. The first process stores its value.

The result of this is that the update made by the second process is lost.

Traditionally, atomic access to a shared variable would be controlled by a mechanism such as semaphores. Compared to such mechanisms, the `fetch_and_add` subroutine requires very little overhead, and provided that the counter variable fits in a single machine word, this subroutine provides a highly efficient way of performing this operation.

**Note:** The word containing the counter variable must be aligned on a full word boundary.

**Parameters**

- **word_addr**
  Specifies the address of the word variable to be incremented.
- **value**
  Specifies the value to be added to the word variable.

**Return Values**

This subroutine returns the original value of the word.

**Related Information**

The `fetch_and_and` ("fetch_and_and or fetch_and_or Subroutine") subroutine, `fetch_and_or` ("fetch_and_and or fetch_and_or Subroutine") subroutine, `compare_and_swap` ("compare_and_swap Subroutine" on page 176) subroutine.

---

### fetch_and_and or fetch_and_or Subroutine

**Purpose**

Sets or clears bits in a single word variable atomically.

**Library**

Standard C library (`libc.a`)

**Syntax**

```c
#include <sys/atomic_op.h>
```
uint fetch_and_and (word_addr, mask)
atomic_p word_addr;
int mask;

uint fetch_and_or (word_addr, mask)
atomic_p word_addr;
int mask;

Description
The fetch_and_and and fetch_and_or subroutines respectively clear and set bits in one word, according to a bit mask, in a single atomic operation. The fetch_and_and subroutine clears bits in the word which correspond to clear bits in the bit mask, and the fetch_and_or subroutine sets bits in the word which correspond to set bits in the bit mask.

These operations are useful when a variable containing bit flags is shared between several threads or processes. When updating such a variable, it is important that the fetch, bit clear or set, and store operations occur atomically (are not interruptible). For example, consider the sequence of events which could occur if the operations were interruptible:
1. A process fetches the flags variable and sets a bit in it.
2. A second process fetches the flags variable, sets a different bit, and stores it.
3. The first process stores its value.

The result is that the update made by the second process is lost.

Traditionally, atomic access to a shared variable would be controlled by a mechanism such as semaphores. Compared to such mechanisms, the fetch_and_and and fetch_and_or subroutines require very little overhead, and provided that the flags variable fits in a single machine word, they provide a highly efficient way of performing this operation.

Note: The word containing the flag bits must be aligned on a full word boundary.

Parameters

word_addr Specifies the address of the single word variable whose bits are to be cleared or set.
mask Specifies the bit mask which is to be applied to the single word variable.

Return Values
These subroutines return the original value of the word.

Related Information
The fetch_and_add ("fetch_and_add Subroutine" on page 268) subroutine, compare_and_swap ("compare_and_swap Subroutine" on page 176) subroutine.

fetestexcept Subroutine

Purpose
Tests floating-point exception flags.
Syntax
```c
#include <fenv.h>

int fetestexcept (excepts);
int excepts;
```

Description
The `fetestexcept` subroutine determines which of a specified subset of the floating-point exception flags are currently set. The `excepts` parameter specifies the floating-point status flags to be queried.

Parameters
`excepts` Specifies the floating-point status flags to be queried.

Return Values
The `fetestexcept` subroutine returns the value of the bitwise-inclusive OR of the floating-point exception macros corresponding to the currently set floating-point exceptions included in `excepts`.

Related Information
"feclearexcept Subroutine" on page 262, "fegetexceptflag or fesetexceptflag Subroutine" on page 263, and "feraiseexcept Subroutine" on page 268

---

**feupdateenv Subroutine**

Purpose
Updates floating-point environment.

Syntax
```c
#include <fenv.h>

int feupdateenv (envp);
const fenv_t *envp;
```

Description
The `feupdateenv` subroutine attempts to save the currently raised floating-point exceptions in its automatic storage, attempts to install the floating-point environment represented by the object pointed to by the `envp` parameter, and attempts to raise the saved floating-point exceptions. The `envp` parameter point to an object set by a call to `feholdexcept` or `fegetenv`, or equal a floating-point environment macro.

Parameters
`envp` Points to an object set by a call to the `feholdexcept` or the `fegetenv` subroutine, or equal a floating-point environment macro.

Return Values
The `feupdateenv` subroutine returns a zero value if all the required actions were successfully carried out.

Related Information
"fegetenv or fesetenv Subroutine" on page 263 and "feholdexcept Subroutine" on page 265.
finfo or ffinfo Subroutine

Purpose
Returns file information.

Library
Standard C library (libc.a)

Syntax
```c
#include <sys/finfo.h>

int finfo(Path1, cmd, buffer, length)
const char *Path1;
int cmd;
void *buffer;
int length;

int ffinfo(fd, cmd, buffer, length)
int fd;
int cmd;
void *buffer;
int length;
```

Description
The `finfo` and `ffinfo` subroutines return specific file information for the specified file.

Parameters
- `Path1` Path name of a file system object to query.
- `fd` File descriptor for an open file to query.
- `cmd` Specifies the type of file information to be returned.
- `buffer` User supplied buffer which contains the file information upon successful return. `/usr/include/sys/finfo.h` describes the buffer.
- `length` Length of the query buffer.

Commands
- **F_PATHCONF** When the `F_PATHCONF` command is specified, a file’s implementation information is returned.
  
  **Note:** The operating system provides another subroutine that retrieves file implementation characteristics, `pathconf` ("pathconf or fpathconf Subroutine" on page 969) command. While the `finfo` and `ffinfo` subroutines can be used to retrieve file information, it is preferred that programs use the `pathconf` interface.

- **F_DIOCAP** When the `F_DIOCAP` command is specified, the file’s direct 1/0 capability information is returned. The buffer supplied by the application is of type `struct diocapbuf`.

Return Values
Upon successful completion, the `finfo` and `ffinfo` subroutines return a value of 0 and the user supplied buffer is correctly filled in with the file information requested. If the `finfo` or `ffinfo` subroutines were unsuccessful, a value of -1 is returned and the global `errno` variable is set to indicate the error.
## Error Codes

<table>
<thead>
<tr>
<th>Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EACCES</td>
<td>Search permission is denied for a component of the path prefix.</td>
</tr>
<tr>
<td>EINVAL</td>
<td>If the length specified for the user buffer is greater than <code>MAX_FINFO_BUF</code>.</td>
</tr>
<tr>
<td>ENAMETOOLONG</td>
<td>The length of the Path parameter string exceeds the <code>PATH_MAX</code> value.</td>
</tr>
<tr>
<td>ENOENT</td>
<td>The named file does not exist or the Path parameter points to an empty string.</td>
</tr>
<tr>
<td>ENOTDIR</td>
<td>A component of the path prefix is not a directory.</td>
</tr>
<tr>
<td>EBADF</td>
<td>File descriptor provided is not valid.</td>
</tr>
</tbody>
</table>

## Related Information

The `pathconf` subroutine.

[Subroutines, Example Programs, and Libraries in AIX 5L Version 5.3 General Programming Concepts: Writing and Debugging Programs](#)

### flockfile, ftrylockfile, funlockfile Subroutine

#### Purpose

Provides for explicit application-level locking of stdio (FILE*) objects.

#### Library

Standard Library (`libc.a`)

#### Syntax

```c
#include <stdio.h>
void flockfile (FILE * file)
int ftrylockfile (FILE * file)
void funlockfile (FILE * file)
```

#### Description

The `flockfile`, `ftrylockfile` and `funlockfile` functions provide for explicit application-level locking of stdio (FILE*) objects. These functions can be used by a thread to delineate a sequence of I/O statements that are to be executed as a unit.

The `flockfile` function is used by a thread to acquire ownership of a (FILE*) object.

The `ftrylockfile` function is used by a thread to acquire ownership of a (FILE*) object if the object is available; `ftrylockfile` is a non-blocking version of `flockfile`.

The `funlockfile` function is used to relinquish the ownership granted to the thread. The behavior is undefined if a thread other than the current owner calls the `funlockfile` function.

Logically, there is a lock count associated with each (FILE*) object. This count is implicitly initialised to zero when the (FILE*) object is created. The (FILE*) object is unlocked when the count is zero. When the count is positive, a single thread owns the (FILE*) object. When the `flockfile` function is called, if the count is zero or if the count is positive and the caller owns the (FILE*) object, the count is incremented.
Otherwise, the calling thread is suspended, waiting for the count to return to zero. Each call to `funlockfile` decrements the count. This allows matching calls to `flockfile` (or successful calls to `ftrylockfile`) and `funlockfile` to be nested.

All functions that reference `(FILE*)` objects behave as if they use `flockfile` and `funlockfile` internally to obtain ownership of these `(FILE*)` objects.

**Return Values**

None for `flockfile` and `funlockfile`. The function `ftrylock` returns zero for success and non-zero to indicate that the lock cannot be acquired.

**Implementation Specifics**

These subroutines are part of Base Operating System (BOS) subroutines.

Realtime applications may encounter priority inversion when using FILE locks. The problem occurs when a high priority thread locks a file that is about to be unlocked by a low priority thread, but the low priority thread is preempted by a medium priority thread. This scenario leads to priority inversion; a high priority thread is blocked by lower priority threads for an unlimited period of time. During system design, realtime programmers must take into account the possibility of this kind of priority inversion. They can deal with it in a number of 7434 ways, such as by having critical sections that are guarded by file locks execute at a high priority, so that a thread cannot be preempted while executing in its critical section.

**Related Information**

The `getc_unlocked`, `getchar_unlocked`, `putc_unlocked`, `putchar_unlocked` subroutine.

`floor`, `floorf`, `floorl`, `nearest`, `trunc`, `itrunc`, or `uitrunc` Subroutine

**Purpose**

The `floor` subroutine, `floorf` subroutine, `floorl` subroutine, `nearest` subroutine, and `trunc` subroutine, round floating-point numbers to floating-point integer values.

The `itrunc` subroutine and `uitrunc` subroutine round floating-point numbers to signed and unsigned integers, respectively.

**Libraries**

IEEE Math Library (`libm.a`)  
or System V Math Library (`libmsaa.a`)  
Standard C Library (`libc.a`) (separate syntax follows)

**Syntax**

```c
#include <math.h>

double floor (x)
double x;
float floorf (x)
float x;
long double floorl (x)
long double x;
double nearest (x)
double x;
```
double trunc (x)
double x;

Standard C Library (libc.a)
#include <stdlib.h>
#include <limits.h>
int itrunc (x)
double x;
unsigned int uitrunc (x)
double x;

Description
The floor subroutine and floorl subroutines return the largest floating-point integer value not greater than the x parameter.

An application wishing to check for error situations should set errno to zero and call feclearexcept(FE_ALL_EXCEPT) before calling these subroutines. Upon return, if errno is nonzero or fetestexcept(FE_INVALID | FE_DIVBYZERO | FE_OVERFLOW | FE_UNDERFLOW) is nonzero, an error has occurred.

The nearest subroutine returns the nearest floating-point integer value to the x parameter. If x lies exactly halfway between the two nearest floating-point integer values, an even floating-point integer is returned.

The trunc subroutine returns the nearest floating-point integer value to the x parameter in the direction of 0. This is equivalent to truncating off the fraction bits of the x parameter.

Note: The default floating-point rounding mode is round to nearest. All C main programs begin with the rounding mode set to round to nearest.

The itrunc subroutine returns the nearest signed integer to the x parameter in the direction of 0. This is equivalent to truncating the fraction bits from the x parameter and then converting x to a signed integer.

The uitrunc subroutine returns the nearest unsigned integer to the x parameter in the direction of 0. This action is equivalent to truncating off the fraction bits of the x parameter and then converting x to an unsigned integer.

Note: Compile any routine that uses subroutines from the libm.a library with the -lm flag. To compile the floor.c file, for example, enter:
    cc floor.c -lm

The itrunc, uitrunc, trunc, and nearest subroutines are not part of the ANSI C Library.

Parameters
x    Specifies a double-precision floating-point value. For the floor subroutine, specifies a long double-precision floating-point value.

y    Specifies a double-precision floating-point value. For the floor subroutine, specifies some long double-precision floating-point value.

Return Values
Upon successful completion, the floor, floor, and floorl subroutine returns the largest integral value not greater than x, expressed as a double, float, or long double, as appropriate for the return type of the function.
If $x$ is NaN, a NaN is returned.

If $x$ is ±0 or ±Inf, $x$ is returned.

If the correct value would cause overflow, a range error occurs and the floor, floorf and floorl subroutines return the value of the macro -HUGE_VAL, -HUGE_VALF and -HUGE_VALL, respectively.

**Error Codes**

The itrunc and uitrunc subroutines return the INT_MAX value if $x$ is greater than or equal to the INT_MAX value and the INT_MIN value if $x$ is equal to or less than the INT_MIN value. The itrunc subroutine returns the INT_MIN value if $x$ is a Quiet NaN(not-a-number) or Silent NaN. The uitrunc subroutine returns 0 if $x$ is a Quiet NaN or Silent NaN. (The INT_MAX and INT_MIN values are defined in the limits.h file.) The uitrunc subroutine INT_MAX if $x$ is greater than INT_MAX and 0 if $x$ is less than or equal 0.0

**Files**

float.h Contains the ANSI C FLT_ROUNDS macro.

**Related Information**

“feclearexcept Subroutine” on page 262, “fetestexcept Subroutine” on page 270, and “class, _class, finite, isnan, or unordered Subroutines” on page 167.

The fp_read_rnd or fp_swap_rnd subroutine.

**Subroutines Overview** in AIX 5L Version 5.3 General Programming Concepts: Writing and Debugging Programs.

**128-Bit long double Floating-Point Format** in AIX 5L Version 5.3 General Programming Concepts: Writing and Debugging Programs.

**math.h** in AIX 5L Version 5.3 Files Reference.

**fma, fmaf, or fmal Subroutine**

**Purpose**

Floating-point multiply-add.

**Syntax**

```c
#include <math.h>

double fma (d1, d2, d3)
double x;
double y;
double z;

double fma (d1, d2, d3)

double fma (d1, d2, d3)

double fma (d1, d2, d3)
```

276 Technical Reference, Volume 1: Base Operating System and Extensions
Description

The fma, fmaf, and fmal subroutines compute \((x \times y) + z\), rounded as one ternary operation. They compute the value (as if) to infinite precision and round once to the result format, according to the rounding mode characterized by the value of FLT_ROUNDS.

An application wishing to check for error situations should set the errno global variable to zero and call feclearexcept(FE_ALL_EXCEPT) before calling these subroutines. Upon return, if errno is nonzero or fetestexcept(FE_INVALID | FE_DIVBYZERO | FE_OVERFLOW | FE_UNDERFLOW) is nonzero, an error has occurred.

Parameters

\(x\) Specifies the value to be multiplied by the \(y\) parameter.
\(y\) Specifies the value to be multiplied by the \(x\) parameter.
\(z\) Specifies the value to be added to the product of the \(x\) and \(y\) parameters.

Return Values

Upon successful completion, the fma, fmaf, and fmal subroutines return \((x \times y) + z\), rounded as one ternary operation.

If \(x\) or \(y\) are NaN, a NaN is returned.

If \(x\) multiplied by \(y\) is an exact infinity and \(z\) is also an infinity but with the opposite sign, a domain error occurs, and a NaN is returned.

If one of the \(x\) and \(y\) parameters is infinite, the other is zero, and the \(z\) parameter is not a NaN, a domain error occurs, and a NaN is returned.

If one of the \(x\) and \(y\) parameters is infinite, the other is zero, and \(z\) is a NaN, a NaN is returned and a domain error may occur.

If \(x\times y\) is not 0\*Inf nor Inf\*0 and \(z\) is a NaN, a NaN is returned.

Related Information

“feclearexcept Subroutine” on page 262 and “fetestexcept Subroutine” on page 270.

math.h in AIX 5L Version 5.3 Files Reference.

fmax, fmaxf, or fmaxl Subroutine

Purpose

Determines the maximum numeric value of two floating-point numbers.

Syntax

```c
#include <math.h>

double fmax (double x, double y);

calculate maximum

float fmaxf (float x, float y);

calculate maximum
```

Base Operating System (BOS) Runtime Services (A-P) 277
long double fmaxl (x, y)
long double x;
long double y;

Description
The fmax, fmaxf, and fmaxl subroutines determine the maximum numeric value of their arguments. NaN arguments are treated as missing data. If one argument is a NaN and the other numeric, the fmax, fmaxf, and fmaxl subroutines choose the numeric value.

Parameters

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>x</td>
<td>Specifies the value to be computed.</td>
</tr>
<tr>
<td>y</td>
<td>Specifies the value to be computed.</td>
</tr>
</tbody>
</table>

Return Values
Upon successful completion, the fmax, fmaxf, and fmaxl subroutines return the maximum numeric value of their arguments.

If one argument is a NaN, the other argument is returned.

If x and y are NaN, a NaN is returned.

Related Information
“fdim, fdimf, or fdiml Subroutine” on page 261 and “madd, msub, mult, mdiv, pow, gcd, invert, rpow, msqrt, mcmp, move, min, omin, fmin, m_in, mout, omout, fmout, m_out, sdiv, or itom Subroutine” on page 776

math.h in AIX 5L Version 5.3 Files Reference.

fminf or fminl Subroutine

Purpose
Determines the minimum numeric value of two floating-point numbers.

Syntax
#include <math.h>

float fminf (x, y)
float x;
float y;

long double fminl (x, y)
long double x;
long double y;

Description
The fminf and fminl subroutines determine the minimum numeric value of their arguments. NaN arguments are treated as missing data. If one argument is a NaN and the other numeric, the fminf and fminl subroutines choose the numeric value.

Parameters

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>x</td>
<td>Specifies the value to be computed.</td>
</tr>
</tbody>
</table>
y Specifies the value to be computed.

Return Values
Upon successful completion, the fminf and fminl subroutines return the minimum numeric value of their arguments.

If one argument is a NaN, the other argument is returned.

If x and y are NaN, a NaN is returned.

Related Information
“fdim, fdimf, or fdiml Subroutine” on page 261, “fmax, fmaxf, or fmaxl Subroutine” on page 277.

math.h in AIX 5L Version 5.3 Files Reference.

fmod, fmodf, or fmodl Subroutine

Purpose
Computes the floating-point remainder value.

Syntax
#include <math.h>

float fmodf (x, y)
float x;
float y;

long double fmodl (x)
long double x, y;

double fmod (x, y)
double x, y;

Description
The fmodf, fmodl, and fmod subroutines return the floating-point remainder of the division of x by y.

An application wishing to check for error situations should set the errno global variable to zero and call feclearexcept(FE_ALL_EXCEPT) before calling these subroutines. Upon return, if errno is nonzero or fetestexcept(FE_INVALID | FE_DIVBYZERO | FE_OVERFLOW | FE_UNDERFLOW) is nonzero, an error has occurred.

Parameters
x Specifies the value to be computed.
y Specifies the value to be computed.

Return Values
The fmodf, fmodl, and fmod subroutines return the value x - i * y, for some integer i such that, if y is nonzero, the result has the same sign as x and magnitude less than the magnitude of y.

If the correct value would cause underflow, and is not representable, a range error may occur, and 0.0 is returned.
If $x$ or $y$ is NaN, a NaN is returned.

If $y$ is zero, a domain error occurs, and a NaN is returned.

If $x$ is infinite, a domain error occurs, and a NaN is returned.

If $x$ is $\pm 0$ and $y$ is not zero, $\pm 0$ is returned.

If $x$ is not infinite and $y$ is $\pm \text{Inf}$, $x$ is returned.

If the correct value would cause underflow, and is representable, a range error may occur and the correct value is returned.

**Related Information**
- "feclearexcept Subroutine" on page 262
- "fetestexcept Subroutine" on page 270
- "class, _class, finite, isnan, or unordered Subroutines" on page 167
- `math.h` in AIX 5L Version 5.3 Files Reference.

---

**fmtmsg Subroutine**

**Purpose**
Display a message in the specified format on standard error, the console, or both.

**Library**
Standard C Library (`libc.a`)

**Syntax**

```c
#include <fmtmsg.h>

int fmtmsg (long Classification,
            const char *Label,
            int Severity,
            const char *Text;
            const char *Action,
            const char *Tag)
```

**Description**
The `fmtmsg` subroutine can be used to display messages in a specified format instead of the traditional `printf` subroutine interface.

Base on a message’s classification component, the `fmtmsg` subroutine either writes a formatted message to standard error, the console, or both.

A formatted message consists of up to five parameters. The `Classification` parameter is not part of a message displayed to the user, but defines the source of the message and directs the display of the formatted message.
Parameters

Classification
Contains identifiers from the following groups of major classifications and subclassifications. Any one identifier from a subclass may be used in combination with a single identifier from a different subclass. Two or more identifiers from the same subclass should not be used together, with the exception of identifiers from the display subclass. (Both display subclass identifiers may be used so that messages can be displayed to both standard error and system console).

major classifications
Identifies the source of the condition. Identifiers are: MM_HARD (hardware), MM_SOFT (software), and MM_FIRM (firmware).

message source subclassifications
Identifies the type of software in which the problem is detected. Identifiers are: MM_APPL (application), MM_UTIL (utility), and MM_OPSYS (operating system).

display subclassification
Indicates where the message is to be displayed. Identifiers are: MM_PRINT to display the message on the standard error stream, MM_CONSOLE to display the message on the system console. One or both identifiers may be used.

status subclassifications
Indicates whether the application will recover from the condition. Identifiers are: MM_RECOVER (recoverable) and MM_RECOV (non-recoverable).

An additional identifier, MM_NULLMC, identifies that no classification component is supplied for the message.

Label
Identifies the source to the message. The format is two fields separated by a colon. The first field is up to 10 bytes, the second field is up to 14 bytes.

Severity
Text
Describes the error condition that produced the message. The character string is not limited to a specific size. If the character string is null then a message will be issued stating that no text has been provided.

Action
Describes the first step to be taken in the error-recovery process. The fmtmsg subroutine precedes the action string with the prefix: TO FIX:. The Action string is not limited to a specific size.

Tag
An identifier which references online documentation for the message. Suggested usage is that tag includes the Label and a unique identifying number. A sample tag is UX:cat:146.

Environment Variables
The MSGVERB (message verbosity) environment variable controls the behavior of the fmtmsg subroutine.

MSGVERB tells the fmtmsg subroutine which message components it is to select when writing messages to standard error. The value of MSGVERB is a colon-separated list of optional keywords. MSGVERB can be set as follows:

MSGVERB=[keyword[:keyword[:...]]]
export  MSGVERB

Valid keywords are: Label, Severity, Text, Action, and Tag. If MSGVERB contains a keyword for a component and the component's value is not the component's null value, fmtmsg subroutine includes that component in the message when writing the message to standard error. If MSGVERB does not include a keyword for a message component, that component is not included in the display of the message. The keywords may appear in any order. If MSGVERB is not defined, if its value is the null string, if its value is not of the correct format, or if it contains keywords other than the valid ones listed previously, the fmtmsg subroutine selects all components.

MSGVERB affects only which components are selected for display to standard error. All message components are included in console messages.
Application Usage

One or more message components may be systematically omitted from messages generated by an application by using the null value of the parameter for that component. The table below indicates the null values and identifiers for `fmtmsg` subroutine parameters. The parameters are of type `char*` unless otherwise indicated.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Null-Value</th>
<th>Identifier</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>label</code></td>
<td>(char*)0</td>
<td><code>MM_NULLLBL</code></td>
</tr>
<tr>
<td><code>severity</code> (type int)</td>
<td>0</td>
<td><code>MM_NULLSEV</code></td>
</tr>
<tr>
<td><code>class</code> (type long)</td>
<td>0L</td>
<td><code>MM_NULLMC</code></td>
</tr>
<tr>
<td><code>text</code></td>
<td>(char*)0</td>
<td><code>MM_NULLTXT</code></td>
</tr>
<tr>
<td><code>action</code></td>
<td>(char*)0</td>
<td><code>MM_NULLACT</code></td>
</tr>
<tr>
<td><code>tag</code></td>
<td>(char*)0</td>
<td><code>MM_NULLTAG</code></td>
</tr>
</tbody>
</table>

Another means of systematically omitting a component is by omitting the component keywords when defining the MSGVERB environment variable.

Return Values

The exit codes for the `fmtmsg` subroutine are the following:

- **MM_OK**: The function succeeded.
- **MM_NOTOK**: The function failed completely.
- **MM_MOMSG**: The function was unable to generate a message on standard error.
- **MM_NOCON**: The function was unable to generate a console message.

Examples

1. The following example of the `fmtmsg` subroutine:

   ```c
   fmtmsg(MM_PRINT, "UX:cat", MM_ERROR, "illegal option", "refer to cat in user's reference manual", "UX:cat:001")
   ```

   produces a complete message in the specified message format:

   ```
   UX:cat ERROR: illegal option
   TO FIX: refer to cat in user's reference manual UX:cat:001
   ```

2. When the environment variable MSGVERB is set as follows:

   ```
   MSGVERB=severity:text:action
   ```

   and the Example 1 is used, the `fmtmsg` subroutine produces:

   ```
   ERROR: illegal option
   TO FIX: refer to cat in user's reference manual UX:cat:001
   ```

Related Information

The `printf` subroutine.

**fnmatch Subroutine**

**Purpose**

Matches file name patterns.
Library
Standard C Library (libc.a)

Syntax
#include <fnmatch.h>

int fnmatch (Pattern, String, Flags);
int Flags;
const char *Pattern, *String;

Description
The fnmatch subroutine checks the string specified by the String parameter to see if it matches the pattern specified by the Pattern parameter.

The fnmatch subroutine can be used by an application or command that needs to read a dictionary and apply a pattern against each entry; the find command is an example of this. It can also be used by the pax command to process its Pattern variables, or by applications that need to match strings in a similar manner.

Parameters

Pattern
Contains the pattern to which the String parameter is to be compared. The Pattern parameter can include the following special characters:

* (asterisk)
Matches zero, one, or more characters.

? (question mark)
Matches any single character, but will not match 0 (zero) characters.

[ ] (brackets)
Matches any one of the characters enclosed within the brackets. If a pair of characters separated by a dash are contained within the brackets, the pattern matches any character that lexically falls between the two characters in the current locale.

String
Contains the string to be compared against the Pattern parameter.

Flags
Contains a bit flag specifying the configurable attributes of the comparison to be performed by the fnmatch subroutine.

The Flags parameter modifies the interpretation of the Pattern and String parameters. It is the bitwise inclusive OR of zero or more of the following flags (defined in the fnmatch.h file):

FNMPATHNAME
Indicates the / (slash) in the String parameter matches a / in the Pattern parameter.

FNMPERIOD
Indicates a leading period in the String parameter matches a period in the Pattern parameter.

FNMNOESCAPE
Enables quoting of special characters using the \ (backslash).

FNMINCLUDECASE
Ignores uppercase and lowercase when matching alphabetic characters (available only in AIX 5.1 or later).

If the FNM_PATHNAME flag is set in the Flags parameter, a / (slash) in the String parameter is explicitly matched by a / in the Pattern parameter. It is not matched by either the * (asterisk) or ? (question-mark) special characters, nor by a bracket expression. If the FNM_PATHNAME flag is not set, the / is treated as an ordinary character.
If the FNM_PERIOD flag is set in the Flags parameter, then a leading period in the String parameter only matches a period in the Pattern parameter; it is not matched by either the asterisk or question-mark special characters, nor by a bracket expression. The setting of the FNM_PATHNAME flag determines a period to be leading, according to the following rules:

- If the FNM_PATHNAME flag is set, a . (period) is leading only if it is the first character in the String parameter or if it immediately follows a /.
- If the FNM_PATHNAME flag is not set, a . (period) is leading only if it is the first character of the String parameter. If FNM_PERIOD is not set, no special restrictions are placed on matching a period.

If the FNM_NOESCAPE flag is not set in the Flags parameter, a \ (backslash) character in the Pattern parameter, followed by any other character, will match that second character in the String parameter. For example, \ will match a backslash in the String parameter. If the FNM_NOESCAPE flag is set, a \ (backslash) will be treated as an ordinary character.

**Return Values**

If the value in the String parameter matches the pattern specified by the Pattern parameter, the fnmatch subroutine returns 0. If there is no match, the fnmatch subroutine returns the FNM_NOMATCH constant, which is defined in the fnmatch.h file. If an error occurs, the fnmatch subroutine returns a nonzero value.

**Files**

/usr/include/fnmatch.h Contains system-defined flags and constants.

**Related Information**

The glob subroutine, globfree subroutine, regcomp subroutine, regfree subroutine, regerror subroutine, regexec subroutine.

The find command, pax command.

Files, Directories, and File Systems for Programmers and Understanding Internationalized Regular Expression Subroutines in AIX 5L Version 5.3 General Programming Concepts: Writing and Debugging Programs

---

**fopen, fopen64, freopen, freopen64 or fdopen Subroutine**

**Purpose**

Opens a stream.

**Library**

Standard C Library (libc.a)

**Syntax**

```c
#include <stdio.h>
FILE *fopen ( Path, Type);
const char *Path, *Type;

FILE *fopen64 ( Path, Type);
char *Path, *Type;

FILE *freopen (Path, Type, Stream);
const char *Path, *Type;
```
Description

The fopen and fopen64 subroutines open the file named by the Path parameter and associate a stream with it and return a pointer to the FILE structure of this stream.

When you open a file for update, you can perform both input and output operations on the resulting stream. However, an output operation cannot be directly followed by an input operation without an intervening fflush subroutine call or a file positioning operation (fseek, fseeko, fseeko64, fsetpos, fsetpos64, or rewind subroutine). Also, an input operation cannot be directly followed by an output operation without an intervening flush or file positioning operation, unless the input operation encounters the end of the file.

When you open a file for appending (that is, when the Type parameter is set to a), it is impossible to overwrite information already in the file.

If two separate processes open the same file for append, each process can write freely to the file without destroying the output being written by the other. The output from the two processes is intermixed in the order in which it is written to the file.

Note: If the data is buffered, it is not actually written until it is flushed.

The freopen and freopen64 subroutines first attempt to flush the stream and close any file descriptor associated with the Stream parameter. Failure to flush the stream or close the file descriptor is ignored.

The freopen and freopen64 subroutines substitute the named file in place of the open stream. The original stream is closed regardless of whether the subsequent open succeeds. The freopen and freopen64 subroutines returns a pointer to the FILE structure associated with the Stream parameter. The freopen and freopen64 subroutines is typically used to attach the pre-opened streams associated with standard input (stdin), standard output (stdout), and standard error (stderr) streams to other files.

The fdopen subroutine associates a stream with a file descriptor obtained from an openx subroutine, dup subroutine, creat subroutine, or pipe subroutine. These subroutines open files but do not return pointers to FILE structures. Many of the standard I/O package subroutines require pointers to FILE structures.

The Type parameter for the fdopen subroutine specifies the mode of the stream, such as r to open a file for reading, or a to open a file for appending (writing at the end of the file). The mode value of the Type parameter specified with the fdopen subroutine must agree with the mode of the file specified when the file was originally opened or created.

Note: Using the fdopen subroutine with a file descriptor obtained from a call to the shm_open subroutine must be avoided and might result in an error on the next fread, fwrite or fflush call.

The largest value that can be represented correctly in an object of type off_t will be established as the offset maximum in the open file description.
Parameters

Path
Points to a character string that contains the name of the file to be opened.

Type
Points to a character string that has one of the following values:

- **r**: Opens a text file for reading.
- **w**: Creates a new text file for writing, or opens and truncates a file to 0 length.
- **a**: Appends (opens a text file for writing at the end of the file, or creates a file for writing).
- **rb**: Opens a binary file for reading.
- **wb**: Creates a binary file for writing, or opens and truncates a file to 0.
- **ab**: Appends (opens a binary file for writing at the end of the file, or creates a file for writing).
- **r+**: Opens a file for update (reading and writing).
- **w+**: Truncates or creates a file for update.
- **a+**: Appends (opens a text file for writing at end of file, or creates a file for writing).
- **r+b , rb+**: Opens a binary file for update (reading and writing).
- **w+b , wb+**: Creates a binary file for update, or opens and truncates a file to 0 length.
- **a+b , ab+**: Appends (opens a binary file for update, writing at the end of the file, or creates a file for writing).

**Note:** The operating system does not distinguish between text and binary files. The **b** value in the **Type** parameter value is ignored.

Stream
Specifies the input stream.

File Descriptor
Specifies a valid open file descriptor.

Return Values
If the **fdopen**, **fopen**, **fopen64**, **freopen** or **freopen64** subroutine is unsuccessful, a null pointer is returned and the **errno** global variable is set to indicate the error.

Error Codes
The **fopen**, **fopen64**, **freopen** and **freopen64** subroutines are unsuccessful if the following is true:

- **EACCESS**: Search permission is denied on a component of the path prefix, the file exists and the permissions specified by the mode are denied, or the file does not exist and write permission is denied for the parent directory of the file to be created.
- **ELOOP**: Too many symbolic links were encountered in resolving path.
- **EINTR**: A signal was received during the process.
- **EISDIR**: The named file is a directory and the process does not have write access to it.
- **ENAMETOOLONG**: The length of the filename exceeds **PATH_MAX** or a pathname component is longer than **NAME_MAX**.
- **EMFILE**: The maximum number of files allowed are currently open.
- **ENOENT**: The named file does not exist or the **File Descriptor** parameter points to an empty string.
- **ENOSPC**: The file is not yet created and the directory or file system to contain the new file cannot be expanded.
- **ENOTDIR**: A component of the path prefix is not a directory.
- **ENXIO**: The named file is a character- or block-special file, and the device associated with this special file does not exist.
The named file is a regular file and the size of the file cannot be represented correctly in an object of type off_t.

The named file resides on a read-only file system and does not have write access.

The file is a pure-procedure (shared-text) file that is being executed and the process does not have write access.

The `fdopen`, `fopen`, `fopen64`, `freopen` and `freopen64` subroutines are unsuccessful if the following is true:

- **EINVAL** The value of the `Type` argument is not valid.
- **EINVAL** The value of the `mode` argument is not valid.
- **EMFILE** `FOPEN_MAX` streams are currently open in the calling process.
- **EMFILE** `STREAM_MAX` streams are currently open in the calling process.
- **ENAMETOOLONG** Pathname resolution of a symbolic link produced an intermediate result whose length exceeds `PATH_MAX`.
- **ENOMEM** Insufficient storage space is available.

The `freopen` and `fopen` subroutines are unsuccessful if the following is true:

- **EOVERFLOW** The named file is a size larger than 2 Gigabytes.

The `fdopen` subroutine is unsuccessful if the following is true:

- **EBADF** The value of the `File Descriptor` parameter is not valid.

**POSIX**

- **w** Truncates to 0 length or creates text file for writing.
- **w+** Truncates to 0 length or creates text file for update.
- **a** Opens or creates text file for writing at end of file.
- **a+** Opens or creates text file for update, writing at end of file.

**SAA**

At least eight streams, including three standard text streams, can open simultaneously. Both binary and text modes are supported.

**Related Information**

The `fclose` or `fflush` subroutine, `fseek`, `fseeko`, `fseeko64`, `rewind`, `ftell`, `ftello`, `ftello64`, `fgetpos`, `fgetpos64` or `fsetpos` subroutine, `setbuf`, `setvbuf`, `setbuffer`, or `setlinebuf` subroutine.

The [Input and Output Handling] in AIX 5L Version 5.3 General Programming Concepts: Writing and Debugging Programs.

**fork, f_fork, or vfork Subroutine**

**Purpose**

Creates a new process.
Libraries
fork, f_fork, and vfork: Standard C Library (libc.a)

Syntax
#include <unistd.h>
pid_t fork(void)
pid_t f_fork(void)
int vfork(void)

Description
The fork subroutine creates a new process. The new process (child process) is an almost exact copy of the calling process (parent process). The child process inherits the following attributes from the parent process:
• Environment
• Close-on-exec flags (described in the exec subroutine)
• Signal handling settings (such as the SIG_DFL value, the SIG_IGN value, and the Function Address parameter)
• Set user ID mode bit
• Set group ID mode bit
• Profiling on and off status
• Nice value
• All attached shared libraries
• Process group ID
• tty group ID (described in the exit subroutine)
• Current directory
• Root directory
• File-mode creation mask (described in the umask subroutine)
• File size limit (described in the ulimit subroutine)
• Attached shared memory segments (described in the shmat subroutine)
• Attached mapped file segments (described in the shmat subroutine)
• Debugger process ID and multiprocess flag if the parent process has multiprocess debugging enabled (described in the ptrace subroutine).

The child process differs from the parent process in the following ways:
• The child process has only one user thread; it is the one that called the fork subroutine.
• The child process has a unique process ID.
• The child process ID does not match any active process group ID.
• The child process has a different parent process ID.
• The child process has its own copy of the file descriptors for the parent process. However, each file descriptor of the child process shares a common file pointer with the corresponding file descriptor of the parent process.
• All semadj values are cleared. For information about semadj values, see the semop subroutine.
• Process locks, text locks, and data locks are not inherited by the child process. For information about locks, see the plock subroutine.
• If multiprocess debugging is turned on, the trace flags are inherited from the parent; otherwise, the trace flags are reset. For information about request 0, see the ptrace subroutine.

• The child process utime, stime, cutime, and cstime subroutines are set to 0. (For more information, see the getrusage subroutine.)

• Any pending alarms are cleared in the child process. (For more information, see the incinterval, setitimer, alarm, ualarm, getitimer or setitimer subroutine.)

• The set of signals pending for the child process is initialized to the empty set.

• The child process can have its own copy of the message catalogue for the parent process.

• The set of signals pending for the child process is initialized as an empty set.

Attention: If you are using the fork or vfork subroutines with an Enhanced X-Windows, X Toolkit, or Motif application, open a separate display connection (socket) for the forked process. If the child process uses the same display connection as the parent, the X Server will not be able to interpret the resulting data.

The f_fork subroutine is similar to fork, except for:
• It is required that the child process calls one of the exec functions immediately after it is created. Since the fork handlers are never called, the application data, mutexes and the locks are all undefined in the child process.

The vfork subroutine is supported as a compatibility interface for older Berkeley Software Distribution (BSD) system programs and can be used by compiling with the Berkeley Compatibility Library (libbsd.a).

In the Version 4 of the operating system, the parent process does not have to wait until the child either exits or executes, as it does in BSD systems. The child process is given a new address space, as in the fork subroutine. The child process does not share any parent address space.

Attention: When using the fork or vfork subroutines with an Enhanced X-Windows, X Toolkit, or Motif application, a separate display connection (socket) should be opened for the forked process. The child process should never use the same display connection as the parent. Display connections are embodied with sockets, and sockets are inherited by the child process. Any attempt to have multiple processes writing to the same display connection results in the random interleaving of X protocol packets at the word level. The resulting data written to the socket will not be valid or undefined X protocol packets, and the X Server will not be able to interpret it.

Attention: Although the fork and vfork subroutine may be used with Graphics Library applications, the child process must not make any additional Graphics Library subroutine calls. The child application inherits some, but not all of the graphics hardware resources of the parent. Drawing by the child process may hang the graphics adapter, the Enhanced X Server, or may cause unpredictable results and place the system into an unpredictable state.

For additional information, see the /usr/lpp/GL/README file.

Return Values

Upon successful completion, the fork subroutine returns a value of 0 to the child process and returns the process ID of the child process to the parent process. Otherwise, a value of -1 is returned to the parent process, no child process is created, and the errno global variable is set to indicate the error.
Error Codes

The **fork** subroutine is unsuccessful if one or more of the following are true:

- **EAGAIN**
  Exceeds the limit on the total number of processes running either systemwide or by a single user, or the system does not have the resources necessary to create another process.

- **ENOMEM**
  Not enough space exists for this process.

- **EPROCLIM**
  If WLM is running, the limit on the number of processes or threads in the class may have been met.

Related Information

The “getinterval, incinterval, absinterval, resinc, resabs, alarm, ualarm, gettimer or settimer Subroutine” on page 382, “bindprocessor Subroutine” on page 120, “exec: execl, execle, execlp, execv, execve, execvp, or exec Subroutine” on page 235, “exit, atexit, unatexit, exit, or Exit Subroutine” on page 242, “getusage, getsusage64, times, or vtimes Subroutine” on page 423, “getinterval, incinterval, absinterval, resinc, resabs, alarm, ualarm, gettimer or settimer Subroutine” on page 382, “getpriority, setpriority, or nice Subroutine” on page 407, “lock Subroutine” on page 1015, “pthread_atfork Subroutine” on page 1188, “ptrace, ptracex, ptrace64 Subroutine” on page 1286, “raise subroutine, semop subroutine, getinterval, incinterval, absinterval, resinc, resabs, alarm, ualarm, gettimer or settimer Subroutine” on page 382, “shmat subroutine, setpriority or getpriority” (“getpriority, setpriority, or nice Subroutine” on page 407) subroutine, “sigaction, sigvec, or signal subroutine, ulimit subroutine, umask subroutine, wait, waitpid, or wait3 subroutine.

The “posix_spawn or posix_spawnp Subroutine” on page 1129.

Subroutines Overview in AIX 5L Version 5.3 General Programming Concepts: Writing and Debugging Programs.

Process Duplication and Termination in AIX 5L Version 5.3 General Programming Concepts: Writing and Debugging Programs

LK provides more information about forking a multi-threaded process.

**fp_any_enable, fp_is_enabled, fp_enable_all, fp_enable, fp_disable_all, or fp_disable Subroutine**

**Purpose**

These subroutines allow operations on the floating-point trap control.

**Library**

Standard C Library (libc.a)

**Syntax**

```c
#include <fptrap.h>

int fp_any_enable()
int fp_is_enabled( Mask)
fptrap_t Mask;
void fp_enable_all()
void fp_enable(Mask)
fptrap_t Mask;
void fp_disable_all()
void fp_disable(Mask)
fptrap_t Mask;
```
Description
Floating point traps must be enabled before traps can be generated. These subroutines aid in manipulating floating-point traps and identifying the trap state and type.

In order to take traps on floating point exceptions, the `fp_trap` subroutine must first be called to put the process in serialized state, and the `fp_enable` subroutine or `fp_enable_all` subroutine must be called to enable the appropriate traps.

The header file `fptrap.h` defines the following names for the individual bits in the floating-point trap control:

<table>
<thead>
<tr>
<th>Bit Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>TRP_INVALID</td>
<td>Invalid Operation Summary</td>
</tr>
<tr>
<td>TRP_DIV_BY_ZERO</td>
<td>Divide by Zero</td>
</tr>
<tr>
<td>TRP_OVERFLOW</td>
<td>Overflow</td>
</tr>
<tr>
<td>TRP_UNDERFLOW</td>
<td>Underflow</td>
</tr>
<tr>
<td>TRP_INEXACT</td>
<td>Inexact Result</td>
</tr>
</tbody>
</table>

Parameters

`Mask` A 32-bit pattern that identifies floating-point traps.

Return Values

The `fp_any_enable` subroutine returns 1 if any floating-point traps are enabled. Otherwise, 0 is returned.

The `fp_is_enabled` subroutine returns 1 if the floating-point traps specified by the `Mask` parameter are enabled. Otherwise, 0 is returned.

The `fp_enable_all` subroutine enables all floating-point traps.

The `fp_enable` subroutine enables all floating-point traps specified by the `Mask` parameter.

The `fp_disable_all` subroutine disables all floating-point traps.

The `fp_disable` subroutine disables all floating-point traps specified by the `Mask` parameter.

Related Information

The `fp_clr_flag`, `fp_set_flag`, `fp_read_flag`, `fp_swap_flag` subroutine, `fp_invalid_op`, `fp_divbyzero`, `fp_overflow`, `fp_underflow`, `fp_inexact`, `fp_any_xcp` subroutine, `fp_iop_snan`, `fp_iop_infsinf`, `fp_iop_infdinf`, `fp_iop_zrdzr`, `fp_iop_infmzr`, `fp_iop_invcmp`, `fp_read_rnd`, and `fp_swap_rnd` subroutine. 

Floating-Point Processor in Assembler Language Reference.

Subroutines Overview in AIX 5L Version 5.3 General Programming Concepts: Writing and Debugging Programs.
**fp_clr_flag, fp_set_flag, fp_read_flag, or fp_swap_flag Subroutine**

**Purpose**
Allows operations on the floating-point exception flags.

**Library**
Standard C Library (libc.a)

**Syntax**
```c
#include <float.h>
#include <fpexc.h>

void fp_clr_flag(Mask);
fpflag_t Mask;
void fp_set_flag(Mask);
fpflag_t Mask;
fpflag_t fp_read_flag();
fpflag_t fp_swap_flag(Mask);
fpflag_t Mask;
```

**Description**
These subroutines aid in determining both when an exception has occurred and the exception type. These subroutines can be called explicitly around blocks of code that may cause a floating-point exception.

According to the *IEEE Standard for Binary Floating-Point Arithmetic*, the following types of floating-point operations must be signaled when detected in a floating-point operation:

- Invalid operation
- Division by zero
- Overflow
- Underflow
- Inexact

An invalid operation occurs when the result cannot be represented (for example, a `sqrt` operation on a number less than 0).

The *IEEE Standard for Binary Floating-Point Arithmetic* states: "For each type of exception, the implementation shall provide a status flag that shall be set on any occurrence of the corresponding exception when no corresponding trap occurs. It shall be reset only at the user's request. The user shall be able to test and to alter the status flags individually, and should further be able to save and restore all five at one time."

Floating-point operations can set flags in the floating-point exception status but cannot clear them. Users can clear a flag in the floating-point exception status using an explicit software action such as the `fp_swap_flag(0)` subroutine.

The `fpexc.h` file defines the following names for the flags indicating floating-point exception status:

- `FP_INVALID`: Invalid operation summary
- `FP_OVERFLOW`: Overflow
- `FP_UNDERFLOW`: Underflow
- `FP_DIV_BY_ZERO`: Division by 0
- `FP_INEXACT`: Inexact result
In addition to these flags, the operating system supports additional information about the cause of an invalid operation exception. The following flags also indicate floating-point exception status and defined in the `fpexc.h` file. The flag number for each exception type varies, but the mnemonics are the same for all ports. The following invalid operation detail flags are not required for conformance to the IEEE floating-point exceptions standard:

- **FP_INV_SNAN**: Signaling NaN
- **FP_INV_ISI**: INF - INF
- **FP_INV_IDI**: INF / INF
- **FP_INV_ZDZ**: 0 / 0
- **FP_INV_QM**: INF x 0
- **FP_INV_CMP**: Unordered compare
- **FP_INV_SQRT**: Square root of a negative number
- **FP_INV_CVI**: Conversion to integer error
- **FP_INV_VXSOFT**: Software request

**Parameters**

*Mask*  
A 32-bit pattern that identifies floating-point exception flags.

**Return Values**

The `fp_clr_flag` subroutine resets the exception status flags defined by the *Mask* parameter to 0 (false). The remaining flags in the exception status are unchanged.

The `fp_set_flag` subroutine sets the exception status flags defined by the *Mask* parameter to 1 (true). The remaining flags in the exception status are unchanged.

The `fp_read_flag` subroutine returns the current floating-point exception status. The flags in the returned exception status can be tested using the flag definitions above. You can test individual flags or sets of flags.

The `fp_swap_flag` subroutine writes the *Mask* parameter into the floating-point status and returns the floating-point exception status from before the write.

Users set or reset multiple exception flags using `fp_set_flag` and `fp_clr_flag` by ANDing or ORing definitions for individual flags. For example, the following resets both the overflow and inexact flags:

```
fp_clr_flag (FP_OVERFLOW | FP_INEXACT)
```

**Related Information**

The `fp_any_enable`, `fp_is_enabled`, `fp_enable_all`, `fp_enable`, `fp_disable`, or `fp_disable_all` subroutine, `fp_xcp`, or `fp_divbyzero` subroutine, `fp_inexact`, `fp_invalid_op`, `fp_overflow`, `fp_underflow` subroutine, `fp_iop_infdinf`, `fp_iop_infmZR`, `fp_iop_infsinf`, `fp_iop_invcmp`, or `fp_iop_snan`, or `fp_iop_zrdzr` subroutine, `fp_read_rnd` or `fp_swap_rnd` subroutine.

Refer to Floating-Point Exceptions Overview and Subroutines Overview in AIX 5L Version 5.3 General Programming Concepts: Writing and Debugging Programs.
fp_cpusync Subroutine

Purpose
Queries or changes the floating-point exception enable (FE) bit in the Machine Status register (MSR).

Note: This subroutine has been replaced by the fp_trapstate subroutine. The fp_cpusync subroutine is supported for compatibility, but the fp_trapstate subroutine should be used for development.

Library
Standard C Library (libc.a)

Syntax
#include <fptrap.h>

int fp_cpusync (Flag);
int Flag;

Description
The fp_cpusync subroutine is a service routine used to query, set, or reset the Machine Status Register (MSR) floating-point exception enable (FE) bit. The MSR FE bit determines whether a processor runs in pipeline or serial mode. Floating-point traps can only be generated by the hardware when the processor is in synchronous mode.

The fp_cpusync subroutine changes only the MSR FE bit. It is a service routine for use in developing custom floating-point exception-handling software. If you are using the fp_enable or fp_enable_all ("fp_any_enable, fp_is_enabled, fp_enable_all, fp_enable, fp_disable_all, or fp_disable Subroutine" on page 290) subroutine or the fp_sh_trap_info or fp_sh_set_stat ("fp_sh_info, fp_sh_trap_info, or fp_sh_set_stat Subroutine" on page 300) subroutine, you must use the fp_trap ("fp_trap Subroutine" on page 302) subroutine to place the process in serial mode.

Parameters
Flag Specifies to query or modify the MSR FE bit:

FP_SYNC_OFF
Sets the FE bit in the MSR to Off, which disables floating-point exception processing immediately.

FP_SYNC_ON
Sets the FE bit in the MSR to On, which enables floating-exception processing for the next floating-point operation.

FP_SYNC_QUERY
Returns the current state of the process (either FP_SYNC_ON or FP_SYNC_OFF) without modifying it.

If called with any other value, the fp_cpusync subroutine returns FP_SYNC_ERROR.

Return Values
If called with the FP_SYNC_OFF or FP_SYNC_ON flag, the fp_cpusync subroutine returns a value indicating which flag was in the previous state of the process.
If called with the `FP_SYNC_QUERY` flag, the `fp_cpusync` subroutine returns a value indicating the current state of the process, either the `FP_SYNC_OFF` or `FP_SYNC_ON` flag.

**Error Codes**

If the `fp_cpusync` subroutine is called with an invalid parameter, the subroutine returns `FP_SYNC_ERROR`. No other errors are reported.

**Related Information**

The `fp_any_enable`, `fp_is_enabled`, `fp_enable_all`, `fp_enable`, `fp_disable_all`, or `fp_disable` subroutine, `fp_clr_flag`, `fp_set_flag`, `fp_read_flag`, or `fp_swap_flag` subroutine, `fp_read_flag`, or `fp_swap_flag` subroutine, `fp_cpusync` subroutine, `fp_trap` subroutine, `sigaction`, `sigvec`, or `signal` subroutine.

Floating-Point Processor inAssembler Language Reference.

Floating-Point Exceptions in AIX 5L Version 5.3 General Programming Concepts: Writing and Debugging Programs.

**fp_flush_imprecise Subroutine**

**Purpose**

Forces imprecise signal delivery.

**Library**

Standard C Library (`libc.a`)

**Syntax**

```c
#include <fptrap.h>
void fp_flush_imprecise ()
```

**Description**

The `fp_flush_imprecise` subroutine forces any imprecise interrupts to be reported. To ensure that no signals are lost when a program voluntarily exits, use this subroutine in combination with the `atexit` subroutine.

**Example**

The following example illustrates using the `atexit` subroutine to run the `fp_flush_imprecise` subroutine before a program exits:

```c
#include <fptrap.h>
#include <stdlib.h>
#include <stdio.h>
if (0!=atexit(fp_flush_imprecise))
    puts("Failure in atexit(fp_flush_imprecise) ");
```

**Related Information**

The `atexit` subroutine, `fp_any_enable`, `fp_is_enabled`, `fp_enable_all`, `fp_enable`, `fp_disable_all`, or `fp_disable` subroutine, `fp_clr_flag`, `fp_read_flag`, `fp_swap_flag`, or `fp_set_flag` subroutine, `fp_read_flag`, or `fp_swap_flag` subroutine, `fp_cpuSync` subroutine, `fp_trap` subroutine, `sigaction` subroutine.
Floating-Point Exceptions in AIX 5L Version 5.3 General Programming Concepts: Writing and Debugging Programs.

fp_invalid_op, fp_divbyzero, fp_overflow, fp_underflow, fp_inexact, fp_any_xcp Subroutine

Purpose
Tests to see if a floating-point exception has occurred.

Library
Standard C Library (libc.a)

Syntax

```c
#include <float.h>
#include <fpxcp.h>

int fp_invalid_op()
int fp_divbyzero()
int fp_overflow()
int fp_underflow()
int fp_inexact()
int fp_any_xcp()
```

Description
These subroutines aid in determining when an exception has occurred and the exception type. These subroutines can be called explicitly after blocks of code that may cause a floating-point exception.

Return Values
The `fp_invalid_op` subroutine returns a value of 1 if a floating-point invalid-operation exception status flag is set. Otherwise, a value of 0 is returned.

The `fp_divbyzero` subroutine returns a value of 1 if a floating-point divide-by-zero exception status flag is set. Otherwise, a value of 0 is returned.

The `fp_overflow` subroutine returns a value of 1 if a floating-point overflow exception status flag is set. Otherwise, a value of 0 is returned.

The `fp_underflow` subroutine returns a value of 1 if a floating-point underflow exception status flag is set. Otherwise, a value of 0 is returned.

The `fp_inexact` subroutine returns a value of 1 if a floating-point inexact exception status flag is set. Otherwise, a value of 0 is returned.

The `fp_any_xcp` subroutine returns a value of 1 if a floating-point invalid operation, divide-by-zero, overflow, underflow, or inexact exception status flag is set. Otherwise, a value of 0 is returned.

Related Information
The `fp_any_enable, fp_is_enabled, fp_enable_all, fp_enable fp_disable_all, or fp_disable Subroutine` on page 290 subroutine, `fp_clr_flag, fp_read_flag, fp_set_flag, or fp_swap_flag` subroutine.
Subroutines

Purpose
Tests to see if a floating-point exception has occurred.

Library
Standard C Library (libc.a)

Syntax
```
#include <float.h>
#include <fpxcp.h>
int fp_iop_snan()
int fp_iop_infsinf()
int fp_iop_infdinf()
int fp_iop_zrdzr()
int fp_iop_infmzr()
int fp_iop_invcmp()
int fp_iop_sqrt()
int fp_iop_convert()
int fp_iop_vxsoft ()
```

Description
These subroutines aid in determining when an exception has occurred and the exception type. These subroutines can be called explicitly after blocks of code that may cause a floating-point exception.

Return Values
The `fp_iop_snan` subroutine returns a value of 1 if a floating-point invalid-operation exception status flag is set due to a signaling NaN (NaN) flag. Otherwise, a value of 0 is returned.

The `fp_iop_infsinf` subroutine returns a value of 1 if a floating-point invalid-operation exception status flag is set due to an INF-INF flag. Otherwise, a value of 0 is returned.

The `fp_iop_infdinf` subroutine returns a value of 1 if a floating-point invalid-operation exception status flag is set due to an INF/INF flag. Otherwise, a value of 0 is returned.

The `fp_iop_zrdzr` subroutine returns a value of 1 if a floating-point invalid-operation exception status flag is set due to a 0.0/0.0 flag. Otherwise, a value of 0 is returned.
The `fp_iop_infmzr` subroutine returns a value of 1 if a floating-point invalid-operation exception status flag is set due to an INF\(^\ast\)0.0 flag. Otherwise, a value of 0 is returned.

The `fp_iop_invcmp` subroutine returns a value of 1 if a floating-point invalid-operation exception status flag is set due to a compare involving a NaN. Otherwise, a value of 0 is returned.

The `fp_iop_sqrt` subroutine returns a value of 1 if a floating-point invalid-operation exception status flag is set due to the calculation of a square root of a negative number. Otherwise, a value of 0 is returned.

The `fp_iop_convert` subroutine returns a value of 1 if a floating-point invalid-operation exception status flag is set due to the conversion of a floating-point number to an integer, where the floating-point number was a NaN, an INF, or was outside the range of the integer. Otherwise, a value of 0 is returned.

The `fp_iop_vxsoft` subroutine returns a value of 1 if the VXSOFT detail bit is on. Otherwise, a value of 0 is returned.

---

**fp_raise_xcp Subroutine**

**Purpose**
Generates a floating-point exception.

**Library**
Standard C Library (`libc.a`)

**Syntax**
```
#include <fpxcp.h>

int fp_raise_xcp(fpflag_t mask);
```

**Description**
The `fp_raise_xcp` subroutine causes any floating-point exceptions defined by the `mask` parameter to be raised immediately. If the exceptions defined by the `mask` parameter are enabled and the program is running in serial mode, the signal for floating-point exceptions, `SIGFPE`, is raised.

If more than one exception is included in the `mask` variable, the exceptions are raised in the following order:
1. Invalid
2. Dividebyzero
3. Underflow
4. Overflow
5. Inexact

Thus, if the user exception handler does not disable further exceptions, one call to the `fp_raise_xcp` subroutine can cause the exception handler to be entered many times.

**Parameters**
- `mask` Specifies a 32-bit pattern that identifies floating-point traps.
Return Values

The **fp_raise_xcp** subroutine returns 0 for normal completion and returns a nonzero value if an error occurs.

Related Information

The **fp_any_enable, fp_is_enabled, fp_enable_all, fp_enable, fp_disable_all, or fp_disable** subroutine, **fp_clr_flag, fp_read_flag, fp_swap_flag, or fpset_flag** subroutine, **fp_cpusync** subroutine, **fp_trap** subroutine, **sigaction** subroutine.

---

**fp_read_rnd or fp_swap_rnd Subroutine**

**Purpose**

Read and set the IEEE floating-point rounding mode.

**Library**

Standard C Library (**libc.a**)

**Syntax**

```c
#include <float.h>

fprnd_t fp_read_rnd()
fprnd_t fp_swap_rnd(RoundMode)
fprnd_t RoundMode;
```

**Description**

The **fp_read_rnd** subroutine returns the current rounding mode. The **fp_swap_rnd** subroutine changes the rounding mode to the **RoundMode** parameter and returns the value of the rounding mode before the change.

Floating-point rounding occurs when the infinitely precise result of a floating-point operation cannot be represented exactly in the destination floating-point format (such as double-precision format).

The **IEEE Standard for Binary Floating-Point Arithmetic** allows floating-point numbers to be rounded in four different ways: round toward zero, round to nearest, round toward +INF, and round toward -INF. Once a rounding mode is selected it affects all subsequent floating-point operations until another rounding mode is selected.

**Note:** The default floating-point rounding mode is round to nearest. All C main programs begin with the rounding mode set to round to nearest.

The encodings of the rounding modes are those defined in the **ANSI C Standard**. The **float.h** file contains definitions for the rounding modes. Below is the **float.h** definition, the **ANSI C Standard** value, and a description of each rounding mode.

<table>
<thead>
<tr>
<th>float.h Definition</th>
<th>ANSI Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>FP_RND_RZ</td>
<td>0</td>
<td>Round toward 0</td>
</tr>
<tr>
<td>FP_RND_RN</td>
<td>1</td>
<td>Round to nearest</td>
</tr>
<tr>
<td>FP_RND_RP</td>
<td>2</td>
<td>Round toward +INF</td>
</tr>
<tr>
<td>FP_RND_RM</td>
<td>3</td>
<td>Round toward -INF</td>
</tr>
</tbody>
</table>
The **fp_swap_rnd** subroutine can be used to swap rounding modes by saving the return value from **fp_swap_rnd**(RoundMode). This can be useful in functions that need to force a specific rounding mode for use during the function but wish to restore the caller's rounding mode on exit. Below is a code fragment that accomplishes this action:

```c
save_mode = fp_swap_rnd(new_mode);
....desired code using new_mode
(void) fp_swap_rnd(save_mode); /*restore caller's mode*/
```

**Parameters**

*RoundMode* Specifies one of the following modes: **FP_RND_RZ**, **FP_RND_RN**, **FP_RND_RP**, or **FP_RND_RM**.

**Related Information**

The **floor**, **ceil**, **nearest**, **trunc**, **rint**, **itrunc**, **uitrunc**, **fmod**, or **fabs** subroutine, **fp_any_enable**, **fp_is_enabled**, **fp_enable_all**, **fp_disable_all**, or **fp_disable** subroutine, **fp_clr_flag**, **fp_set_flag**, **fp_read_flag**, or **fp_swap_flag** subroutine.

**Subroutines Overview** in AIX 5L Version 5.3 General Programming Concepts: Writing and Debugging Programs.

---

### fp_sh_info, fp_sh_trap_info, or fp_sh_set_stat Subroutine

**Purpose**

From within a floating-point signal handler, determines any floating-point exception that caused the trap in the process and changes the state of the Floating-Point Status and Control register (FPSCR) in the user process.

**Library**

Standard C Library (**libc.a**)

**Syntax**

```c
#include <fpxcp.h>
#include <fptrap.h>
#include <signal.h>

void fp_sh_info( scp, fcp, struct_size)
struct sigcontext  *scp;
struct fp_sh_info *fcp;
size_t struct_size;

void fp_sh_trap_info( scp, fcp)
struct sigcontext  *scp;
struct fp_ctx *fcp;

void fp_sh_set_stat( scp, fpscr)
struct sigcontext *scp;
fpstat_t fpscr;
```
Description

These subroutines are for use within a user-written signal handler. They return information about the process that was running at the time the signal occurred, and they update the Floating-Point Status and Control register for the process.

**Note:** The `fp_sh_trap_info` subroutine is maintained for compatibility only. It has been replaced by the `fp_sh_info` subroutine, which should be used for development.

These subroutines operate only on the state of the user process that was running at the time the signal was delivered. They read and write the `sigcontext` structure. They do not change the state of the signal handler process itself.

The state of the signal handler process can be modified by the `fp_any_enable`, `fp_is_enabled`, `fp_enable_all`, `fp_enable`, `fp_disable_all`, or `fp_disable` subroutine.

**fp_sh_info**

The `fp_sh_info` subroutine returns information about the process that caused the trap by means of a floating-point context (`fp_sh_info`) structure. This structure contains the following information:

```c
typedef struct fp_sh_info {
    fstat_t fpscr;
    fflag_t trap;
    short trap_mode;
    char flags;
    char extra;
} fp_sh_info_t;
```

The fields are:

- **fpscr**
  The Floating-Point Status and Control register (FPSCR) in the user process at the time the interrupt occurred.

- **trap**
  A mask indicating the trap or traps that caused the signal handler to be entered. This mask is the logical OR operator of the enabled floating-point exceptions that occurred to cause the trap. This mask can have up to two exceptions; if there are two, the `INEXACT` signal must be one of them. If the mask is 0, the `SIGFPE` signal was raised not by a floating-point operation, but by the `kill` or `raise` subroutine or the `kill` command.

- **trap_mode**
  The trap mode in effect in the process at the time the signal handler was entered. The values returned in the `fp_sh_info.trap_mode` file use the following argument definitions:

  - **FP_TRAP_OFF**
    Trapping off
  - **FP_TRAP_SYNC**
    Precise trapping on
  - **FP_TRAP_IMP_REC**
    Recoverable imprecise trapping on
  - **FP_TRAP_IMP**
    Non-recoverable imprecise trapping on

- **flags**
  This field is interpreted as an array of bits and should be accessed with masks. The following mask is defined:

  - **FP_IAR_STAT**
    If the value of the bit at this mask is 1, the exception was precise and the IAR points to the instruction that caused the exception. If the value bit at this mask is 0, the exception was imprecise.
The \textit{fp\_sh\_trap\_info} subroutine is maintained for compatibility only. The \textit{fp\_sh\_trap\_info} subroutine returns information about the process that caused the trap by means of a floating-point context (\textit{fp\_ctx}) structure. This structure contains the following information:

\begin{verbatim}
fpstat_t fpscr;
fpflag_t trap;
\end{verbatim}

The fields are:

- \textbf{fpscr} is the Floating-Point Status and Control register (FPSCR) in the user process at the time the interrupt occurred.
- \textbf{trap} is a mask indicating the trap or traps that caused the signal handler to be entered. This mask is the logical OR operator of the enabled floating-point exceptions that occurred to cause the trap. This mask can have up to two exceptions; if there are two, the \texttt{INEXACT} signal must be one of them. If the mask is 0, the \texttt{SIGFPE} signal was raised not by a floating-point operation, but by the \texttt{kill} or \texttt{raise} subroutine or the \texttt{kill} command.

The \textit{fp\_sh\_set\_stat} subroutine updates the Floating-Point Status and Control register (FPSCR) in the user process with the value in the \textit{fpscr} field.

The signal handler must either clear the exception bit that caused the trap to occur or disable the trap to prevent a recurrence. If the instruction generated more than one exception, and the signal handler clears only one of these exceptions, a signal is raised for the remaining exception when the next floating-point instruction is executed in the user process.

### Parameters

- \textit{fcp} specifies a floating-point context structure.
- \textit{scp} specifies a \texttt{sigcontext} structure for the interrupt.
- \textit{struct\_size} specifies the size of the \textit{fp\_sh\_info} structure.
- \textit{fpscr} specifies which Floating-Point Status and Control register to update.

### Related Information

The \texttt{fp\_any\_enable}, \texttt{fp\_disable\_all}, \texttt{fp\_disable}, \texttt{fp\_enable\_all}, \texttt{fp\_enable}, or \texttt{fp\_is\_enabled} subroutine, \texttt{fp\_clr\_flag}, \texttt{fp\_read\_flag}, \texttt{fp\_set\_flag}, or \texttt{fp\_swap\_flag} subroutine, \texttt{fp\_trap} subroutine.

[Floating-Point Exceptions] in \textit{AIX 5L Version 5.3 General Programming Concepts: Writing and Debugging Programs}.

### fp\_trap Subroutine

### Purpose

Queries or changes the mode of the user process to allow floating-point exceptions to generate traps.

### Library

Standard C Library (\texttt{libc.a})
Syntax

```c
#include <fptrap.h>

int fp_trap(int flag);
int flag;
```

Description

The `fp_trap` subroutine queries and changes the mode of the user process to allow or disallow floating-point exception trapping. Floating-point traps can only be generated when a process is executing in a traps-enabled mode.

The default state is to execute in pipelined mode and not to generate floating-point traps.

Note: The `fp_trap` routines only change the execution state of the process. To generate floating-point traps, you must also enable traps. Use the `fp_enable` [“fp_any_enable, fp_is_enabled, fp_enable_all, fp_enable, fp_disable_all, or fp_disable Subroutine” on page 290] and `fp_enable_all` subroutines to enable traps.

Before calling the `fp_trap(FP_TRAP_SYNC)` routine, previous floating-point operations can set to True certain exception bits in the Floating-Point Status and Control register (FPSCR). Enabling these exceptions and calling the `fp_trap(FP_TRAP_SYNC)` routine does not cause an immediate trap to occur. That is, the operation of these traps is edge-sensitive, not level-sensitive.

The `fp_trap` subroutine does not clear the exception history. You can query this history by using any of the following subroutines:

- `fp_any_xcp`
- `fp_divbyzero`
- `fp_iop_convert`
- `fp_iop_infdinf`
- `fp_iop_infmZR`
- `fp_iop_infsinf`
- `fp_iop_invcmp`
- `fp_iop_snan`
- `fp_iop_sqrt`
- `fp_iop_vxsoft`
- `fp_iop_zrdzr`
- `fp_inexact`
- `fp_invalid_op`
- `fp_overflow`
- `fp_underflow`
Parameters

flag Specifies a query of or change in the mode of the user process:

FP_TRAP_OFF
Puts the user process into trapping-off mode and returns the previous mode of the process,
either FP_TRAP_SYNC, FP_TRAP_IMP, FP_TRAP_IMP_REC, or FP_TRAP_OFF.

FP_TRAP_QUERY
Returns the current mode of the user process.

FP_TRAP_SYNC
Puts the user process into precise trapping mode and returns the previous mode of the
process.

FP_TRAP_IMP
Puts the user process into non-recoverable imprecise trapping mode and returns the previous
mode.

FP_TRAP_IMP_REC
Puts the user process into recoverable imprecise trapping mode and returns the previous
mode.

FP_TRAP_FASTMODE
Puts the user process into the fastest trapping mode available on the hardware platform.

Note: Some hardware models do not support all modes. If an unsupported mode is requested, the
fp_trap subroutine returns FP_TRAP_UNIMPL.

Return Values
If called with the FP_TRAP_OFF, FP_TRAP_IMP, FP_TRAP_IMP_REC, or FP_TRAP_SYNC flag, the
fp_trap subroutine returns a value indicating which flag was in the previous mode of the process if the
hardware supports the requested mode. If the hardware does not support the requested mode, the fp_trap
subroutine returns FP_TRAP_UNIMPL.

If called with the FP_TRAP_QUERY flag, the fp_trap subroutine returns a value indicating the current
mode of the process, either the FP_TRAP_OFF, FP_TRAP_IMP, FP_TRAP_IMP_REC, or
FP_TRAP_SYNC flag.

If called with FP_TRAP_FASTMODE, the fp_trap subroutine sets the fastest mode available and returns
the mode selected.

Error Codes
If the fp_trap subroutine is called with an invalid parameter, the subroutine returns FP_TRAP_ERROR.

If the requested mode is not supported on the hardware platform, the subroutine returns
FP_TRAP_UNIMPL.

fp_trapstate Subroutine

Purpose
Queries or changes the trapping mode in the Machine Status register (MSR).

Note: This subroutine replaces the fp_cpusync subroutine. The
fp_cpusync subroutine is supported for compatibility, but the fp_trapstate subroutine should be
used for development.
Library
Standard C Library (libc.a)

Syntax
#include <fptrap.h>
int fp_trapstate (int)

Description
The fp_trapstate subroutine is a service routine used to query or set the trapping mode. The trapping mode determines whether floating-point exceptions can generate traps, and can affect execution speed. See Floating-Point Exceptions Overview in AIX 5L Version 5.3 General Programming Concepts: Writing and Debugging Programs for a description of precise and imprecise trapping modes. Floating-point traps can be generated by the hardware only when the processor is in a traps-enabled mode.

The fp_trapstate subroutine changes only the trapping mode. It is a service routine for use in developing custom floating-point exception-handling software. If you are using the fp_enable (fp any enable, fp is enabled, fp_enable_all, fp_enable, fp disable_all, or fp disable Subroutine* on page 290) or fp_enable_all subroutine or the fp_sh_info (fp_sh_info, fp_sh_trap_info, or fp_sh_set_stat Subroutine” on page 300) or fp_sh_set_stat subroutine, you must use the fp_trap (“fp_trap Subroutine” on page 302) subroutine to change the process’ trapping mode.

Parameters

flag Specifies a query of, or change in, the trap mode:

FP_TRAPSTATE_OFF
Sets the trapping mode to Off and returns the previous mode.

FP_TRAPSTATE_QUERY
Returns the current trapping mode without modifying it.

FP_TRAPSTATE_IMP
Puts the process in non-recoverable imprecise trapping mode and returns the previous state.

FP_TRAPSTATE_IMP_REC
Puts the process in recoverable imprecise trapping mode and returns the previous state.

FP_TRAPSTATE_PRECISE
Puts the process in precise trapping mode and returns the previous state.

FP_TRAPSTATE_FASTMODE
Puts the process in the fastest trap-generating mode available on the hardware platform and returns the state selected.

Note: Some hardware models do not support all modes. If an unsupported mode is requested, the fp_trapstate subroutine returns FP_TRAP_UNIMPL and the trapping mode is not changed.

Return Values
If called with the FP_TRAPSTATE_OFF, FP_TRAPSTATE_IMP, FP_TRAPSTATE_IMP_REC, or FP_TRAPSTATE_PRECISE flag, the fp_trapstate subroutine returns a value indicating the previous mode of the process. The value may be FP_TRAPSTATE_OFF, FP_TRAPSTATE_IMP, FP_TRAPSTATE_IMP_REC, or FP_TRAPSTATE_PRECISE. If the hardware does not support the requested mode, the fp_trapstate subroutine returns FP_TRAP_UNIMPL.
If called with the `FP_TRAP_QUERY` flag, the `fp_trapstate` subroutine returns a value indicating the current mode of the process. The value may be `FP_TRAPSTATE_OFF`, `FP_TRAPSTATE_IMP`, `FP_TRAPSTATE_IMP_REC`, or `FP_TRAPSTATE_PRECISE`.

If called with the `FP_TRAPSTATE_FASTMODE` flag, the `fp_trapstate` subroutine returns a value indicating which mode was selected. The value may be `FP_TRAPSTATE_OFF`, `FP_TRAPSTATE_IMP`, `FP_TRAPSTATE_IMP_REC`, or `FP_TRAPSTATE_PRECISE`.

**Related Information**

The `fp_any_enable, fp_disable_all, fp_disable, fp_enable_all, fp_enable, or fp_is_enabled` subroutine, `fp_clr_flag, fp_read_flag, fpset_flag, or fp_swap_flag` subroutine, `sigaction, signal, or sigvec` subroutine.

The `Floating-Point Processor` in Assembler Language Reference.

`Floating-Point Exceptions` in AIX 5L Version 5.3 General Programming Concepts: Writing and Debugging Programs.

### fpclassify Macro

**Purpose**

Classifies real floating type.

**Syntax**

```
#include <math.h>

int fpclassify(real-floating x);
```

**Description**

The `fpclassify` macro classifies the `x` parameter as NaN, infinite, normal, subnormal, zero, or into another implementation-defined category. An argument represented in a format wider than its semantic type is converted to its semantic type. Classification is based on the type of the argument.

**Parameters**

- `x` Specifies the value to be classified.

**Return Values**

The `fpclassify` macro returns the value of the number classification macro appropriate to the value of its argument.

**Related Information**

`isinfinite Macro` on page 561, `isinf Subroutine` on page 563, `class, _class, finite, isnan, or unordered Subroutines` on page 167, `isnormal Macro` on page 565.


`math.h` in AIX 5L Version 5.3 Files Reference.
fread or fwrite Subroutine

Purpose
Reads and writes binary files.

Library
Standard C Library (libc.a)

Syntax
```
#include <stdio.h>
size_t fread ( void * Pointer, Size, NumberOfItems, Stream );
size_t fwrite ( Pointer, Size, NumberOfItems, Stream );
```

Description
The `fread` subroutine copies the number of data items specified by the `NumberOfItems` parameter from the input stream into an array beginning at the location pointed to by the `Pointer` parameter. Each data item has the form `* Pointer`.

The `fread` subroutine stops copying bytes if an end-of-file (EOF) or error condition is encountered while reading from the input specified by the `Stream` parameter, or when the number of data items specified by the `NumberOfItems` parameter have been copied. This subroutine leaves the file pointer of the `Stream` parameter, if defined, pointing to the byte following the last byte read. The `fread` subroutine does not change the contents of the `Stream` parameter.

The `fwrite` subroutine writes items from the array pointed to by the `Pointer` parameter to the stream pointed to by the `Stream` parameter. Each item’s size is specified by the `Size` parameter. The `fwrite` subroutine writes the number of items specified by the `NumberOfItems` parameter. The file-position indicator for the stream is advanced by the number of bytes successfully written. If an error occurs, the resulting value of the file-position indicator for the stream is indeterminate.

The `fwrite` subroutine appends items to the output stream from the array pointed to by the `Pointer` parameter. The `fwrite` subroutine appends as many items as specified in the `NumberOfItems` parameter.

The `fwrite` subroutine stops writing bytes if an error condition is encountered on the stream, or when the number of items of data specified by the `NumberOfItems` parameter have been written. The `fwrite` subroutine does not change the contents of the array pointed to by the `Pointer` parameter.

Note: The `fread` subroutine is a buffered `read` subroutine library call. It reads data in 4KB blocks. For tape block sizes greater than 4KB, use the `open` subroutine using a stream that returns data not supplied by a prior call to the `ungetc` subroutine.

The `fwrite` subroutine writes items from the array pointed to by the `Pointer` parameter to the stream pointed to by the `Stream` parameter. Each item’s size is specified by the `Size` parameter. The `fwrite` subroutine writes the number of items specified by the `NumberOfItems` parameter. The file-position indicator for the stream is advanced by the number of bytes successfully written. If an error occurs, the resulting value of the file-position indicator for the stream is indeterminate.

The `fwrite` subroutine appends items to the output stream from the array pointed to by the `Pointer` parameter. The `fwrite` subroutine appends as many items as specified in the `NumberOfItems` parameter.

The `fwrite` subroutine stops writing bytes if an error condition is encountered on the stream, or when the number of items of data specified by the `NumberOfItems` parameter have been written. The `fwrite` subroutine does not change the contents of the array pointed to by the `Pointer` parameter.
The st_ctime and st_mtime fields will be marked for update between the successful run of the fwrite subroutine and the next completion of a call to the fflush subroutine or the fclose subroutine on the same stream, the next call to the exit subroutine, or the next call to the abort subroutine.

**Parameters**

- **Pointer**
  - Points to an array.

- **Size**
  - Specifies the size of the variable type of the array pointed to by the Pointer parameter. The Size parameter can be considered the same as a call to sizeof subroutine.

- **NumberOfItems**
  - Specifies the number of items of data.

- **Stream**
  - Specifies the input or output stream.

**Return Values**

The fread and fwrite subroutines return the number of items actually transferred. If the NumberOfItems parameter contains a 0, no characters are transferred, and a value of 0 is returned. If the NumberOfItems parameter contains a negative number, it is translated to a positive number, since the NumberOfItems parameter is of the unsigned type.

**Error Codes**

If the fread subroutine is unsuccessful because the I/O stream is unbuffered or data needs to be read into the I/O stream's buffer, it returns one or more of the following error codes:

- **EAGAIN**
  - Indicates that the O_NONBLOCK flag is set for the file descriptor specified by the Stream parameter, and the process would be delayed in the fread operation.

- **EBADF**
  - Indicates that the file descriptor specified by the Stream parameter is not a valid file descriptor open for reading.

- **EINTR**
  - Indicates that the read operation was terminated due to receipt of a signal, and no data was transferred.

Note: Depending upon which library routine the application binds to, this subroutine may return EINTR. Refer to the signal subroutine regarding sa_restart.

- **EIO**
  - Indicates that the process is a member of a background process group attempting to perform a read from its controlling terminal, and either the process is ignoring or blocking the SIGTTIN signal or the process group has no parent process.

- **ENOMEM**
  - Indicates that insufficient storage space is available.

- **ENXIO**
  - Indicates that a request was made of a nonexistent device.

If the fwrite subroutine is unsuccessful because the I/O stream is unbuffered or the I/O stream's buffer needs to be flushed, it returns one or more of the following error codes:

- **EAGAIN**
  - Indicates that the O_NONBLOCK or O_NDELAY flag is set for the file descriptor specified by the Stream parameter, and the process is delayed in the write operation.

- **EBADF**
  - Indicates that the file descriptor specified by the Stream parameter is not a valid file descriptor open for writing.

- **EFBIG**
  - Indicates that an attempt was made to write a file that exceeds the file size of the process limit or the systemwide maximum file size.

- **EINTR**
  - Indicates that the write operation was terminated due to the receipt of a signal, and no data was transferred.

- **EIO**
  - Indicates that the process is a member of a background process group attempting to perform a write to its controlling terminal, the TOSTOP signal is set, the process is neither ignoring nor blocking the SIGTTOU signal, and the process group of the process is orphaned.
ENOSPC  Indicates that there was no free space remaining on the device containing the file.

EPIPE   Indicates that an attempt is made to write to a pipe or first-in-first-out (FIFO) process that is not open for reading by any process. A SIGPIPE signal is sent to the process.

The fwrite subroutine is also unsuccessful due to the following error conditions:

ENOMEM  Indicates that insufficient storage space is available.

ENXIO   Indicates that a request was made of a nonexistent device, or the request was outside the capabilities of the device.

Related Information
The abort (“abort Subroutine” on page 2) subroutine, exit (“exit, atexit, unatexit, exit, or _Exit Subroutine” on page 242) subroutine, fclose (”fclose or fflush Subroutine” on page 252) subroutine, fopen, freopen, or fdopen subroutine, fgetwc, or getwchar (“getwc, fgetwc, or getwchar Subroutine” on page 472) subroutine, gets or fgets (“gets or fgets Subroutine” on page 429) subroutine, gets or fgets (“gets or fgets Subroutine” on page 429) subroutine, getwc (“getwc, fgetwc, or getwchar Subroutine” on page 472) subroutine, printf, fprintf, or sprintf (“printf, fprintf, sprintf, vsprintf, vsprintf, or vfprintf Subroutine” on page 1148) subroutine, ungetwc or ungetchar subroutine.

The Input and Output Handling in AIX 5L Version 5.3 General Programming Concepts: Writing and Debugging Programs.

freehostent Subroutine

Purpose
To free memory allocated by getipnodebyname and getipnodebyaddr.

Library
Standard C Library (libc.a)

Syntax
#include <netdb.h>
void freehostent (ptr)
struct hostent * ptr;

Description
The freehostent subroutine frees any dynamic storage pointed to by elements of ptr. This includes the hostent structure and the data areas pointed to by the h_name, h_addr_list, and h_aliases members of the hostent structure.

Related Information
The getipnodebyaddr subroutine and getipnodebyname subroutine.
freelmb Subroutine

Purpose
Returns a block of memory allocated by allocmb() to the system.

Syntax
#include <sys/dr.h>
int freelmb(long long laddr)

Description
The freelmb() subroutine returns a block of memory, allocated by allocmb(), for general system use.

Parameters
laddr A previously allocated LMB address.

Execution Environment
This freelmb() interface should only be called from the process environment.

Return Values
0 The LMB is successfully freed.

Error Codes
ENOTSUP LMB allocation not supported on this system.
EINVAL laddr does not describe a previously allocated LMB.
EINVAL Not in the process environment.

Related Information
allocmb Subroutine on page 68

frevoke Subroutine

Purpose
Revoke access to a file by other processes.

Library
Standard C Library (libc.a)

Syntax
int frevoke (FileDescriptor)
int FileDescriptor;

Description
The frevoke subroutine revokes access to a file by other processes.
All accesses to the file are revoked, except through the file descriptor specified by the FileDescriptor parameter to the frevoke subroutine. Subsequent attempts to access the file, using another file descriptor established before the frevoke subroutine was called, fail and cause the process to receive a return value of -1, and the errno global variable is set to EBADF.

A process can revoke access to a file only if its effective user ID is the same as the file owner ID or if the invoker has root user authority.

Note: The frevoke subroutine has no affect on subsequent attempts to open the file. To ensure exclusive access to the file, the caller should change the mode of the file before issuing the frevoke subroutine. Currently the frevoke subroutine works only on terminal devices.

Parameters

FileDescriptor A file descriptor returned by a successful open subroutine.

Return Values

Upon successful completion, the frevoke subroutine returns a value of 0.

If the frevoke subroutine fails, it returns a value of -1 and the errno global variable is set to indicate the error.

Error Codes

The frevoke subroutine fails if the following is true:

<table>
<thead>
<tr>
<th>Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EBADF</td>
<td>The FileDescriptor value is not the valid file descriptor of a terminal.</td>
</tr>
<tr>
<td>EPERM</td>
<td>The effective user ID of the calling process is not the same as the file owner ID.</td>
</tr>
<tr>
<td>EINVAL</td>
<td>Revocation of access rights is not implemented for this file.</td>
</tr>
</tbody>
</table>

frexp, frexpl, or frexp Subroutine

Purpose

Extracts the mantissa and exponent from a double precision number.

Syntax

#include <math.h>

float frexp (num, exp)
float num;
int *exp;

long double frexpl (num, exp)
long double num;
int exp;

double frexp (num, exp)
double num;
int *exp;

Description

The frexp, frexpl, and frexp subroutines break a floating-point number num into a normalized fraction and an integral power of 2. The integer exponent is stored in the int object pointed to by exp.
Parameters

*num* Specifies the floating-point number to be broken into a normalized fraction and an integral power of 2.

*exp* Points to where the integer exponent is stored.

Return Values

For finite arguments, the `frexf`, `frexpl`, and `frexp` subroutines return the value *x*, such that *x* has a magnitude in the interval \([\frac{1}{2}, 1)\) or 0, and *num* equals *x* times 2 raised to the power *exp*.

If *num* is NaN, a NaN is returned, and the value of "exp" is unspecified.

If *num* is ±0, ±0 is returned, and the value of "exp" is 0.

If *num* is ±Inf, *num* is returned, and the value of "exp" is unspecified.

Related Information

"class, _class, finite, isnan, or unordered Subroutines" on page 167 and "modf, modff, or modfl Subroutine" on page 839

In AIX 5L Version 5.3 Files Reference.

fscntl Subroutine

Purpose

Controls file system control operations.

Library

Standard C Library (libc.a)

Syntax

```c
#include <sys/types.h>
#include <j2/j2_cnt1.h>

int fscntl (vfs_id, Command, Argument, ArgumentSize);
```

Description

The `fscntl` subroutine performs a variety of file system-specific functions. These functions typically require root user authority.

The Enhanced Journaled File System (JFS2) supports several `Command` values that can be used by applications. Each of these `Command` values requires root authority.

FSCNTL_FREEZE

The file system specified by `vfs_id` is "frozen" for a specified amount of time. The act of freezing a file system produces a nearly consistent on-disk image of the file system, and writes all dirty file system metadata and user data to the disk. In its frozen state, the file system is read-only, and anything that attempts to modify the file system or its contents must wait for the freeze to end. The `Argument` is treated as an integral timeout value in seconds (instead of a pointer). The file system
is thawed by **FSCNTL_THAW** or when the timeout expires. The timeout, which must be a positive value, can be renewed using **FSCNTL_REFREEZE**. The **ArgumentSize** must be 0.

**Note:** For all applications using this interface, use **FSCNTL_THAW** to thaw the file system rather than waiting for the timeout to expire. If the timeout expires, an error log entry is generated as an advisory.

**FSCNTL_REFREEZE**

The file system specified by **vfs_id**, which must be already frozen, has its timeout value reset. If the command is used on a file system that is not frozen, an error is returned. The **Argument** is treated as an integral timeout value in seconds (instead of a pointer). The file system is thawed by **FSCNTL_THAW** or when the new timeout expires. The timeout must be a positive value. The **ArgumentSize** must be 0.

**FSCNTL_THAW**

The file system specified by **vfs_id** is thawed. Modifications to the file system are still allowed after it is thawed, and the file system image might no longer be consistent after the thaw occurs. If the file system is not frozen at the time of the call, an error is returned. The **Argument** and **ArgumentSize** must both be 0.

The Journaled File System (JFS) supports only internal **fscntl** interfaces. Application programs should not call this function on a JFS file system, because **fscntl** is reserved for system management commands, such as the **chfs** command.

**Parameters**

- **vfs_id**: Identifies the file system to be acted upon. This information is returned by the **stat** subroutine in the **st_vfs** field of the **stat.h** file.
- **Command**: Identifies the operation to be performed.
- **Argument**: Specifies a pointer to a block of file system specific information that defines how the operation is to be performed.
- **ArgumentSize**: Defines the size of the buffer pointed to by the **Argument** parameter.

**Return Values**

Upon successful completion, the **fscntl** subroutine returns a value of 0. Otherwise, a value of -1 is returned and the **errno** global variable is set to indicate the error.

**Error Codes**

The **fscntl** subroutine fails if any of the following errors are true:

- **EINVAL**: The **vfs_id** parameter does not identify a valid file system.
- **EINVAL**: The **Command** parameter is not recognized by the file system.
- **EINVAL**: The timeout specified to **FSCNTL_FREEZE** or **FSCNTL_REFREEZE** is invalid.
- **EALREADY**: The **Command** parameter was **FSCNTL_FREEZE** and the file system specified was already frozen.
- **EALREADY**: The **Command** parameter was **FSCNTL_REFREEZE** or **FSCNTL_THAW** and the file system specified was not frozen.

**Related Information**

The **chfs** command.

The **stat.h** file.
fseek, fseeko, fseeko64, rewind, ftell, ftello, ftello64, fgetpos, fgetpos64, fsetpos, or fsetpos64 Subroutine

Purpose
Repositions the file pointer of a stream.

Library
Standard C Library (libc.a)

Syntax
#include <stdio.h>

int fseek (Stream, Offset, Whence)
FILE *Stream;
long int Offset;
int Whence;
void rewind (Stream)
FILE *Stream;
long int ftell (Stream)
FILE *Stream;
int fgetpos (Stream, Position)
FILE *Stream;
  fpos_t *Position;

int fsetpos (Stream, Position)
FILE *Stream;
  const fpos_t *Position;

int fseeko (Stream, Offset, Whence)
FILE *Stream;
off_t Offset;
int Whence;

int fseeko64 (Stream, Offset, Whence)
FILE *Stream;
off64_t Offset;
int Whence;
off_t int ftello (Stream)
FILE *Stream;
off64_t int ftello64 (Stream)
FILE *Stream;
int fgetpos64 (Stream, Position)
FILE *Stream;
  fpos64_t *Position;

int fsetpos64 (Stream, Position)
FILE *Stream;
  const fpos64_t *Position;
Description

The `fseek`, `fseeko` and `fseeko64` subroutines set the position of the next input or output operation on the I/O stream specified by the `Stream` parameter. The position if the next operation is determined by the `Offset` parameter, which can be either positive or negative.

The `fseek`, `fseeko` and `fseeko64` subroutines set the file pointer associated with the specified `Stream` as follows:

- If the `Whence` parameter is set to the `SEEK_SET` value, the pointer is set to the value of the `Offset` parameter.
- If the `Whence` parameter is set to the `SEEK_CUR` value, the pointer is set to its current location plus the value of the `Offset` parameter.
- If the `Whence` parameter is set to the `SEEK_END` value, the pointer is set to the size of the file plus the value of the `Offset` parameter.

The `fseek`, `fseeko` and `fseeko64` subroutine are unsuccessful if attempted on a file that has not been opened using the `fopen` subroutine. In particular, the `fseek` subroutine cannot be used on a terminal or on a file opened with the `popen` subroutine. The `fseek` and `fseeko` subroutines will also fail when the resulting offset is larger than can be properly returned.

The `rewind` subroutine is equivalent to calling the `fseek` subroutine using parameter values of `(Stream, SEEK_SET, SEEK_SET)`, except that the `rewind` subroutine does not return a value.

The `fseek`, `fseeko`, `fseeko64` and `rewind` subroutines undo any effects of the `ungetc` and `ungetwc` subroutines and clear the end-of-file (EOF) indicator on the same stream.

The `fseek`, `fseeko`, and `fseeko64` function allows the file-position indicator to be set beyond the end of existing data in the file. If data is written later at this point, subsequent reads of data in the gap will return bytes of the value 0 until data is actually written into the gap.

A successful calls to the `fsetpos` or `fsetpos64` subroutines clear the EOF indicator and undoes any effects of the `ungetc` and `ungetwc` subroutines.

After an `fseek`, `fseeko`, `fseeko64` or a `rewind` subroutine, the next operation on a file opened for update can be either input or output.

`ftell`, `ftello` and `ftello64` subroutines return the position current value of the file-position indicator for the stream pointed to by the `Stream` parameter. `ftell` and `ftello` will fail if the resulting offset is larger than can be properly returned.

The `fgetpos` and `fgetpos64` subroutines store the current value of the file-position indicator for the stream pointed to by the `Stream` parameter in the object pointed to by the `Position` parameter. The `fsetpos` and `fsetpos64` set the file-position indicator for `Stream` according to the value of the `Position` parameter, which must be the result of a prior call to `fgetpos` or `fgetpos64` subroutine. `fgetpos` and `fsetpos` will fail if the resulting offset is larger than can be properly returned.

Parameters

- `Stream` Specifies the input/output (I/O) stream.
- `Offset` Determines the position of the next operation.
- `Whence` Determines the value for the file pointer associated with the `Stream` parameter.
- `Position` Specifies the value of the file-position indicator.
Return Values
Upon successful completion, the **fseek**, **fseeko** and **fseeko64** subroutine return a value of 0. Otherwise, it returns a value of -1.

Upon successful completion, the **ftell**, **ftello** and **ftello64** subroutine return the offset of the current byte relative to the beginning of the file associated with the named stream. Otherwise, a **long int** value of -1 is returned and the **errno** global variable is set.

Upon successful completion, the **fgetpos**, **fgetpos64**, **fsetpos** and **fsetpos64** subroutines return a value of 0. Otherwise, a nonzero value is returned and the **errno** global variable is set to the specific error.

The **errno** global variable is used to determine if an error occurred during a **rewind** subroutine call.

Error Codes
If the **fseek**, **fseeko**, **fseeko64**, **ftell**, **ftello**, **ftello64** or **rewind** subroutine are unsuccessful because the stream is unbuffered or the stream buffer needs to be flushed and the call to the subroutine causes an underlying **lseek** or **write** subroutine to be invoked, it returns one or more of the following error codes:

- **EAGAIN** Indicates that the **O_NONBLOCK** flag is set for the file descriptor, delaying the process in the write operation.
- **EBADF** Indicates that the file descriptor underlying the **Stream** parameter is not open for writing.
- **EFBIG** Indicates that an attempt has been made to write to a file that exceeds the file-size limit of the process or the maximum file size.
- **EIO** Indicates that the process is a member of a background process group attempting to perform a **write** subroutine to its controlling terminal, the **TOSTOP** flag is set, the process is not ignoring or blocking the **SIGTTOU** signal, and the process group of the process is orphaned. This error may also be returned under implementation-dependent conditions.
- **ENOSPC** Indicates that no remaining free space exists on the device containing the file.
- **EPipe** Indicates that an attempt has been made to write to a pipe or FIFO that is not open for reading by any process. A **SIGPIPE** signal will also be sent to the process.
- **EINVAL** Indicates that the **Whence** parameter is not valid. The resulting file-position indicator will be set to a negative value. The **EINVAL** error code does not apply to the **ftell** and **rewind** subroutines.
- **ESPIPE** Indicates that the file descriptor underlying the **Stream** parameter is associated with a pipe, **FIFO**, or socket.
- **EOVERFLOW** Indicates that for **fseek**, the resulting file offset would be a value that cannot be represented correctly in an object of type **long**.
- **EOVERFLOW** Indicates that for **fseeko**, the resulting file offset would be a value that cannot be represented correctly in an object of type **off_t**.
- **ENXIO** Indicates that a request was made of a non-existent device, or the request was outside the capabilities of the device.

The **fgetpos** and **fsetpos** subroutines are unsuccessful due to the following conditions:

- **EINVAL** Indicates that either the **Stream** or the **Position** parameter is not valid. The **EINVAL** error code does not apply to the **fgetpos** subroutine.
- **EBADF** Indicates that the file descriptor underlying the **Stream** parameter is not open for writing.
- **ESPIPE** Indicates that the file descriptor underlying the **Stream** parameter is associated with a pipe, **FIFO**, or socket.
The `fseek`, `fseeko`, `ftell`, `fgetpos`, and `fsetpos` subroutines are unsuccessful under the following condition:

**EOVERFLOW** The resulting could not be returned properly.

**Related Information**

The `closedir` subroutine, `fopen`, `fopen64`, `freopen`, `freopen64` or `fdopen` subroutine, `lseek` or `lseek64` subroutine, `opendir`, `readdir`, `rewinddir`, `seekdir`, `telldir` or `telldir64` subroutine, `popen` subroutine, `ungetc` or `ungetwc` subroutine, `write`, `writex`, `writev`, or `writevx` subroutine.

Input and Output Handling in AIX 5L Version 5.3 General Programming Concepts: Writing and Debugging Programs.

---

### fsync or fsync_range Subroutine

**Purpose**

Writes changes in a file to permanent storage.

**Library**

Standard C Library (`libc.a`)

**Syntax**

```c
#include <unistd.h>

int fsync (FileDescriptor);
int FileDescriptor;

int fsync_range (FileDescriptor, how, start, length);
int FileDescriptor;
int how;
off_t start;
off_t length;
```

**Description**

The `fsync` subroutine causes all modified data in the open file specified by the `FileDescriptor` parameter to be saved to permanent storage. On return from the `fsync` subroutine, all updates have been saved on permanent storage.

The `fsync_range` subroutine causes all modified data in the specified range of the open file specified by the `FileDescriptor` parameter to be saved to permanent storage. On return from the `fsync_range` subroutine, all updates in the specified range have been saved on permanent storage.

Data written to a file that a process has opened for deferred update (with the `O_DEFER` flag) is not written to permanent storage until another process issues an `fsync_range` or `fsync` call against this file or runs a synchronous `write` subroutine (with the `O_SYNC` flag) on this file. See the `fcntl.h` file and the `open` subroutine for descriptions of the `O_DEFER` and `O_SYNC` flags respectively.
Note: The file identified by the `FileDescriptor` parameter must be open for writing when the `fsync_range` or `fsync` subroutine is issued or the call is unsuccessful. This restriction was not enforced in BSD systems.

Parameters

- **FileDescriptor**
  A valid, open file descriptor.

- **how**
  How to flush, `O_DSYNC`, `O_NOCACHE`, or `O_SYNC`.

  - `O_DSYNC`
    Write file data and metadata to retrieve the data for the specified range.

  - `O_NOCACHE`
    Write the data in the range and release full memory pages in the byte range. The data will no longer be cached.

  - `O_SYNC`
    Write all modified file data and metadata for the specified range.

- **start**
  Starting file offset.

- **length**
  Length, or zero for everything.

Return Values

Upon successful completion, the `fsync` subroutine returns a value of 0. Otherwise, a value of -1 is returned and the `errno` global variable is set to indicate the error.

Upon successful completion, the `fsync_range` subroutine returns a value of 0. Otherwise, a value of -1 is returned and the `errno` global variable is set to indicate the error.

Error Codes

The `fsync` subroutine is unsuccessful if one or more of the following are true:

- **EIO**
  An I/O error occurred while reading from or writing to the file system.

- **EBADF**
  The `FileDescriptor` parameter is not a valid file descriptor open for writing.

- **EINVAL**
  The file is not a regular file.

- **EINTR**
  The `fsync` subroutine was interrupted by a signal.

Related Information

The `open`, `openx`, or `creat` subroutine, `sync` subroutine, `write`, `writex`, `writev`, or `writevx` subroutine.

The `fcntl.h` file.

Files, Directories, and File Systems Overview for Programmers in AIX 5L Version 5.3 General Programming Concepts: Writing and Debugging Programs contains information about i-nodes, file descriptors, file-space allocation, and more.

ftok Subroutine

Purpose

Generates a standard interprocess communication key.

Library

Standard C Library (`libc.a`)
Syntax

```c
#include <sys/types.h>
#include <sys/ipc.h>

key_t ftok (Path, ID);
char *Path;
int ID;
```

Description

**Attention:** If the `Path` parameter of the `ftok` subroutine names a file that has been removed while keys still refer to it, the `ftok` subroutine returns an error. If that file is then re-created, the `ftok` subroutine will probably return a key different from the original one.

**Attention:** Each installation should define standards for forming keys. If standards are not adhered to, unrelated processes may interfere with each other’s operation.

**Attention:** The `ftok` subroutine does not guarantee unique key generation. However, the occurrence of key duplication is very rare and mostly for across file systems.

The `ftok` subroutine returns a key, based on the `Path` and `ID` parameters, to be used to obtain interprocess communication identifiers. The `ftok` subroutine returns the same key for linked files if called with the same `ID` parameter. Different keys are returned for the same file if different `ID` parameters are used.

All interprocess communication facilities require you to supply a key to the `msgget`, `semget`, and `shmget` subroutines in order to obtain interprocess communication identifiers. The `ftok` subroutine provides one method for creating keys, but other methods are possible. For example, you can use the project ID as the most significant byte of the key, and use the remaining portion as a sequence number.

Parameters

- `Path`: Specifies the path name of an existing file that is accessible to the process.
- `ID`: Specifies a character that uniquely identifies a project.

Return Values

When successful, the `ftok` subroutine returns a key that can be passed to the `msgget`, `semget`, or `shmget` subroutine.

Error Codes

The `ftok` subroutine returns the value (key_t)-1 if one or more of the following are true:

- The file named by the `Path` parameter does not exist.
- The file named by the `Path` parameter is not accessible to the process.
- The `ID` parameter has a value of 0.

Related Information

- `msgget` subroutine
- `semget` subroutine
- `shmget` subroutine
- Subroutines Overview and Understanding Memory Mapping in AIX 5L Version 5.3 General Programming Concepts: Writing and Debugging Programs.
ftw or ftw64 Subroutine

Purpose
Walks a file tree.

Library
Standard C Library (libc.a)

Syntax
```c
#include <ftw.h>

int ftw (Path, Function, Depth);
char *Path;
int (*Function)(const char*, const struct stat*, int);
int Depth;

int ftw64 (Path, Function, Depth);
char *Path;
int (*Function)(const char*, const struct stat64*, int);
int Depth;
```

Description
The ftw and ftw64 subroutines recursively searches the directory hierarchy that descends from the directory specified by the Path parameter.

For each file in the hierarchy, the ftw and ftw64 subroutines call the function specified by the Function parameter. ftw passes it a pointer to a null-terminated character string containing the name of the file, a pointer to a stat structure containing information about the file, and an integer. ftw64 passes it a pointer to a null-terminated character string containing the name of the file, a pointer to a stat64 structure containing information about the file, and an integer.

The integer passed to the Function parameter identifies the file type with one of the following values:

<table>
<thead>
<tr>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>FTW_F</td>
<td>Regular file</td>
</tr>
<tr>
<td>FTW_D</td>
<td>Directory</td>
</tr>
<tr>
<td>FTW_DNR</td>
<td>Directory that cannot be read</td>
</tr>
<tr>
<td>FTW_SL</td>
<td>Symbolic Link</td>
</tr>
<tr>
<td>FTW_NS</td>
<td>File for which the stat structure could not be executed successfully</td>
</tr>
</tbody>
</table>

If the integer is FTW-DNR, the files and subdirectories contained in that directory are not processed.

If the integer is FTW-NS, the stat structure contents are meaningless. An example of a file that causes FTW-NS to be passed to the Function parameter is a file in a directory for which you have read permission but not execute (search) permission.

The ftw and ftw64 subroutines finish processing a directory before processing any of its files or subdirectories.

The ftw and ftw64 subroutines continue the search until the directory hierarchy specified by the Path parameter is completed, an invocation of the function specified by the Function parameter returns a nonzero value, or an error is detected within the ftw and ftw64 subroutines, such as an I/O error.
The `ftw` and `ftw64` subroutines traverse symbolic links encountered in the resolution of the `Path` parameter, including the final component. Symbolic links encountered while walking the directory tree rooted at the `Path` parameter are not traversed.

The `ftw` and `ftw64` subroutines use one file descriptor for each level in the tree. The `Depth` parameter specifies the maximum number of file descriptors to be used. In general, the `ftw` and `ftw64` subroutines runs faster if the value of the `Depth` parameter is at least as large as the number of levels in the tree. However, the value of the `Depth` parameter must not be greater than the number of file descriptors currently available for use. If the value of the `Depth` parameter is 0 or a negative number, the effect is the same as if it were 1.

Because the `ftw` and `ftw64` subroutines are recursive, it is possible for it to terminate with a memory fault due to stack overflow when applied to very deep file structures.

The `ftw` and `ftw64` subroutines use the `malloc` subroutine to allocate dynamic storage during its operation. If the `ftw` and `ftw64` subroutine is terminated prior to its completion, such as by the `longjmp` subroutine being executed by the function specified by the `Function` parameter or by an interrupt routine, the `ftw` and `ftw64` subroutines cannot free that storage. The storage remains allocated. A safe way to handle interrupts is to store the fact that an interrupt has occurred, and arrange to have the function specified by the `Function` parameter return a nonzero value the next time it is called.

**Parameters**

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>Path</code></td>
<td>Specifies the directory hierarchy to be searched.</td>
</tr>
<tr>
<td><code>Function</code></td>
<td>Specifies the file type.</td>
</tr>
<tr>
<td><code>Depth</code></td>
<td>Specifies the maximum number of file descriptors to be used. <code>Depth</code> cannot be greater than <code>OPEN_MAX</code> which is described in the <code>sys/limits.h</code> header file.</td>
</tr>
</tbody>
</table>

**Return Values**

If the tree is exhausted, the `ftw` and `ftw64` subroutines returns a value of 0. If the subroutine pointed to by `fn` returns a nonzero value, `ftw` and `ftw64` subroutines stops its tree traversal and returns whatever value was returned by the subroutine pointed to by `fn`. If the `ftw` and `ftw64` subroutines detects an error, it returns a -1 and sets the `errno` global variable to indicate the error.

**Error Codes**

If the `ftw` or `ftw64` subroutines detect an error, a value of -1 is returned and the `errno` global variable is set to indicate the error.

The `ftw` and `ftw64` subroutine are unsuccessful if:

- **EACCES**  
  Search permission is denied for any component of the `Path` parameter or read permission is denied for `Path`.

- **ENAMETOOLONG**  
  The length of the path exceeds `PATH_MAX` while `_POSIX_NO_TRUNC` is in effect.

- **ENOENT**  
  The `Path` parameter points to the name of a file that does not exist or points to an empty string.

- **ENOTDIR**  
  A component of the `Path` parameter is not a directory.

The `ftw` subroutine is unsuccessful if:

- **EOVERFLOW**  
  A file in `Path` is of a size larger than 2 Gigabytes.
fwide Subroutine

Purpose
Set stream orientation.

Library
Standard Library (libc.a)

Syntax
#include <stdio.h>
#include <wchar.h>
int fwide (FILE * stream, int mode),

Description
The fwide function determines the orientation of the stream pointed to by stream. If mode is greater than zero, the function first attempts to make the stream wide-oriented. If mode is less than zero, the function first attempts to make the stream byte-oriented. Otherwise, mode is zero and the function does not alter the orientation of the stream.

If the orientation of the stream has already been determined, fwide does not change it.

Because no return value is reserved to indicate an error, an application wishing to check for error situations should set errno to 0, then call fwide, then check errno and if it is non-zero, assume an error has occurred.

A call to fwide with mode set to zero can be used to determine the current orientation of a stream.

Return Values
The fwide function returns a value greater than zero if, after the call, the stream has wide-orientation, a value less than zero if the stream has byte-orientation, or zero if the stream has no orientation.

Errors
The fwide function may fail if:

EBADF The stream argument is not a valid stream.

Related Information
The wchar.h file
fwprintf, wprintf, swprintf Subroutines

Purpose
Print formatted wide-character output.

Library
Standard Library (libc.a)

Syntax
#include <stdio.h>
#include <wchar.h>

int fprintf (FILE * stream, const wchar_t * format, . . .)
int wprintf (const wchar_t * format, . . .)
int swprintf (wchar_t *s, size_t n, const wchar_t * format, . . .)

Description
The fprintf function places output on the named output stream. The wprintf function places output on
the standard output stream stdout. The swprintf function places output followed by the null
wide-character in consecutive wide-characters starting at *s; no more than n wide-characters are written,
including a terminating null wide-character, which is always added (unless n is zero).

Each of these functions converts, formats and prints its arguments under control of the format
wide-character string. The format is composed of zero or more directives: ordinary wide-characters,
which are simply copied to the output stream and conversion specifications, each of which results in
the fetching of zero or more arguments. The results are undefined if there are insufficient arguments for
the format. If the format is exhausted while arguments remain, the excess arguments are evaluated but
are otherwise ignored.

EX Conversions can be applied to the nth argument after the format in the argument list, rather than to
the next unused argument. In this case, the conversion wide-character % (see below) is replaced by the
sequence %n$, where n is a decimal integer in the range [1, {NL_ARGMAX}], giving the position of the
argument in the argument list. This feature provides for the definition of format wide-character strings that
select arguments in an order appropriate to specific languages (see the EXAMPLES section).

In format wide-character strings containing the %n$ form of conversion specifications, numbered
arguments in the argument list can be referenced from the format wide-character string as many times as
required.

In format wide-character strings containing the % form of conversion specifications, each argument in the
argument list is used exactly once.

All forms of the fprintf functions allow for the insertion of a language-dependent radix character in the
output string, output as a wide-character value. The radix character is defined in the program’s locale
(category LC_NUMERIC). In the POSIX locale, or in a locale where the radix character is not defined, the
radix character defaults to a period (.).

EX Each conversion specification is introduced by the % wide-character or by the wide-character
sequence %n$, after which the following appear in sequence:
• Zero or more flags (in any order), which modify the meaning of the conversion specification.
• An optional minimum field width. If the converted value has fewer wide-characters than the field width,
it will be padded with spaces by default on the left; it will be padded on the right, if the left-adjustment
flag (-), described below, is given to the field width. The field width takes the form of an asterisk (*),
described below, or a decimal integer.
• An optional **precision** that gives the minimum number of digits to appear for the `d`, `i`, `o`, `u`, `x` and `X` conversions; the number of digits to appear after the radix character for the `e`, `E` and `f` conversions; the maximum number of significant digits for the `g` and `G` conversions; or the maximum number of wide-characters to be printed from a string in `s` conversions. The precision takes the form of a period (.) followed either by an asterisk (*), described below, or an optional decimal digit string, where a null digit string is treated as 0. If a precision appears with any other conversion wide-character, the behaviour is undefined.

• An optional **l** (ell) specifying that a following `c` conversion wide-character applies to a `wint_t` argument; an optional **l** specifying that a following `s` conversion wide-character applies to a `wchar_t` argument; an optional **h** specifying that a following `d`, `i`, `o`, `u`, `x` or `X` conversion wide-character applies to a type `short` or type `unsigned short` argument (the argument will have been promoted according to the integral promotions, and its value will be converted to type `short` or `unsigned short` before printing); an optional **h** specifying that a following `n` conversion wide-character applies to a pointer to a type `short` or `unsigned short` argument; an optional **l** (ell) specifying that a following `d`, `i`, `o`, `u`, `x` or `X` conversion wide-character applies to a type `long` or type `unsigned long` argument; an optional **l** (ell) specifying that a following `n` conversion wide-character applies to a pointer to a type `long` or `unsigned long` argument; or an optional **L** specifying that a following `e`, `E`, `f`, `g` or `G` conversion wide-character applies to a type `long double` argument. If an **h**, **l** or **L** appears with any other conversion wide-character, the behavior is undefined.

• A **conversion wide-character** that indicates the type of conversion to be applied.

A field width, or precision, or both, may be indicated by an asterisk (*). In this case an argument of type `int` supplies the field width or precision. Arguments specifying field width, or precision, or both must appear in that order before the argument, if any, to be converted. A negative field width is taken as a - flag followed by a positive field width. A negative precision is taken as if EX the precision were omitted. In format wide-character strings containing the `%n$` form of a conversion specification, a field width or precision may be indicated by the sequence "m$, where m is a decimal integer in the range [1, {NL_ARGMAX}] giving the position in the argument list (after the format argument) of an integer argument containing the field width or precision, for example:

```c
wprintf(L"%1$d:%2$.*3$d:%4$.*3$d\n", hour, min, precision, sec);
```

The **format** can contain either numbered argument specifications (that is, `%n$` and "m$"), or unnumbered argument specifications (that is, % and "), but normally not both. The only exception to this is that %% can be mixed with the `%n$` form. The results of mixing numbered and unnumbered argument specifications in a **format** wide-character string are undefined. When numbered argument specifications are used, specifying the Nth argument requires that all the leading arguments, from the first to the (N-1)th, are specified in the format wide-character string.

The flag wide-characters and their meanings are:

- The integer portion of the result of a decimal conversion (`%i`, `%d`, `%u`, `%f`, `%g` or `%G`) will be formatted with thousands’ grouping wide-characters. For other conversions the behaviour is undefined. The non-monetary grouping wide-character is used.

- The result of the conversion will be left-justified within the field. The conversion will be right-justified if this flag is not specified.

+ The result of a signed conversion will always begin with a sign (+ or -). The conversion will begin with a sign only when a negative value is converted if this flag is not specified.

space If the first wide-character of a signed conversion is not a sign or if a signed conversion results in no wide-characters, a space will be prefixed to the result. This means that if the space and + flags both appear, the space flag will be ignored.
This flag specifies that the value is to be converted to an alternative form. For o conversion, it increases the precision (if necessary) to force the first digit of the result to be 0. For x or X conversions, a non-zero result will have 0x (or 0X) prefixed to it. For e, E, f, g or G conversions, the result will always contain a radix character, even if no digits follow it. Without this flag, a radix character appears in the result of these conversions only if a digit follows it. For g and G conversions, trailing zeros will not be removed from the result as they normally are. For other conversions, the behavior is undefined.

For d, i, u, o, x, e, E, f, g and G conversions, leading zeros (following any indication of sign or base) are used to pad to the field width; no space padding is performed. If the 0 and - flags both appear, the 0 flag will be ignored. For d, i, o, u, x and X conversions, if a precision is specified, the 0 flag will be ignored. If the 0 and ' flags both appear, the grouping wide-characters are inserted before zero padding. For other conversions, the behavior is undefined.

The conversion wide-characters and their meanings are:

- **d, i**: The int argument is converted to a signed decimal in the style [-] dddd. The precision specifies the minimum number of digits to appear; if the value being converted can be represented in fewer digits, it will be expanded with leading zeros. The default precision is 1. The result of converting 0 with an explicit precision of 0 is no wide-characters.

- **o**: The unsigned int argument is converted to unsigned octal format in the style dddd. The precision specifies the minimum number of digits to appear; if the value being converted can be represented in fewer digits, it will be expanded with leading zeros. The default precision is 1. The result of converting 0 with an explicit precision of 0 is no wide-characters.

- **u**: The unsigned int argument is converted to unsigned decimal format in the style dddd. The precision specifies the minimum number of digits to appear; if the value being converted can be represented in fewer digits, it will be expanded with leading zeros. The default precision is 1. The result of converting 0 with an explicit precision of 0 is no wide-characters.

- **x**: The unsigned int argument is converted to unsigned hexadecimal format in the style dddd; the letters abcdedef are used. The precision specifies the minimum number of digits to appear; if the value being converted can be represented in fewer digits, it will be expanded with leading zeros. The default precision is 1. The result of converting 0 with an explicit precision of 0 is no wide-characters.

- **X**: Behaves the same as the x conversion wide-character except that letters ABCDEF are used instead of abcdedef.

- **f**: The double argument is converted to decimal notation in the style [-] dddd.dd, where the number of digits after the radix character is equal to the precision specification. If the precision is missing, it is taken as 6; if the precision is explicitly 0 and no # flag is present, no radix character appears. If a radix character appears, at least one digit appears before it. The value is rounded to the appropriate number of digits.

- **e, E**: The double argument is converted in the style [-] d.ddde +/- dd, where there is one digit before the radix character (which is non-zero if the argument is non-zero) and the number of digits after it is equal to the precision; if the precision is missing, it is taken as 6; if the precision is 0 and no # flag is present, no radix character appears. The value is rounded to the appropriate number of digits. The E conversion wide-character will produce a number with E instead of e introducing the exponent. The exponent always contains at least two digits. If the value is 0, the exponent is 0.

- **g, G**: The double argument is converted in the style f or e (or in the style E in the case of a G conversion wide-character), with the precision specifying the number of significant digits. If an explicit precision is 0, it is taken as 1. The style used depends on the value converted; style e (or E) will be used only if the exponent resulting from such a conversion is less than -4 or greater than or equal to the precision. Trailing zeros are removed from the fractional portion of the result; a radix character appears only if it is followed by a digit.

The fwprintf family of functions may make available wide-character string representations for infinity and NaN.
If no l (ell) qualifier is present, the int argument is converted to a wide-character as if by calling the \texttt{btowc} function and the resulting wide-character is written. Otherwise the \texttt{wint_t} argument is converted to \texttt{wchar_t}, and written.

If no l (ell) qualifier is present, the argument must be a pointer to a character array containing a character sequence beginning in the initial shift state. Characters from the array are converted as if by repeated calls to the \texttt{mbrtowc} function, with the conversion state described by an \texttt{mbstate_t} object initialised to zero before the first character is converted, and written up to (but not including) the terminating null wide-character. If the precision is specified, no more than that many wide-characters are written. If the precision is not specified or is greater than the size of the array, the array must contain a null wide-character.

If an l (ell) qualifier is present, the argument must be a pointer to an array of type \texttt{wchar_t}. Wide characters from the array are written up to (but not including) a terminating null wide-character. If no precision is specified or is greater than the size of the array, the array must contain a null wide-character. If a precision is specified, no more than that many wide-characters are written.

The argument must be a pointer to void. The value of the pointer is converted to a sequence of printable wide-characters, in an implementation-dependent manner. The argument must be a pointer to an integer into which is written the number of wide-characters written to the output so far by this call to one of the \texttt{fwprintf} functions. No argument is converted.

C
Same as lc.
S
Same as ls.
%
Output a % wide-character; no argument is converted. The entire conversion specification must be \texttt{%%}.

If a conversion specification does not match one of the above forms, the behavior is undefined.

In no case does a non-existent or small field width cause truncation of a field; if the result of a conversion is wider than the field width, the field is simply expanded to contain the conversion result. Characters generated by \texttt{fwprintf} and \texttt{wprintf} are printed as if \texttt{fputwc} had been called.

The \texttt{st_ctime} and \texttt{st_mtime} fields of the file will be marked for update between the call to a successful execution of \texttt{fwprintf} or \texttt{wprintf} and the next successful completion of a call to \texttt{fflush} or \texttt{fclose} on the same stream or a call to exit or abort.

Return Values
Upon successful completion, these functions return the number of wide-characters transmitted excluding the terminating null wide-character in the case of \texttt{swprintf} or a negative value if an output error was encountered.

Error Codes
For the conditions under which \texttt{fwprintf} and \texttt{wprintf} will fail and may fail, refer to \texttt{fputwc} . In addition, all forms of \texttt{fwprintf} may fail if:

\texttt{EILSEQ} A wide-character code that does not correspond to a valid character has been detected.
\texttt{EINVAL} There are insufficient arguments.

In addition, \texttt{wprintf} and \texttt{fwprintf} may fail if:

\texttt{ENOMEM} Insufficient storage space is available.

Examples
To print the language-independent date and time format, the following statement could be used:

\texttt{wprintf (format, weekday, month, day, hour, min);}

For American usage, format could be a pointer to the wide-character string:

\texttt{L"%s, %s %d, %d:%.2d\n"}
producing the message:
Sunday, July 3, 10:02

whereas for German usage, format could be a pointer to the wide-character string:
L"%1$s, %3$d, %2$s, %4$d:%5$.2d\n"

producing the message:
Sonntag, 3. July, 10:02

**Related Information**
The **btowc** subroutine, **fputwc** subroutine, **putwc, putwchar, or fputwc Subroutine** subroutine, **fwscanf** subroutine, **wscanf, swscanf Subroutines** subroutine, **setlocale** subroutine, **mbrtowc** subroutine, **mbtowc Subroutine on page 784** subroutine.

The **wchar.h** file.

---

**fwscanf, wscanf, swscanf Subroutines**

**Purpose**
Convert formatted wide-character input.

**Library**
Standard Library (**libc.a**)

**Syntax**

```c
#include <stdio.h>
#include <wchar.h>

int fwscanf (FILE * stream, const wchar_t * format, ...);
int wscanf (const wchar_t * format, ...);
int swscanf (const wchar_t * s, const wchar_t * format, ...);
```

**Description**

The **fwscanf** function reads from the named input stream. The **wscanf** function reads from the standard input stream stdin. The **swscanf** function reads from the wide-character string **s**. Each function reads wide-characters, interprets them according to a format, and stores the results in its arguments. Each expects, as arguments, a control wide-character string format described below, and a set of pointer arguments indicating where the converted input should be stored. The result is undefined if there are insufficient arguments for the format. If the format is exhausted while arguments remain, the excess arguments are evaluated but are otherwise ignored.

Conversions can be applied to the **nth** argument after the **format** in the argument list, rather than to the next unused argument. In this case, the conversion wide-character % (see below) is replaced by the sequence **%n$**, where n is a decimal integer in the range [1, **NL_ARGMAX**]). This feature provides for the definition of format wide-character strings that select arguments in an order appropriate to specific languages. In format wide-character strings containing the **%n$** form of conversion specifications, it is unspecified whether numbered arguments in the argument list can be referenced from the format wide-character string more than once.

The format can contain either form of a conversion specification, that is, % or %n$, but the two forms cannot normally be mixed within a single format wide-character string. The only exception to this is that %% or %* can be mixed with the %n$ form.
The `fwscanf` function in all its forms allows for detection of a language-dependent radix character in the input string, encoded as a wide-character value. The radix character is defined in the program's locale (category LC_NUMERIC). In the POSIX locale, or in a locale where the radix character is not defined, the radix character defaults to a period (.)

The format is a wide-character string composed of zero or more directives. Each directive is composed of one of the following: one or more white-space wide-characters (space, tab, newline, vertical-tab or form-feed characters); an ordinary wide-character (neither % nor a white-space character); or a conversion specification. Each conversion specification is introduced by a % or the sequence `%n\$` after which the following appear in sequence:

- An optional assignment-suppressing character *.
- An optional non-zero decimal integer that specifies the maximum field width.
- An optional size modifier h, l (ell) or L indicating the size of the receiving object. The conversion wide-characters c, s and [ must be preceded by l (ell) if the corresponding argument is a pointer to `wchar_t` rather than a pointer to a character type. The conversion wide-characters d, i and n must be preceded by h if the corresponding argument is a pointer to `short int` rather than a pointer to `int`, or by l (ell) if it is a pointer to `long int`. Similarly, the conversion wide-characters o, u and x must be preceded by h if the corresponding argument is a pointer to `unsigned short int` rather than a pointer to `unsigned int`, or by l (ell) if it is a pointer to `unsigned long int`. The conversion wide-characters e, f and g must be preceded by l (ell) if the corresponding argument is a pointer to `double` rather than a pointer to `float`, or by L if it is a pointer to `long double`. If an h, l (ell) or L appears with any other conversion wide-character, the behavior is undefined.
- A conversion wide-character that specifies the type of conversion to be applied. The valid conversion wide-characters are described below.

The `fwscanf` functions execute each directive of the format in turn. If a directive fails, as detailed below, the function returns. Failures are described as input failures (due to the unavailability of input bytes) or matching failures (due to inappropriate input).

A directive composed of one or more white-space wide-characters is executed by reading input until no more valid input can be read, or up to the first wide-character which is not a white-space wide-character, which remains unread.

A directive that is an ordinary wide-character is executed as follows. The next wide-character is read from the input and compared with the wide-character that comprises the directive; if the comparison shows that they are not equivalent, the directive fails, and the differing and subsequent wide-characters remain unread.

A directive that is a conversion specification defines a set of matching input sequences, as described below for each conversion wide-character. A conversion specification is executed in the following steps:

Input white-space wide-characters (as specified by `iswspace`) are skipped, unless the conversion specification includes a [, c or n conversion character.

An item is read from the input, unless the conversion specification includes an n conversion wide-character. An input item is defined as the longest sequence of input wide-characters, not exceeding any specified field width, which is an initial subsequence of a matching sequence. The first wide-character, if any, after the input item remains unread. If the length of the input item is 0, the execution of the conversion specification fails; this condition is a matching failure, unless end-of-file, an encoding error, or a read error prevented input from the stream, in which case it is an input failure.

Except in the case of a % conversion wide-character, the input item (or, in the case of a %n conversion specification, the count of input wide-characters) is converted to a type appropriate to the conversion wide-character. If the input item is not a matching sequence, the execution of the conversion specification fails; this condition is a matching failure. Unless assignment suppression was indicated by a *, the result of
the conversion is placed in the object pointed to by the first argument following the format argument that has not already received a conversion result if the conversion specification is introduced by %, or in the nth argument if introduced by the wide-character sequence %n$. If this object does not have an appropriate type, or if the result of the conversion cannot be represented in the space provided, the behavior is undefined.

The following conversion wide-characters are valid:

- **d** Matches an optionally signed decimal integer, whose format is the same as expected for the subject sequence of wcstol with the value 10 for the base argument. In the absence of a size modifier, the corresponding argument must be a pointer to int.
- **i** Matches an optionally signed integer, whose format is the same as expected for the subject sequence of wcstol with 0 for the base argument. In the absence of a size modifier, the corresponding argument must be a pointer to int.
- **o** Matches an optionally signed octal integer, whose format is the same as expected for the subject sequence of wcstoul with the value 8 for the base argument. In the absence of a size modifier, the corresponding argument must be a pointer to unsigned int.
- **u** Matches an optionally signed decimal integer, whose format is the same as expected for the subject sequence of wcstoul with the value 10 for the base argument. In the absence of a size modifier, the corresponding argument must be a pointer to unsigned int.
- **x** Matches an optionally signed hexadecimal integer, whose format is the same as expected for the subject sequence of wcstoul with the value 16 for the base argument. In the absence of a size modifier, the corresponding argument must be a pointer to unsigned int.
- **e**, **f**, **g** Matches an optionally signed floating-point number, whose format is the same as expected for the subject sequence of wcstod. In the absence of a size modifier, the corresponding argument must be a pointer to float.

If the fwprintf family of functions generates character string representations for infinity and NaN (a 7858 symbolic entity encoded in floating-point format) to support the ANSI/IEEE Std 754:1985 standard, the fscanf family of functions will recognise them as input.

- **s** Matches a sequence of non white-space wide-characters. If no l (ell) qualifier is present, characters from the input field are converted as if by repeated calls to the wcrtomb function, with the conversion state described by an mbstate_t object initialised to zero before the first wide-character is converted. The corresponding argument must be a pointer to a character array large enough to accept the sequence and the terminating null character, which will be added automatically. Otherwise, the corresponding argument must be a pointer to an array of wchar_t large enough to accept the sequence and the terminating null wide-character, which will be added automatically.
- **[**
  Matches a non-empty sequence of wide-characters from a set of expected wide-characters (the scanset). If no l (ell) qualifier is present, wide-characters from the input field are converted as if by repeated calls to the wcrtomb function, with the conversion state described by an mbstate_t object initialised to zero before the first wide-character is converted. The corresponding argument must be a pointer to a character array large enough to accept the sequence and the terminating null character, which will be added automatically.

If an l (ell) qualifier is present, the corresponding argument must be a pointer to an array of wchar_t large enough to accept the sequence and the terminating null wide-character, which will be added automatically.

The conversion specification includes all subsequent wide characters in the format string up to and including the matching right square bracket (]). The wide-characters between the square brackets (the scanlist) comprise the scanset, unless the wide-character after the left square bracket is a circumflex (^), in which case the scanset contains all wide-characters that do not appear in the scanlist between the circumflex and the right square bracket. If the conversion specification begins with [ ] or [^], the right square bracket is included in the scanlist and the next right square bracket is the matching right square bracket that ends the conversion specification; otherwise the first right square bracket is the one that ends the conversion specification. If a - is in the scanlist and is not the first wide-character, nor the second where the first wide-character is a ^, nor the last wide-character, the behavior is implementation-dependent.
Matches a sequence of wide-characters of the number specified by the field width (1 if no field width is
present in the conversion specification). If no l (ell) qualifier is present, wide-characters from the input
field are converted as if by repeated calls to the \texttt{wcrtomb} function, with the conversion state described
by an \texttt{mbstate_t} object initialised to zero before the first wide-character is converted. The
corresponding argument must be a pointer to a character array large enough to accept the sequence.
No null character is added.
Otherwise, the corresponding argument must be a pointer to an array of \texttt{wchar_t} large enough to
accept the sequence. No null wide-character is added.

Matches an implementation-dependent set of sequences, which must be the same as the set of
sequences that is produced by the \texttt{%p} conversion of the corresponding \texttt{fwprintf} functions. The
corresponding argument must be a pointer to a pointer to void. The interpretation of the input item is
implementation-dependent. If the input item is a value converted earlier during the same program
execution, the pointer that results will compare equal to that value; otherwise the behavior of the \texttt{%p}
conversion is undefined.

No input is consumed. The corresponding argument must be a pointer to the integer into which is to be
written the number of wide-characters read from the input so far by this call to the \texttt{fwscanf} functions.
Execution of a \texttt{%n} conversion specification does not increment the assignment count returned at the
completion of execution of the function.

Same as lc.

Same as ls.

Matches a single \texttt{%;} no conversion or assignment occurs. The complete conversion specification must
be \texttt{%%}.

If a conversion specification is invalid, the behavior is undefined.

The conversion characters E, G and X are also valid and behave the same as, respectively, e, g and x.

If end-of-file is encountered during input, conversion is terminated. If end-of-file occurs before any
wide-characters matching the current conversion specification (except for \texttt{%n}) have been read (other than
leading white-space, where permitted), execution of the current conversion specification terminates with an
input failure. Otherwise, unless execution of the current conversion specification is terminated with a
matching failure, execution of the following conversion specification (if any) is terminated with an input
failure.

Reaching the end of the string in \texttt{swscanf} is equivalent to encountering end-of-file for \texttt{fwscanf}.

If conversion terminates on a conflicting input, the offending input is left unread in the input. Any trailing
white space (including newline) is left unread unless matched by a conversion specification. The success
of literal matches and suppressed assignments is only directly determinable via the \texttt{%n} conversion
specification.

The \texttt{fwscanf} and \texttt{wscanf} functions may mark the \texttt{st_atime} field of the file associated with stream for
update. The \texttt{st_atime} field will be marked for update by the first successful execution of \texttt{fgetc}, \texttt{fgetwc},
\texttt{fgets}, \texttt{fgetws}, \texttt{fread}, \texttt{getc}, \texttt{getwc}, \texttt{getwchar}, \texttt{getchar}, \texttt{getchar}, \texttt{gets}, \texttt{fscanf} or \texttt{fwscanf} using stream that returns
data not supplied by a prior call to \texttt{ungetc}.

In format strings containing the \texttt{%} form of conversion specifications, each argument in the argument list is
used exactly once.

**Return Values**

Upon successful completion, these functions return the number of successfully matched and assigned
input items; this number can be 0 in the event of an early matching failure. If the input ends before the first
matching failure or conversion, EOF is returned. If a read error occurs the error indicator for the stream is
set, EOF is returned, and errno is set to indicate the error.
Error Codes
For the conditions under which the `fwscanf` functions will fail and may fail, refer to `fgetwc`. In addition, `fwscanf` may fail if:

- **EILSEQ** Input byte sequence does not form a valid character.
- **EINVAL** There are insufficient arguments.

Examples
The call:
```
int i, n; float x; char name[50];
```
```
n = wscanf(L"%d%f%s", &i, &x, name);
```

with the input line:
```
25 54.32E-1 Hamster
```

will assign to `n` the value 3, to `i` the value 25, to `x` the value 5.432, and `name` will contain the string Hamster.

The call:
```
int i; float x; char name[50];
```
```
(void) wscanf(L"%2d%f%*d %[0123456789]", &i, &x, name);
```

with input:
```
56789 0123 56a72
```

will assign 56 to `i`, 789.0 to `x`, skip 0123, and place the string 56\0 in `name`. The next call to `getchar` will return the character `a`.

Related Information
The `getwc` subroutine, `fwprintf` subroutine, `swprintf` subroutine, `setlocale` subroutine, `wcstod` subroutine, `wcstol` subroutine, `wctomb` subroutine.

The `wchar.h` file.

---

gai_strerror Subroutine

**Purpose**
Facilitates consistent error information from EAI_* values returned by the `getaddrinfo` subroutine.

**Library**
Library (`libc.a`)

**Syntax**
```
#include <sys/socket.h>
#include <netdb.h>
char *
gai_strerror (ecode)
int ecode;
int
```
**gai_strerror_r**(ecode, buf, buflen)

```c
int ecode;
char *buf;
int buflen;
```

### Description

For multithreaded environments, the second version should be used. In **gai_strerror_r**, `buf` is a pointer to a data area to be filled in. `buflen` is the length (in bytes) available in `buf`.

It is the caller's responsibility to insure that `buf` is sufficiently large to store the requested information, including a trailing null character. It is the responsibility of the function to insure that no more than `buflen` bytes are written into `buf`.

### Return Values

If successful, a pointer to a string containing an error message appropriate for the EAI_* errors is returned. If `ecode` is not one of the EAI_* values, a pointer to a string indicating an unknown error is returned.

### Related Information

The [getaddrinfo Subroutine](#), [freeaddrinfo Subroutine](#) and [getnameinfo Subroutine](#) articles in AIX 5L Version 5.3 Technical Reference: Communications Volume 2.

---

**gamma Subroutine**

### Purpose

Computes the natural logarithm of the gamma function.

### Libraries

The **gamma**:

- IEEE Math Library (**libm.a**)
- or System V Math Library (**libmsaa.a**)

### Syntax

```c
#include <math.h>
extern int signgam;
double gamma (x)
double x;
```

### Description

The **gamma** subroutine computes the logarithm of the gamma function.

The sign of `gamma(x)` is returned in the external integer `signgam`.

**Note:** Compile any routine that uses subroutines from the **libm.a** with the `-lm` flag. To compile the `lgamma.c` file, enter:

```bash
c c lgamma.c -lm
```

### Parameters

- `x` Specifies the value to be computed.
Related Information
"exp, expf, or expl Subroutine" on page 244, "f cleare xcept Subroutine" on page 262, "f testexcept Subroutine" on page 270, and "class, _class, finite, isnan, or unordered Subroutines" on page 167.

The exp, expm1, log, log10, log1p or pow subroutine, matherr subroutine.

Subroutines Overview in AIX 5L Version 5.3 General Programming Concepts: Writing and Debugging Programs.

128-Bit long double Floating-Point Format in AIX 5L Version 5.3 General Programming Concepts: Writing and Debugging Programs.

math.h in AIX 5L Version 5.3 Files Reference.

gencore or coredump Subroutine

Purpose
Creates a core file without terminating the process.

Library
Standard C Library (libc.a)

Syntax
#include <core.h>

int gencore (coredumpinfop);
struct coredumpinfo *coredumpinfop;

int coredump (coredumpinfop);
struct coredumpinfo *coredumpinfop;

Description
The gencore and coredump subroutines create a core file of a process without terminating it. The core file contains the snapshot of the process at the time the call is made and can be used with the dbx command for debugging purposes.

If any thread of the process is in a system call when its snapshot core file is generated, the register information returned may not be reliable (except for the stack pointer). To save all user register contents when a system call is made so that they are available to the gencore and coredump subroutines, the application should be built using the "-bM:UR" flags.

If any thread of the process is sleeping inside the kernel or stopped (possibly for job control), the caller of the gencore and coredump subroutines will also be blocked until the thread becomes runnable again. Thus, these subroutines may take a long time to complete depending upon the target process state.

The coredump subroutine always generates a core file for the process from which it is called. This subroutine has been replaced by the gencore subroutine and is being provided for compatibility reasons only.
The `gencore` subroutine creates a core file for the process whose process ID is specified in the `pid` field of the `coredumpinfo` structure. For security measures, the user ID (uid) and group ID (gid) of the core file are set to the uid and gid of the process.

Both these subroutines return success even if the core file cannot be created completely because of filesystem space constraints. When using the `dbx` command with an incomplete core file, `dbx` may warn that the core file is truncated.

In the “Change / Show Characteristics of Operating System” smitty screen, there are two options regarding the creation of the core file. The core file will always be created in the default core format and will ignore the value specified in the "Use pre-430 style CORE dump" option. However, the value specified for the "Enable full CORE dump" option will be considered when creating the core file. Resource limits of the target process for file and `coredump` will be enforced.

The `coredumpinfo` structure contains the following fields:

<table>
<thead>
<tr>
<th>Member Type</th>
<th>Member Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>unsigned int</td>
<td>length</td>
<td>Length of the core file name.</td>
</tr>
<tr>
<td>char *</td>
<td>name</td>
<td>Name of the core file.</td>
</tr>
<tr>
<td>pid_t</td>
<td>pid</td>
<td>ID of the process to be coredumped.</td>
</tr>
<tr>
<td>int</td>
<td>flags</td>
<td>Flags-version flag. Set this to <code>COREGEN_VERSION_1</code>.</td>
</tr>
</tbody>
</table>

Note: The `pid` and `flags` fields are required for the `gencore` subroutine, but are ignored for the `coredump` subroutine

**Parameters**

`coredumpinfop` Specifies the address of the `coredumpinfo` structure that provides the file name to save the core snapshot and its length. For the `gencore` subroutine, it also provides the process id of the process whose core is to be dumped and a flag which includes version flag bits. The version flag value must be set to `COREGEN_VERSION_1`.

**Return Values**

Upon successful completion, the `gencore` and `coredump` subroutines return a 0. If unsuccessful, a -1 is returned, and the `errno` global variable is set to indicate the error.

**Error Codes**

- **EACCESS** Search permission is denied on a component of the path prefix, the file exists and permissions specified by the mode are denied, or the file does not exist and write permission is denied for the parent directory of the file to be created.

- **ENOENT** The `name` field in the `coredumpinfo` parameter points to an empty string.

- **EINTR** The subroutine was interrupted by a signal before it could complete.

- **ENAMETOOLONG** The value of the `length` field in the `coredumpinfop` structure or the length of the absolute path of the specified core file name is greater than `MAXPATHLEN` (as defined in the `sys/param.h` file).

- **EINVAL** The value of the `length` field in the `coredumpinfop` structure is 0.

- **EAGAIN** The target process is already in the middle of another `gencore` or `coredump` subroutine.

- **ENOMEM** Unable to allocate memory resources to complete the subroutine.
In addition to the above, the following errno values can be set when the gencore subroutine is unsuccessful:

**EPERM**
The real or effective user ID of the calling process does not match the real or effective user ID of target process or the calling process does not have root user authority.

**ESRCH**
There is no process whose ID matches the value specified in the pid field of the coredumpinfop parameter or the process is exiting.

**EINVAL**
The flags field in the coredumpinfop parameter is not set to a valid version value.

### Related Information
The [adb Command](#) in *AIX 5L Version 5.3 Commands Reference, Volume 1*.

The [dbx command](#) and [gencore Command](#) in *AIX 5L Version 5.3 Commands Reference, Volume 2*.

The [core file format](#) in *AIX 5L Version 5.3 Files Reference*.

### genpagvalue Subroutine

**Purpose**
Sets the current process credentials.

**Library**
Security Library (libc.a)

**Syntax**
```
#include <pag.h>
int genpagvalue (pag_name, pag_value, pag_flags);
char * pag_name;
uint64_t * pag_value;
int pag_flags;
```

**Description**
The genpagvalue subroutine generates a new PAG value for a given PAG name. For this function to succeed, the PAG name must be registered with the operating system before calling the genpagvalue subroutine. The genpagvalue subroutine is limited to maintaining information about the last generated PAG number and accordingly generating a new number. This service can optionally store the PAG value in the process’s cred structure. It does not monitor the PAG values stored in the cred structure by other means.

The PAG value returned is of size 64 bits. The number of significant bits is determined by the requested PAG type. 32-bit PAGs have 32 significant bits. 64-bit PAGs have 62 significant bits.

A process must have root authority to invoke this function for 32-bit PAG types. Any process may invoke this function for 64-bit PAG types.

The pag_flags parameter with the value PAG_SET_VALUE causes the generated value to be atomically stored in the process’s credentials. The pag_flags parameter with both the PAG_SET_VALUE and PAG_COPY_CRED values set causes the current process’s credentials to be duplicated before the generated value is stored.
Parameters

- **pag_name**: The name parameter is a 1 to 4 character, NULL terminated name for the PAG type. Typical values include afs, dfs, pki and krb5.
- **pag_value**: This pointer points to a buffer where the OS will return the newly generated PAG value.
- **pag_flags**: These flags control the behavior of the getpagvalue subroutine. This must be set to 0 or one or more of the values `PAG_SET_VALUE` or `PAG_COPY_CRED`.

Return Values

A value of 0 is returned upon successful completion. If the genpagvalue subroutine fails a value of -1 is returned and the errno global variable is set to indicate the error.

Error Codes

The genpagvalue subroutine fails if one or more of the following are true:

- **EINVAL**: The PAG value cannot be generated because the named PAG type does not exist as part of the table.
- **EPERM**: The process does not have the correct authority to use the service.

Other errors might be set by subroutines invoked by the genpagvalue subroutine.

Related Information

- `__pag_getid System Call`
- `__pag_getname System Call`
- `__pag_getvalue System Call`
- `__pag_setname System Call`
- `__pag_setvalue System Call`
- `kcred_genpagvalue Kernel Service`
- `kcred_getpagid Kernel Service`
- `kcred_getpagname Kernel Service`


List of Security and Auditing Subroutines in AIX 5L Version 5.3 General Programming Concepts: Writing and Debugging Programs.

get_malloc_log Subroutine

Purpose

Retrieves information about the malloc subsystem.

Syntax

```c
#include <malloc.h>

size_t get_malloc_log (addr, buf, bufsize)
void* addr;
void* buf;
size_t bufsize;
```

Description

The get_malloc_log subroutine retrieves a record of currently active malloc allocations. These records are stored as an array of malloc_log structures, which are copied from the process heap into the buffer specified by the buf parameter. No more than bufsize bytes are copied into the buffer. Only records corresponding to the heap of which addr is a member are copied, unless addr is NULL, in which case records from all heaps are copied. The addr parameter must be either a pointer to space allocated previously by the malloc subsystem or NULL.
Parameters

addr Pointer to a space allocated by the malloc subsystem.
buf Specifies into which buffer the malloc_log structures are stored.
bufsize Specifies the number of bytes that can be copied into the buffer.

Return Values

The get_malloc_log subroutine returns the number of bytes actually transferred into the bufsize parameter. If Malloc Log is not enabled, 0 is returned. If addr is not a pointer allocated by the malloc subsystem, 0 is returned and the errno global variable is set to EINVAL.

Related Information

“malloc, free, realloc, calloc, mallopt, mallinfo, mallinfo_heap, alloca, valloc, or posix_memalign Subroutine” on page 769, and “get_malloc_log Subroutine” on page 336.


get_malloc_log_live Subroutine

Purpose

Provides information about the malloc subsystem.

Syntax

#include <malloc.h>
struct malloc_log* get_malloc_log_live (addr);
void *addr;

Description

The get_malloc_log_live subroutine provides access to a record of currently active malloc allocations. The information is stored as an array of malloc_log structures, which are located in the process heap. This data is volatile and subject to update. The addr parameter must be either a pointer to space allocated previously by the malloc subsystem or NULL.

Parameters

addr Pointer to space allocated previously by the malloc subsystem

Return Values

The get_malloc_log_live subroutine returns a pointer to the process heap at which the records of current malloc allocations are stored. If the addr parameter is NULL, a pointer to the beginning of the array is returned. If addr is a pointer to space allocated previously by the malloc subsystem, the pointer returned corresponds to records of the same heap as addr. If Malloc Log is not enabled, NULL is returned. If addr is not a pointer allocated by the malloc subsystem, NULL is returned and the errno global variable is set to EINVAL.

Related Information

“malloc, free, realloc, calloc, mallopt, mallinfo, mallinfo_heap, alloca, valloc, or posix_memalign Subroutine” on page 769, and “get_malloc_log Subroutine” on page 336.
**get_speed, set_speed, or reset_speed Subroutines**

**Purpose**
Set and get the terminal baud rate.

**Library**
Standard C Library (libc.a)

**Syntax**
```c
#include <sys/str_tty.h>
int get_speed (FileDescriptor)
int set_speed (FileDescriptor, Speed)
int reset_speed (FileDescriptor)
```

**Description**
The baud rate functions `set_speed` subroutine and `get_speed` subroutine are provided to allow the user applications to program any value of the baud rate that is supported by the asynchronous adapter, but that cannot be expressed using the termios subroutines `cfsetospeed`, `cfsetispeed`, `cfgetospeed`, and `cfsgetispeed`. Those subroutines are indeed limited to the set values {BO, B50, ..., B38400} described in `<termios.h>`.

**Interaction with the termios Baud flags:**
If the terminal’s device driver supports these subroutines, it has two interfaces for baud rate manipulation.

**Operation for Baud Rate:**

normal mode: This is the default mode, in which a termios supported speed is in use.

speed-extended mode: This mode is entered either by calling `set_speed` subroutine a non-termios supported speed at the configuration of the line.

In this mode, all the calls to `tcgetattr` subroutine or `TCGETS ioctl` subroutine will have B50 in the returned termios structure.

If `tcsetatt` subroutine or `TCSETS, TCSETAF, TCSETAW ioctl` subroutines is called and attempt to set B50, the actual baud rate is not changed. If is attempts to set any other termios-supported speed, the driver will switch back to the normal mode and the requested baud rate is set. Calling `reset_speed` subroutine is another way to switch back to the normal mode.

**Parameters**
- `FileDescriptor` Specifies an open file descriptor.
- `Speed` The integer value of the requested speed.
**Return Values**

Upon successful completion, `set_speed` and `reset_speed` return a value of 0, and `get_speed` returns a positive integer specifying the current speed of the line. Otherwise, a value of -1 is returned and the `errno` global variable is set to indicate the error.

**Error Codes**

**EINVAL**  
The `FileDescriptor` parameter does not specify a valid file descriptor for a `tty` the recognizes the `set_speed`, `get_speed` and `reset_speed` subroutines, or the `Speed` parameter of `set_speed` is not supported by the terminal.

Plus all the `errno` codes that may be set in case of failure in an `ioctl` subroutine issued to a streams based `tty`.

**Related Information**

`cfgetospeed`, `cfsetospeed`, `cfgetispeed`, or `cfsetispeed`  
(`cfgetospeed, cfsetospeed, cfgetispeed, or cfsetispeed Subroutine" on page 142) subroutines.

---

**getargs Subroutine**

**Purpose**

Gets arguments of a process.

**Library**

Standard C library (`libc.a`)

**Syntax**

```c
#include <procinfo.h>
#include <sys/types.h>

int getargs (processBuffer, bufferLen, argsBuffer, argsLen)
struct procsinfo *processBuffer
or struct procentry64 *processBuffer;
int bufferLen;
char *argsBuffer;
int argsLen;
```

**Description**

The `getargs` subroutine returns a list of parameters that were passed to a command when it was started. Only one process can be examined per call to `getargs`.

The `getargs` subroutine uses the `pi_pid` field of `processBuffer` to determine which process to look for. `bufferLen` should be set to the size of `struct procsinfo` or `struct procentry64`. Parameters are returned in `argsBuffer`, which should be allocated by the caller. The size of this array must be given in `argsLen`.

On return, `argsBuffer` consists of a succession of strings, each terminated with a null character (ascii `\0`). Hence, two consecutive NULLs indicate the end of the list.

**Note:** The arguments may be changed asynchronously by the process, but results are not guaranteed to be consistent.
Parameters

processBuffer
Specifies the address of a procsinfo or procentry64 structure, whose pi_pid field should contain the pid of the process that is to be looked for.

bufferLen
Specifies the size of a single procsinfo or procentry64 structure.

argsBuffer
Specifies the address of an array of characters to be filled with a series of strings representing the parameters that are needed. An extra NULL character marks the end of the list. This array must be allocated by the caller.

argsLen
Specifies the size of the argsBuffer array. No more than argsLen characters are returned.

Return Values
If successful, the getargs subroutine returns zero. Otherwise, a value of -1 is returned and the errno global variable is set to indicate the error.

Error Codes
The getargs subroutine does not succeed if the following are true:

ESRCH
The specified process does not exist.

EFAULT
The copy operation to the buffer was not successful or the processBuffer or argsBuffer parameters are invalid.

EINVAL
The bufferLen parameter does not contain the size of a single procsinfo or procentry64 structure.

ENOMEM
There is no memory available in the address space.

Related Information
The getevars ("getevars Subroutine" on page 361), getpid ("getpid, getpgrp, or getppid Subroutine" on page 402), getpgrp ("getpid, getpgrp, or getppid Subroutine" on page 402), getppid ("getpid, getpgrp, or getppid Subroutine" on page 402), getprocs or getthrds ("getthrds Subroutine" on page 438) subroutines.

The ps command.

getaudithostattr, IDtohost, hosttoID, nexthost or putaudithostattr Subroutine

Purpose
Accesses the host information in the audit host database.

Library
Security Library (libc.a)

Syntax
#include <usersec.h>

int getaudithostattr (Hostname, Attribute, Value, Type)
char *Hostname;
char *Attribute;
void *Value;

getaudithostattr, IDtohost, hosttoID, nexthost or putaudithostattr Subroutine

Purpose
Accesses the host information in the audit host database.

Library
Security Library (libc.a)

Syntax
#include <usersec.h>

int getaudithostattr (Hostname, Attribute, Value, Type)
char *Hostname;
char *Attribute;
void *Value;
Description

These subroutines access the audit host information.

The `getaudithostattr` subroutine reads a specified attribute from the host database. If the database is not already open, this subroutine does an implicit open for reading.

Similarly the `putaudithostattr` subroutine writes a specified attribute into the host database. If the database is not already open, this subroutine does an implicit open for reading and writing. Data changed by the `putaudithostattr` must be explicitly committed by calling the `putaudithostattr` subroutine with a Type of `SEC_COMMIT`. Until all the data is committed, only these subroutines within the process return written data.

New entries in the host database must first be created by invoking `putaudithostattr` with the `SEC_NEW` type.

The `IDtohost` subroutine converts an 8 byte host identifier into a hostname.

The `hosttoID` subroutine converts a hostname to a pointer to an array of valid 8 byte host identifiers. A pointer to the array of identifiers is returned on success. A NULL pointer is returned on failure. The number of known host identifiers is returned in `*Count`.

The `nexthost` subroutine returns a pointer to the name of the next host in the audit host database.

Parameters

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Specifies which attribute is read. The following possible attributes are defined in the <code>usersec.h</code> file:</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>Attribute</code></td>
<td><code>S_AUD_CPUID</code> Host identifier list. The attribute type is <code>SEC_LIST</code>.</td>
</tr>
<tr>
<td><code>Count</code></td>
<td>Specifies the number of 8 byte host identifier entries that are available in the <code>IDArray</code> parameter or that have been returned in the <code>IDArray</code> parameter.</td>
</tr>
<tr>
<td><code>Hostname</code></td>
<td>Specifies the name of the host for the operation.</td>
</tr>
<tr>
<td><code>ID</code></td>
<td>An 8 byte host identifier.</td>
</tr>
<tr>
<td><code>IDArray</code></td>
<td>Specifies a pointer to an array of 1 or more 8 byte host identifiers.</td>
</tr>
<tr>
<td><code>Type</code></td>
<td>Specifies the type of attribute expected. Valid types are defined in <code>usersec.h</code>. The only valid Type value is <code>SEC_LIST</code>.</td>
</tr>
</tbody>
</table>
Value

The return value for read operations and the new value for write operations.

Return Values
On successful completion, the getaudithostattr, IDtohost, hosttoID, nexthost, or putaudithostattr subroutine returns 0. If unsuccessful, the subroutine returns non-zero.

Error Codes
The getaudithostattr, IDtohost, hosttoID, nexthost, or putaudithostattr subroutine fails if the following is true:

EINVAL
If invalid attribute Name or if Count is equal to zero for the hosttoID subroutine.

ENOENT
If there is no matching Hostname entry in the database.

Related Information
The auditmerge command, auditpr command, auditselect command, auditstream command.

The auditread ("auditread, auditread_r Subroutines" on page 111) subroutine.

getauthdb or getauthdb_r Subroutine

Purpose
Finds the current administrative domain.

Library
Standard C Library (libc.a)

Syntax
#include <usersec.h>

int getauthdb (Value)
authdb_t *Value;

int getauthdb_r (Value)
authdb_t *Value;

Description
The getauthdb and getauthdb_r subroutines return the value of the current authentication domain in the Value parameter. The getauthdb subroutine returns the value of the current process-wide authentication domain. The getauthdb_r subroutine returns the authentication domain for the current thread if one has been set. The subroutines return -1 if no administrative domain has been set.

Parameters
| Value | A pointer to a variable of type authdb_t. The authdb_t type is a 16-character array that contains the name of a loadable authentication module. |
Return Values

1
The value returned is from the process-wide data.

0
The value returned is from the thread-specific data. An authentication database module has been specified by an earlier call to the `setauthdb` subroutine. The name of the current database module has been copied to the `Value` parameter.

-1
The subroutine failed. An authentication database module has not been specified by an earlier call to the `setauthdb` subroutine.

Related Information


getc, getchar, fgetc, or getw Subroutine

Purpose

Gets a character or word from an input stream.

Library

Standard I/O Package (libc.a)

Syntax

```c
#include <stdio.h>

int getc (Stream)
FILE *Stream;
int fgetc (Stream)
FILE *Stream;
int getchar (void)
int getw (Stream)
FILE *Stream;
```

Description

The `getc` macro returns the next byte as an `unsigned char` data type converted to an `int` data type from the input specified by the `Stream` parameter and moves the file pointer, if defined, ahead one byte in the `Stream` parameter. The `getc` macro cannot be used where a subroutine is necessary; for example, a subroutine pointer cannot point to it.

Because it is implemented as a macro, the `getc` macro does not work correctly with a `Stream` parameter value that has side effects. In particular, the following does not work:
```
getc(*f++)
```

In such cases, use the `fgetc` subroutine.

The `fgetc` subroutine performs the same function as the `getc` macro, but `fgetc` is a true subroutine, not a macro. The `fgetc` subroutine runs more slowly than `getc` but takes less disk space.

The `getchar` macro returns the next byte from `stdin` (the standard input stream). The `getchar` macro is equivalent to `getc(stdin)`.
The first successful run of the `fgetc`, `fgets`, `fgetwc`, `fgetws`, `fread`, `fscanf`, `getc`, `getchar`, `gets` or `scanf` subroutine using a stream that returns data not supplied by a prior call to the `ungetc` or `ungetwc` subroutine marks the `st_atime` field for update.

The `getc` and `getchar` macros have also been implemented as subroutines for ANSI compatibility. To access the subroutines instead of the macros, insert `#undef getc` or `#undef getchar` at the beginning of the source file.

The `getw` subroutine returns the next word (int) from the input specified by the `Stream` parameter and increments the associated file pointer, if defined, to point to the next word. The size of a word varies from one machine architecture to another. The `getw` subroutine returns the constant `EOF` at the end of the file or when an error occurs. Since `EOF` is a valid integer value, the `feof` and `ferror` subroutines should be used to check the success of `getw`. The `getw` subroutine assumes no special alignment in the file.

Because of additional differences in word length and byte ordering from one machine architecture to another, files written using the `putw` subroutine are machine-dependent and may not be readable using the `getw` macro on a different type of processor.

**Parameters**

- `Stream`: Points to the file structure of an open file.

**Return Values**

Upon successful completion, the `getc`, `fgetc`, `getchar`, and `getw` subroutines return the next byte or `int` data type from the input stream pointed by the `Stream` parameter. If the stream is at the end of the file, an end-of-file indicator is set for the stream and the integer constant `EOF` is returned. If a read error occurs, the `errno` global variable is set to reflect the error, and a value of `EOF` is returned. The `ferror` and `feof` subroutines should be used to distinguish between the end of the file and an error condition.

**Error Codes**

If the stream specified by the `Stream` parameter is unbuffered or data needs to be read into the stream’s buffer, the `getc`, `getchar`, `fgetc`, or `getw` subroutine is unsuccessful under the following error conditions:

- **EAGAIN**: Indicates that the `O_NONBLOCK` flag is set for the file descriptor underlying the stream specified by the `Stream` parameter. The process would be delayed in the `fgetc` subroutine operation.
- **EBADF**: Indicates that the file descriptor underlying the stream specified by the `Stream` parameter is not a valid file descriptor opened for reading.
- **EFBIG**: Indicates that an attempt was made to read a file that exceeds the process’ file-size limit or the maximum file size. See the `ulimit` subroutine.
- **EINTR**: Indicates that the read operation was terminated due to the receipt of a signal, and either no data was transferred, or the implementation does not report partial transfer for this file. **Note**: Depending upon which library routine the application binds to, this subroutine may return `EINTR`. Refer to the `signal` subroutine regarding `sa_restart`.
- **EIO**: Indicates that a physical error has occurred, or the process is in a background process group attempting to perform a `read` subroutine call from its controlling terminal, and either the process is ignoring (or blocking) the `SIGTTIN` signal or the process group is orphaned.
- **EPIPE**: Indicates that an attempt is made to read from a pipe or first-in-first-out (FIFO) that is not open for reading by any process. A `SIGPIPE` signal will also be sent to the process.
- **EOVERFLOW**: Indicates that the file is a regular file and an attempt was made to read at or beyond the offset maximum associated with the corresponding stream.
The `getc`, `getchar`, `fgetc`, or `getw` subroutine is also unsuccessful under the following error conditions:

- **ENOMEM** Indicates insufficient storage space is available.
- **ENXIO** Indicates either a request was made of a nonexistent device or the request was outside the capabilities of the device.

**Related Information**

The `feof`, `ferror`, `clearerr`, or `fileno` subroutine, `fopen`, `fopen64`, or `fdopen` subroutine, `fread` or `fwrite` subroutine, `getwc`, `fgetwc`, or `getwchar` subroutine, `get` or `fgets` subroutine, `putc`, `putchar`, `fputc`, or `putw` subroutine, `scanf`, `sscanf`, `fscanf`, or `wscanf` subroutine.

**getc_unlocked, getchar_unlocked, putc_unlocked, putchar_unlocked Subroutines**

**Purpose**

stdio with explicit client locking.

**Library**

Standard Library (`libc.a`)

**Syntax**

```c
#include <stdio.h>
int getc_unlocked (FILE * stream);
int getchar_unlocked (void);
int putc_unlocked (int c, FILE * stream);
int putchar_unlocked (int c);
```

**Description**

Versions of the functions `getc`, `getchar`, `putc`, and `putchar` respectively named `getc_unlocked`, `getchar_unlocked`, `putc_unlocked`, and `putchar_unlocked` are provided which are functionally identical to the original versions with the exception that they are not required to be implemented in a thread-safe manner. They may only safely be used within a scope protected by `flockfile` (or `ftrylockfile`) and `funlockfile`. These functions may safely be used in a multi-threaded program if and only if they are called while the invoking thread owns the (FILE*) object, as is the case after a successful call of the `flockfile` or `ftrylockfile` functions.

**Return Values**

See `getc`, `getchar`, `putc`, and `putchar`.

**Related Information**

The `getc` or `getchar` subroutine, `putc` or `putchar` subroutine.
getconfattr or putconfattr Subroutine

**Purpose**
Accesses the system information in the user database.

**Library**
Security Library (libc.a)

**Syntax**
```
#include <usersec.h>
#include <userconf.h>

int getconfattr (sys, Attribute, Value, Type)
char * sys;
char * Attribute;
void * Value;
int  Type;

int putconfattr (sys, Attribute, Value, Type)
char * sys;
char * Attribute;
void * Value;
int  Type;
```

**Description**
The `getconfattr` subroutine reads a specified attribute from the user database. The `putconfattr` subroutine writes a specified attribute to the user database.

**Parameters**
- **sys** System attribute. The following possible attributes are defined in the `userconf.h` file:
  - SC_SYS_LOGIN
  - SC_SYS_USER
  - SC_SYS_ADMUSER
  - SC_SYS_AUDIT
  - SC_SYS_AUSERS
  - SC_SYS_ASYS
  - SC_SYS_ABIN
  - SC_SYS_ASTREAM

- **Attribute** Specifies which attribute is read. The following possible attributes are defined in the `usersec.h` file:
  - S_CORECOMP Core compression status. The attribute type is `SEC_CHAR`.
  - S_COREPATH Core path specification status. The attribute type is `SEC_CHAR`.
  - S_COREPNAME Core path specification location. The attribute type is `SEC_CHAR`.
  - S_CORENAMING Core naming status. The attribute type is `SEC_CHAR`. 
S_ID  User ID. The attribute type is SEC_INT.

S_PGRP  Principle group name. The attribute type is SEC_CHAR.

S_GROUPS  Groups to which the user belongs. The attribute type is SEC_LIST.

S_ADMGROUPS  Groups for which the user is an administrator. The attribute type is SEC_LIST.

S_ADMIN  Administrative status of a user. The attribute type is SEC_BOOL.

S_AUDITCLASSES  Audit classes to which the user belongs. The attribute type is SEC_LIST.

S_AUTHSYSTEM  Defines the user's authentication method. The attribute type is SEC_CHAR.

S_HOME  Home directory. The attribute type is SEC_CHAR.

S_SHELL  Initial program run by a user. The attribute type is SEC_CHAR.

S_GECOS  Personal information for a user. The attribute type is SEC_CHAR.

S_USRENV  User-state environment variables. The attribute type is SEC_LIST.

S_SYSENV  Protected-state environment variables. The attribute type is SEC_LIST.

S_LOGINCHK  Specifies whether the user account can be used for local logins. The attribute type is SEC_BOOL.

S_HISTEXPIRE  Defines the period of time (in weeks) that a user cannot reuse a password. The attribute type is SEC_INT.

S_HISTSIZE  Specifies the number of previous passwords that the user cannot reuse. The attribute type is SEC_INT.

S_MAXREPEAT  Defines the maximum number of times a user can repeat a character in a new password. The attribute type is SEC_INT.

S_MINAGE  Defines the minimum age in weeks that the user's password must exist before the user can change it. The attribute type is SEC_INT.

S_PWDCHECKS  Defines the password restriction methods for this account. The attribute type is SEC_LIST.

S_MINALPHA  Defines the minimum number of alphabetic characters required in a new user's password. The attribute type is SEC_INT.

S_MINDIFF  Defines the minimum number of characters required in a new password that were not in the old password. The attribute type is SEC_INT.
S_MINLEN
Defines the minimum length of a user’s password. The attribute type is SEC_INT.

S_MINOTHER
Defines the minimum number of non-alphabetic characters required in a new user’s password. The attribute type is SEC_INT.

S_DICCTIONLIST
Defines the password dictionaries for this account. The attribute type is SEC_LIST.

S_SUCHK
Specifies whether the user account can be accessed with the su command. Type SEC_BOOL.

S_REGISTRY
Defines the user’s authentication registry. The attribute type is SEC_CHAR.

S_RLOGINCHK
Specifies whether the user account can be used for remote logins using the telnet or rlogin commands. The attribute type is SEC_BOOL.

S_DAEMONCHK
Specifies whether the user account can be used for daemon execution of programs and subsystems using the cron daemon or src. The attribute type is SEC_BOOL.

S_TPATH
Defines how the account may be used on the trusted path. The attribute type is SEC_CHAR. This attribute must be one of the following values:

- nosak  The secure attention key is not enabled for this account.
- notsh  The trusted shell cannot be accessed from this account.
- always This account may only run trusted programs.
- on     Normal trusted-path processing applies.

S_TTYS
List of ttys that can or cannot be used to access this account. The attribute type is SEC_LIST.

S_SUGROUPS
Groups that can or cannot access this account. The attribute type is SEC_LIST.

S_EXPIRATION
Expiration date for this account, in seconds since the epoch. The attribute type is SEC_CHAR.

S_AUTH1
Primary authentication methods for this account. The attribute type is SEC_LIST.

S_AUTH2
Secondary authentication methods for this account. The attribute type is SEC_LIST.

S_UFSIZE
Process file size soft limit. The attribute type is SEC_INT.

S_UCPU
Process CPU time soft limit. The attribute type is SEC_INT.

S_UDATA
Process data segment size soft limit. The attribute type is SEC_INT.
S_USTACK
Process stack segment size soft limit. Type: SEC_INT.

S URSS
Process real memory size soft limit. Type: SEC_INT.

S_UCORE
Process core file size soft limit. The attribute type is SEC_INT.

S_PWD
Specifies the value of the passwd field in the /etc/passwd file. The attribute type is SEC_CHAR.

S_UMASK
File creation mask for a user. The attribute type is SEC_INT.

S_LOCKED
Specifies whether the user’s account can be logged into. The attribute type is SEC_BOOL.

S_UFSIZE_HARD
Process file size hard limit. The attribute type is SEC_INT.

S_UCPU_HARD
Process CPU time hard limit. The attribute type is SEC_INT.

S_UDATA_HARD
Process data segment size hard limit. The attribute type is SEC_INT.

S_USTACK_HARD
Process stack segment size hard limit. Type: SEC_INT.

S_URSS_HARD
Process real memory size hard limit. Type: SEC_INT.

S_UCORE_HARD
Process core file size hard limit. The attribute type is SEC_INT.

Note: These values are string constants that should be used by applications both for convenience and to permit optimization in latter implementations.

Type
Specifies the type of attribute expected. Valid types are defined in the usersec.h file and include:

SEC_INT
The format of the attribute is an integer.

For the getconfattr subroutine, the user should supply a pointer to a defined integer variable. For the putconfattr subroutine, the user should supply an integer.

SEC_CHAR
The format of the attribute is a null-terminated character string.

SEC_LIST
The format of the attribute is a series of concatenated strings, each null-terminated. The last string in the series is terminated by two successive null characters.

SEC_BOOL
The format of the attribute from the getconfattr subroutine is an integer with the value of either 0 (false) or 1 (true). The format of the attribute for the putconfattr subroutine is a null-terminated string containing one of the following strings: true, false, yes, no, always, or never.

Note: These values are string constants that should be used by applications both for convenience and to permit optimization in latter implementations.
SEC_COMMIT
For the putconfattr subroutine, this value specified by itself indicates that changes to the named user are to be committed to permanent storage. The Attribute and Value parameters are ignored. If no user is specified, the changes to all modified users are committed to permanent storage.

SEC_DELETE
The corresponding attribute is deleted from the database.

SEC_NEW
Updates all the user database files with the new user name when using the putconfattr subroutine.

Security
Files Accessed:

<table>
<thead>
<tr>
<th>Mode</th>
<th>File</th>
</tr>
</thead>
<tbody>
<tr>
<td>rw</td>
<td>/etc/security/user</td>
</tr>
<tr>
<td>rw</td>
<td>/etc/security/limits</td>
</tr>
<tr>
<td>rw</td>
<td>/etc/security/login.cfg</td>
</tr>
<tr>
<td>rw</td>
<td>/usr/lib/security/mkuser.default</td>
</tr>
<tr>
<td>rw</td>
<td>/etc/security/audit/config</td>
</tr>
</tbody>
</table>

Return Values
If successful, returns 0
If unsuccessful, returns -1

Error Codes
ENOENT The specified Sys parameter does not exist or the attribute is not defined for this Sys parameter.

Files
/etc/passwd Contains user IDs.

Related Information
The "getuserattr, IDtouser, nextuser, or putuserattr Subroutine" on page 449.

List of Security and Auditing Subroutines, Subroutines Overview in AIX 5L Version 5.3 General Programming Concepts: Writing and Debugging Programs.

getconfattrs Subroutine

Purpose
Accesses system information in the system information database.
Library
Security Library (libc.a)

Syntax
#include <usersec.h>
#include <userconf.h>

int getconfattrs (Sys, Attributes, Count)
char *Sys;
dbattr_t *Attributes;
int Count

Description
The getconfattrs subroutine accesses system configuration information.

The getconfattrs subroutine reads one or more attributes from the system configuration database. If the
database is not already open, this subroutine does an implicit open for reading.

The Attributes array contains information about each attribute that is to be written. The dbattr_t data
structure contains the following fields:

attr_name
  The name of the desired attribute.

attr_idx
  Used internally by the getconfattrs subroutine.

attr_type
  The type of the desired attribute. The list of attribute types is defined in the usersec.h header file.

attr_flag
  The results of the request to read the desired attribute.

attr_un
  A union containing the values to be written. Its union members that follow correspond to the
definitions of the attr_char, attr_int, attr_long, and attr_llong macros, respectively:

  un_char
    Attributes of type SEC_CHAR and SEC_LIST store a pointer to the value to be written.

  un_int
    Attributes of type SEC_INT and SEC_BOOL contain the value of the attribute to be
    written.

  un_long
    Attributes of type SEC_LONG contain the value of the attribute to be written.

  un_llong
    Attributes of type SEC_LLONG contain the value of the attribute to be written.

attr_domain
  The authentication domain containing the attribute. The getconfattrs subroutine is responsible for
  managing the memory referenced by this pointer.

Use the setuserdb and enduserdb subroutines to open and close the system configuration database.
Failure to explicitly open and close the system database can result in loss of memory and performance.

Parameters
Sys
  Specifies the name of the subsystem for which the attributes are to be read.
Attributes
A pointer to an array of one or more elements of type `dbattr_t`. The list of system attributes is defined in the `usersec.h` header file.

Count
The number of array elements in Attributes.

Security
Files accessed:

<table>
<thead>
<tr>
<th>Mode</th>
<th>File</th>
</tr>
</thead>
<tbody>
<tr>
<td>r</td>
<td><code>/etc/security/.ids</code></td>
</tr>
<tr>
<td>r</td>
<td><code>/etc/security/audit/config</code></td>
</tr>
<tr>
<td>r</td>
<td><code>/etc/security/audit/events</code></td>
</tr>
<tr>
<td>r</td>
<td><code>/etc/security/audit/objects</code></td>
</tr>
<tr>
<td>r</td>
<td><code>/etc/security/login.cfg</code></td>
</tr>
<tr>
<td>r</td>
<td><code>/etc/security/portlog</code></td>
</tr>
<tr>
<td>r</td>
<td><code>/etc/security/roles</code></td>
</tr>
<tr>
<td>r</td>
<td><code>/usr/lib/security/methods.cfg</code></td>
</tr>
<tr>
<td>r</td>
<td><code>/usr/lib/security/mkuser.default</code></td>
</tr>
</tbody>
</table>

Return Values
If the named subsystem is valid, or the Sys attribute references an existing user or group for those attributes where the Sys parameter is a named user or group, the `getconfattrs` subroutine returns 0. Otherwise, a value of -1 is returned and the `errno` global variable is set to indicate the error. Each element in the Attributes array must be examined on a successful call to `getconfattrs` to determine if the Attributes array entry was successfully retrieved.

Error Codes
The `getconfattrs` subroutine returns an error that indicates that the system attribute does or does not exist. Additional errors can indicate an error querying the information databases for the requested attributes.

- `EINVAL` The Attributes parameter is NULL.
- `EINVAL` The Count parameter is less than 1.
- `ENOENT` The specified Sys does not exist.

If the `getconfattrs` subroutine fails to query an attribute, one or more of the following errors is returned in the attr_flag field of the corresponding Attributes element:

- `EACCESS` The user does not have access to the attribute specified in the attr_name field.
- `EINVAL` The attr_type field in the Attributes entry contains an invalid type.
- `EINVAL` The attr_un field in the Attributes entry does not point to a valid buffer or to valid data for this type of attribute. Limited testing is possible and all errors might not be detected.
- `ENOMEM` Memory could not be allocated to store the return value.
- `ENOATTR` The attr_name field in the Attributes entry specifies an attribute that is not defined for this system table.

Files

- `/etc/security/.ids` The next available user and group ID values.
- `/etc/security/audit/config` Bin and stream mode audit configuration parameters.
- `/etc/security/audit/events` Format strings for audit event records.
- `/etc/security/audit/objects` File system objects that are explicitly audited.
getcontext or setcontext Subroutine

Purpose
Initializes the structure pointed to by \textit{ucp} to the context of the calling process.

Library
(libc.a)

Syntax

\begin{verbatim}
#include <ucontext.h>

int getcontext (ucontext_t *ucp);

int setcontext (const uncontext_t *ucp);
\end{verbatim}

Description
The \texttt{getcontext} subroutine initializes the structure pointed to by \textit{ucp} to the current user context of the calling process. The \texttt{ucontext_t} type that \textit{ucp} points to defines the user context and includes the contents of the calling process' machine registers, the signal mask, and the current execution stack.

The \texttt{setcontext} subroutine restores the user context pointed to by \textit{ucp}. A successful call to \texttt{setcontext} subroutine does not return; program execution resumes at the point specified by the \textit{ucp} argument passed to \texttt{setcontext} subroutine. The \textit{ucp} argument should be created either by a prior call to \texttt{getcontext} subroutine, or by being passed as an argument to a signal handler. If the \textit{ucp} argument was created with \texttt{getcontext} subroutine, program execution continues as if the corresponding call of \texttt{getcontext} subroutine had just returned. If the \textit{ucp} argument was created with \texttt{makecontext} subroutine, program execution continues with the function passed to \texttt{makecontext} subroutine. When that function returns, the process continues as if after a call to \texttt{setcontext} subroutine with the \textit{ucp} argument that was input to \texttt{makecontext} subroutine. If the \textit{ucp} argument was passed to a signal handler, program execution continues with the program instruction following the instruction interrupted by the signal. If the \texttt{ucp} argument is equal to 0, then this context is the main context, and the process will exit when this context returns.
Parameters

ucp
A pointer to a user structure.

Return Values

If successful, a value of 0 is returned. If unsuccessful, a value of -1 is returned and the errno global variable is set to indicate the error.

Error Codes

The getcontext and setcontext subroutines are unsuccessful if one of the following is true:

<table>
<thead>
<tr>
<th>Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EINVAL</td>
<td>NULL ucp address</td>
</tr>
<tr>
<td>EFAULT</td>
<td>Invalid ucp address</td>
</tr>
</tbody>
</table>

Related Information

The makecontext subroutine, setjmp subroutine, sigaltstack subroutine, sigaction subroutine, sigprocmask subroutine, and sigsetjmp subroutine.

getcwd Subroutine

Purpose

Gets the path name of the current directory.

Library

Standard C Library (libc.a)

Syntax

```c
#include <unistd.h>

char *getcwd (Buffer, Size);
char *Buffer;
size_t Size;
```

Description

The getcwd subroutine places the absolute path name of the current working directory in the array pointed to by the Buffer parameter, and returns that path name. The size parameter specifies the size in bytes of the character array pointed to by the Buffer parameter.

Parameters

Buffer
Points to string space that will contain the path name. If the Buffer parameter value is a null pointer, the getcwd subroutine, using the malloc subroutine, obtains the number of bytes of free space as specified by the Size parameter. In this case, the pointer returned by the getcwd subroutine can be used as the parameter in a subsequent call to the free subroutine. Starting the getcwd subroutine with a null pointer as the Buffer parameter value is not recommended.

Size
Specifies the length of the string space. The value of the Size parameter must be at least 1 greater than the length of the path name to be returned.
Return Values
If the getcwd subroutine is unsuccessful, a null value is returned and the errno global variable is set to indicate the error. The getcwd subroutine is unsuccessful if the Size parameter is not large enough or if an error occurs in a lower-level function.

Error Codes
If the getcwd subroutine is unsuccessful, it returns one or more of the following error codes:

- **EACCES** Indicates that read or search permission was denied for a component of the path name
- **EINVAL** Indicates that the Size parameter is 0 or a negative number.
- **ENOMEM** Indicates that insufficient storage space is available.
- **ERANGE** Indicates that the Size parameter is greater than 0, but is smaller than the length of the path name plus 1.

Related Information
The getwd subroutine, malloc subroutine, mmap, malloc, free, realloc, calloc, malloc, malloc, mallinfo, mallinfo_heap, alloca, valloc, or posix_memalign subroutine.

Files, Directories, and File Systems for Programmers in AIX 5L Version 5.3 General Programming Concepts: Writing and Debugging Programs.

getdate Subroutine

Purpose
Convert user format date and time.

Library
Standard C Library (libc.a)

Syntax
```
#include <time.h>

struct tm *getdate (const char *string)
extern int getdate_err
```

Description
The getdate subroutine converts user definable date and/or time specifications pointed to by string, into a struct tm. The structure declaration is in the time.h header file (see ctime subroutine).

User supplied templates are used to parse and interpret the input string. The templates are contained in text files created by the user and identified by the environment variable DATEMSK. The DATEMSK variable should be set to indicate the full pathname of the file that contains the templates. The first line in the template that matches the input specification is used for interpretation and conversation into the internal time format.

The templates should follow a format where complex field descriptors are preceded by simpler ones. For example:

- %M
- %H:%M
- %m/%d/%y
- %m/%d/%Y %H:%M:%S

getdate Subroutine

Purpose
Convert user format date and time.

Library
Standard C Library (libc.a)

Syntax
```
#include <time.h>

struct tm *getdate (const char *string)
extern int getdate_err
```

Description
The getdate subroutine converts user definable date and/or time specifications pointed to by string, into a struct tm. The structure declaration is in the time.h header file (see ctime subroutine).

User supplied templates are used to parse and interpret the input string. The templates are contained in text files created by the user and identified by the environment variable DATEMSK. The DATEMSK variable should be set to indicate the full pathname of the file that contains the templates. The first line in the template that matches the input specification is used for interpretation and conversation into the internal time format.

The templates should follow a format where complex field descriptors are preceded by simpler ones. For example:

- %M
- %H:%M
- %m/%d/%y
- %m/%d/%Y %H:%M:%S
The following field descriptors are supported:

%%  Same as %.
%a  Abbreviated weekday name.
%A  Full weekday name.
%b  Abbreviated month name.
%B  Full month name.
%c  Locale's appropriate date and time representation.
%C  Century number (00-99; leading zeros are permitted but not required)
%d  Day of month (01 - 31: the leading zero is optional.
%e  Same as %d.
%D  Date as %m/%d/%y.
%h  Abbreviated month name.
%H  Hour (00 - 23)
%i  Hour (01 - 12)
%m  Month number (01 - 12)
%M  Minute (00 - 59)
%n  Same as \n.
%p  Locale's equivalent of either AM or PM.
%r  Time as %I:%M:%S %p
%R  Time as %H: %M
%S  Seconds (00 - 61) Leap seconds are allowed but are not predictable through use of algorithms.
%s  Same as tab.
%t  Time as %H: %M:%S
%w  Weekday number (Sunday = 0 - 6)
%x  Locale's appropriate date representation.
%X  Locale's appropriate time representation.
%y  Year within century.
%Y  Year as ccyy (such as 1986)
%Z  Time zone name or no characters if no time zone exists. If the time zone supplied by %Z is not the same as the time zone getdate subroutine expects, an invalid input specification error will result. The getdate subroutine calculates an expected time zone based on information supplied to the interface (such as hour, day, and month).

The match between the template and input specification performed by the getdate subroutine is case sensitive.

The month and weekday names can consist of any combination of upper and lower case letters. The used can request that the input date or time specification be in a specific language by setting the LC_TIME category (See the setlocale subroutine).

Leading zero's are not necessary for the descriptors that allow leading zero's. However, at most two digits are allowed for those descriptors, including leading zero's. Extra whitespace in either the template file or in string is ignored.

The field descriptors %c, %x, and %X will not be supported if they include unsupported field descriptors.

Example 1 is an example of a template. Example 2 contains valid input specifications for the template. Example 3 shows how local date and time specifications can be defined in the template.

The following rules apply for converting the input specification into the internal format:
If only the weekday is given, today is assumed if the given month is equal to the current day and next week if it is less.

If only the month is given, the current month is assumed if the given month is equal to the current month and next year if it is less and no year is given (the first day of month is assumed if no day is given).

If no hour, minute, and second are given, the current hour, minute and second are assumed.

If no date is given, today is assumed if the given hour is greater than the current hour and tomorrow is assumed if it is less.

**Return Values**

Upon successful completion, the `getdate` subroutine returns a pointer to `struct tm`; otherwise, it returns a null pointer and the external variable `getdate_err` is set to indicate the error.

**Error Codes**

Upon failure, a null pointer is returned and the variable `getdate_err` is set to indicate the error.

The following is a complete list of the `getdate_err` settings and their corresponding descriptions:

1. The `DATEMSK` environment variable is null or undefined.
2. The template file cannot be opened for reading.
3. Failed to get file status information.
4. The template file is not a regular file.
5. An error is encountered while reading the template file.
6. Memory allocation failed (not enough memory available.
7. There is no line in the template that matches the input.
8. Invalid input specification, Example: February 31 or a time is specified that can not be represented in a time_t (representing the time in seconds since 00:00:00 UTC, January 1, 1970).

**Examples**

1. The following example shows the possible contents of a template:

   ```
   %m
   %A %B %d, %Y, %H:%M:%S
   %A
   %B
   %m/%d/%y %I %p
   %d, %m, %Y %H:%M
   at %A the %dst of %B in %Y
   run job at %I %p, %B %dnd
   &A den %d, %B %Y %H.%M Uhr
   ```

2. The following are examples of valid input specifications for the template in Example 1:

   ```
   getdate ("10/1/87 4 PM")
   getdate ("Friday")
   getdate ("Friday September 18, 1987, 10:30:30")
   getdate ("24,9,1986 10:30")
   getdate ("at monday the 1st of december in 1986")
   getdate ("run job at 3 PM. december 2nd")
   ```

   If the LC_TIME category is set to a German locale that includes freitag as a weekday name and oktober as a month name, the following would be valid:

   ```
   getdate ("freitag den 10. oktober 1986 10.30 Uhr")
   ```

3. The following examples shows how local date and time specification can be defined in the template.

<table>
<thead>
<tr>
<th>Invocation</th>
<th>Line in Template</th>
</tr>
</thead>
<tbody>
<tr>
<td>getdate (&quot;11/27/86&quot;)</td>
<td>%m/%d/%y</td>
</tr>
</tbody>
</table>
The following examples help to illustrate the above rules assuming that the current date Mon Sep 22 12:19:47 EDT 1986 and the LC_TIME category is set to the default "C" locale.

<table>
<thead>
<tr>
<th>Input</th>
<th>Line in Template</th>
<th>Date</th>
</tr>
</thead>
<tbody>
<tr>
<td>Mon</td>
<td>%a</td>
<td>Mon Sep 22 12:19:47 EDT 1986</td>
</tr>
<tr>
<td>Sun</td>
<td>%a</td>
<td>Sun Sep 28 12:19:47 EDT 1986</td>
</tr>
<tr>
<td>Fri</td>
<td>%a</td>
<td>Fri Sep 26 12:19:47 EDT 1986</td>
</tr>
<tr>
<td>September</td>
<td>%B</td>
<td>Mon Sep 1 12:19:47 EDT 1986</td>
</tr>
<tr>
<td>January</td>
<td>%B</td>
<td>Thu Jan 1 12:19:47 EDT 1986</td>
</tr>
<tr>
<td>December</td>
<td>%B</td>
<td>Mon Dec 1 12:19:47 EDT 1986</td>
</tr>
<tr>
<td>Sep Mon</td>
<td>%b %a</td>
<td>Mon Sep 1 12:19:47 EDT 1986</td>
</tr>
<tr>
<td>Jan Fri</td>
<td>%b %a</td>
<td>Fri Jan 2 12:19:47 EDT 1986</td>
</tr>
<tr>
<td>Dec Mon</td>
<td>%b %a</td>
<td>Mon Dec 1 12:19:47 EDT 1986</td>
</tr>
<tr>
<td>Jan Wed 1989</td>
<td>%b %a %Y</td>
<td>Wed Jan 4 12:19:47 EDT 1986</td>
</tr>
<tr>
<td>Fri 9</td>
<td>%a %H</td>
<td>Fri Sep 26 12:19:47 EDT 1986</td>
</tr>
<tr>
<td>Feb 10:30</td>
<td>%b %a %H: %S</td>
<td>Sun Feb 1 12:19:47 EDT 1986</td>
</tr>
<tr>
<td>10:30</td>
<td>%H: %M</td>
<td>Tue Sep 23 12:19:47 EDT 1986</td>
</tr>
<tr>
<td>13:30</td>
<td>%H: %M</td>
<td>Mon Sep 22 12:19:47 EDT 1986</td>
</tr>
</tbody>
</table>

Related Information
The ctime ("ctime, localtime, gmtime, mktime, difftime, asctime, or tzset Subroutine" on page 199), ctype ("ctype, isalpha, isupper, islower, isdigit, isxdigit, isalnum, isspace, ispunct, isprint, isgraph, iscntrl, or isascii Subroutines" on page 208), setlocale, strftime, and times ("getrusage, getrusage64, times, or vtimes Subroutine" on page 423) subroutines.

getdtablesiz subroutine

Purpose
Gets the descriptor table size.

Library
Standard C Library (libc.a)

Syntax
#include <unistd.h>
int getdtablesiz (void)

Description
The getdtablesiz subroutine is used to determine the size of the file descriptor table.
The size of the file descriptor table for a process is set by the `ulimit` command or by the `setrlimit` subroutine. The `getdtablesize` subroutine returns the current size of the table as reported by the `getrlimit` subroutine. If `getrlimit` reports that the table size is unlimited, `getdtablesize` instead returns the value of OPEN_MAX, which is the largest possible size of the table.

**Note:** The `getdtablesize` subroutine returns a runtime value that is specific to the version of the operating system on which the application is running. In AIX 4.3.1, `getdtablesize` returns a value that is set in the `limits` file, which can be different from system to system.

**Return Values**
The `getdtablesize` subroutine returns the size of the descriptor table.

**Related Information**
The `close` subroutine, `open`, `openx`, `open64`, `creat`, or `creat64` subroutine, `select` subroutine.

---

**getea Subroutine**

**Purpose**
Reads the value of an extended attribute.

**Syntax**
```
#include <sys/ea.h>

ssize_t getea(const char *path, const char *name, void *value, size_t size);
ssize_t fgetea(int filedes, const char *name, void *value, size_t size);
ssize_t lgetea(const char *path, const char *name, void *value, size_t size);
```

**Description**
Extended attributes are name:value pairs associated with the file system objects (such as files, directories, and symlinks). They are extensions to the normal attributes that are associated with all objects in the file system (that is, the `stat(2)` data).

Do not define an extended attribute name with the eight characters prefix "(0xF8)SYSTEM(0xF8)". Prefix "(0xF8)SYSTEM(0xF8)" is reserved for system use only.

**Note:** The 0xF8 prefix represents a non-printable character.

The `getea` subroutine retrieves the value of the extended attribute identified by `name` and associated with the given `path` in the file system. The length of the attribute `value` is returned. The `fgetea` subroutine is identical to `getea`, except that it takes a file descriptor instead of a path. The `lgetea` subroutine is identical to `getea`, except, in the case of a symbolic link, the link itself is interrogated rather than the file that it refers to.

**Parameters**
- `path` The path name of the file.
- `name` The name of the extended attribute. An extended attribute name is a NULL-terminated string.
- `value` A pointer to a buffer in which the attribute will be stored. The value of an extended attribute is an opaque byte stream of specified length.
The size of the buffer. If size is 0, getea returns the current size of the named extended attribute, which can be used to estimate whether the size of a buffer is sufficiently large enough to hold the value associated with the extended attribute.

A file descriptor for the file.

Return Values
If the getea subroutine succeeds, a nonnegative number is returned that indicates the size of the extended attribute value. Upon failure, -1 is returned and errno is set appropriately.

Error Codes

- **EACCES**
  Caller lacks read permission on the base file, or lacks the appropriate ACL privileges for named attribute read.
- **EFAULT**
  A bad address was passed for path, name, or value.
- **EFORMAT**
  File system is capable of supporting EAs, but EAs are disabled.
- **EINVAL**
  A path-like name should not be used (such as zml/file, ., and ..).
- **ENAMETOOLONG**
  The path or name value is too long.
- **ENOATTR**
  The named attribute does not exist, or the process has no access to this attribute.
- **ERANGE**
  The size of the value buffer is too small to hold the result.
- **ENOTSUP**
  Extended attributes are not supported by the file system.

The errors documented for the stat(2) system call are also applicable here.

**Related Information**

- listea Subroutine on page 718
- removea Subroutine
- setea Subroutine
- statea Subroutine

**getenv Subroutine**

**Purpose**

Returns the value of an environment variable.

**Library**

Standard C Library (libc.a)

**Syntax**

```c
#include <stdlib.h>

char *getenv (const char *Name);
```

**Description**

The getenv subroutine searches the environment list for a string of the form Name=Value. Environment variables are sometimes called shell variables because they are frequently set with shell commands.

**Parameters**

Name 
Specifies the name of an environment variable. If a string of the proper form is not present in the current environment, the getenv subroutine returns a null pointer.
Return Values
The `getenv` subroutine returns a pointer to the value in the current environment, if such a string is present. If such a string is not present, a null pointer is returned. The `getenv` subroutine normally does not modify the returned string. The `putenv` subroutine, however, may overwrite or change the returned string. Do not attempt to free the returned pointer. The `getenv` subroutine returns a pointer to the user's copy of the environment (which is static), until the first invocation of the `putenv` subroutine that adds a new environment variable. The `putenv` subroutine allocates an area of memory large enough to hold both the user's environment and the new variable. The next call to the `getenv` subroutine returns a pointer to this newly allocated space that is not static. Subsequent calls by the `putenv` subroutine use the `realloc` subroutine to make space for new variables. Unsuccessful completion returns a null pointer.

Related Information
The `putenv` subroutine.

getevars Subroutine

Purpose
Gets environment of a process.

Library
Standard C library (libc.a)

Syntax
```c
#include <procinfo.h>
#include <sys/types.h>

int getevars (processBuffer, bufferLen, argsBuffer, argsLen)
struct procsinfo *processBuffer
or struct procentry64 *processBuffer;
int bufferLen;
char *argsBuffer;
int argsLen;
```

Description
The `getevars` subroutine returns the environment that was passed to a command when it was started. Only one process can be examined per call to `getevars`.

The `getevars` subroutine uses the pi_pid field of `processBuffer` to determine which process to look for. `bufferLen` should be set to size of `struct procsinfo` or `struct procentry64`. Parameters are returned in `argsBuffer`, which should be allocated by the caller. The size of this array must be given in `argsLen`.

On return, `argsBuffer` consists of a succession of strings, each terminated with a null character (ascii '\0'). Hence, two consecutive NULLs indicate the end of the list.

Note: The arguments may be changed asynchronously by the process, but results are not guaranteed to be consistent.

Parameters

`processBuffer`
Specifies the address of a `procsinfo` or `procentry64` structure, whose pi_pid field should contain the pid of the process that is to be looked for.
bufferLen
Specifies the size of a single procsinfo or procentry64 structure.

argsBuffer
Specifies the address of an array of characters to be filled with a series of strings representing the parameters that are needed. An extra NULL character marks the end of the list. This array must be allocated by the caller.

argsLen
Specifies the size of the argsBuffer array. No more than argsLen characters are returned.

Return Values
If successful, the getevars subroutine returns zero. Otherwise, a value of -1 is returned and the errno global variable is set to indicate the error.

Error Codes
The getevars subroutine does not succeed if the following are true:

- ESRCH: The specified process does not exist.
-EFAULT: The copy operation to the buffer was not successful or the processBuffer or argsBuffer parameters are invalid.
-EINVAL: The bufferLen parameter does not contain the size of a single procsinfo or procentry64 structure.
-ENOMEM: There is no memory available in the address space.

Related Information
The getargs subroutine does not succeed if the following are true:

- getargs Subroutine on page 339.
- getpid, getpgrp, or getppid Subroutine on page 402.
- getpgrp, getppid Subroutine on page 402.
- getprocs or getthrds Subroutine on page 438.

The ps command.

getfilehdr Subroutine

Purpose
Retrieves the header details of the advanced accounting data file.

Library
The libaacct.a library.

Syntax
#define <sys/aacct.h>
getfilehdr(filename, hdrinfo)
char *filename;
struct aacct_file_header *hdrinfo;

Description
The getfilehdr subroutine retrieves the advanced accounting data file header information in a structure of type aacct_file_header and returns it to the caller through the structure pointer passed to it. The data file header contains the system details such as the name of the host, the partition number, and the system model.
Parameters

filename
Name of the advanced accounting data file.

hdrinfo
Pointer to the `aacct_file_header` structure in which the header information is returned.

Security
No restrictions. Any user can call this function.

Return Values

<table>
<thead>
<tr>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>The call to the subroutine was successful.</td>
</tr>
<tr>
<td>-1</td>
<td>The call to the subroutine failed.</td>
</tr>
</tbody>
</table>

Error Codes

<table>
<thead>
<tr>
<th>Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EINVAL</td>
<td>The passed pointer is NULL.</td>
</tr>
<tr>
<td>ENOENT</td>
<td>Specified data file does not exist.</td>
</tr>
<tr>
<td>EPERM</td>
<td>Permission denied. Unable to read the data file.</td>
</tr>
</tbody>
</table>

Related Information

The [“buildproclist Subroutine” on page 125](#), [“buildtranlist or freetranlist Subroutine” on page 126](#), [“getproclist, getparlist, or getarmlist Subroutine” on page 409](#), [Understanding the Advanced Accounting Subsystem](#)

getfirstprojdb Subroutine

Purpose
Retrieves details of the first project from the specified project database.

Library
The `libaacct.a` library.

Syntax

```c
getfirstprojdb(void *handle, struct project *project, char *comm)
```

Description
The `getfirstprojdb` subroutine retrieves the first project definitions from the project database, which is controlled through the `handle` parameter. The caller must initialize the project database prior to calling this routine with the `projdballoc` routine. Upon successful completion, the project information is copied to the project structure specified by the caller. In addition, the associated project comment, if present, is copied to the buffer pointed to by the `comm` parameter. The comment buffer is allocated by the caller and must have a length of 1024 bytes.

There is an internal state (that is, the current project) associated with the project database. When the project database is initialized, the current project is the first project in the database. The `getnextprojdb` subroutine returns the current project and advances the current project assignment to the next project in
the database so that successive calls read each project entry in the database. The **getfirstprojdb** subroutine can be used to reset the database, so that the initial project is the current project assignment.

**Parameters**

- **handle**: Pointer to the **projdb** handle.
- **project**: Pointer to project structure where the retrieved data is stored.
- **comm**: Pointer to the comment buffer.

**Security**

No restriction. Any user can call this function.

**Return Values**

- **0**: Success
- **-1**: Failure

**Error Codes**

- **EINVAL**: Invalid arguments, if passed pointer is NULL.
- **ENOENT**: No projects available.

**Related Information**

The **addprojdb Subroutine** on page 32, **chprojattrdb Subroutine** on page 159, **getnextprojdb Subroutine** on page 391, **getprojdb Subroutine** on page 414, **getprojs Subroutine** on page 415, **projdballoc Subroutine** on page 1158, **projdbfinit Subroutine** on page 1159, **projdbfree Subroutine** on page 1160, **rmprojdb Subroutine**.

**getfsent, getfsspec, getfsfile, getfstype, setfsent, or endfsent Subroutine**

**Purpose**

Gets information about a file system.

**Library**

Standard C Library (**libc.a**)

**Syntax**

```c
#include <fstab.h>

struct fstab *getfsent()

struct fstab *getfsspec (char *Special)

struct fstab *getfsfile (char *File)

struct fstab *getfstype (char *Type)
```
void setfsent( )
void endfsent( )

Description
The getfsent subroutine reads the next line of the /etc/filesystems file, opening the file if necessary.
The setfsent subroutine opens the /etc/filesystems file and positions to the first record.
The endfsent subroutine closes the /etc/filesystems file.

The getfsspec and getfsfile subroutines sequentially search from the beginning of the file until a matching special file name or file-system file name is found, or until the end of the file is encountered. The getfstype subroutine does likewise, matching on the file-system type field.

Note: All information is contained in a static area, which must be copied to be saved.

Parameters
Special Specifies the file-system file name.
File Specifies the file name.
Type Specifies the file-system type.

Return Values
The getfsent, getfsspec, getfstype, and getfsfile subroutines return a pointer to a structure that contains information about a file system. The header file fstab.h describes the structure. A null pointer is returned when the end of file (EOF) is reached or if an error occurs.

Files
/etc/filesystems Centralizes file system characteristics.

Related Information
The getvfsent, getvfsbytype, getvfsbyname, getvfsbyflag, setvfsent, or endvfsent subroutine.
The filesystems file.

getgid, getegid or gegidx Subroutine

Purpose
Gets the process group IDs.

Library
Standard C Library (libc.a)

Syntax
#include <unistd.h>
#include <sys/types.h>
Description
The `getgid` subroutine returns the real group ID of the calling process.

The `getegid` subroutine returns the effective group ID of the calling process.

The `getgidx` subroutine returns the group ID indicated by the `type` parameter of the calling process.

These subroutines are part of Base Operating System (BOS) Runtime.

Return Values
The `getgid`, `getegid` and `getgidx` subroutines return the requested group ID. The `getgid` and `getegid` subroutines are always successful.

The `getgidx` subroutine will return -1 and set the global `errno` variable to `EINVAL` if the `type` parameter is not one of `ID_REAL`, `ID_EFFECTIVE` or `ID_SAVED`.

Parameters

`type` Specifies the group ID to get. Must be one of `ID_REAL` (real group ID), `ID_EFFECTIVE` (effective group ID) or `ID_SAVED` (saved set-group ID).

Error Codes
If the `getgidx` subroutine fails the following is returned:

`EINVAL` Indicates the value of the type parameter is invalid.

Related Information
The “getgroups Subroutine” on page 378, `initgroups` subroutine, `setgid` subroutine, `setgroups` subroutine.

The `groups` command, `setgroups` command.

List of Security and Auditing Subroutines and Subroutines Overview in AIX 5L Version 5.3 General Programming Concepts: Writing and Debugging Programs.

getgrent, getgrgid, getgrnam, setgrent, or endgrent Subroutine

Purpose
Accesses the basic group information in the user database.

Library
Standard C Library (`libc.a`)
Syntax
#include <sys/types.h>
#include <grp.h>
struct group *getgrent ( );

struct group *getgrgid (GID)
gid_t GID;

struct group *getgrnam (Name)
const char * Name;

void setgrent ( );
void endgrent ( );

Description
Attention: The information returned by the getgrent, getgrnam, and getgrgid subroutines is stored in a static area and is overwritten on subsequent calls. You must copy this information to save it.

Attention: These subroutines should not be used with the getgroupattr subroutine. The results are unpredictable.

The setgrent subroutine opens the user database if it is not already open. Then, this subroutine sets the cursor to point to the first group entry in the database.

The getgrent, getgrnam, and getgrgid subroutines return information about the requested group. The getgrent subroutine returns the next group in the sequential search. The getgrnam subroutine returns the first group in the database whose name matches that of the Name parameter. The getgrgid subroutine returns the first group in the database whose group ID matches the GID parameter. The endgrent subroutine closes the user database.

Note: An ! (exclamation mark) is written into the gr_passwd field. This field is ignored and is present only for compatibility with older versions of UNIX.

These subroutines also return the list of user members for the group. Currently, the list is limited to 2000 entries (this could change in the future to where all the entries for the group are returned).

The Group Structure
The group structure is defined in the grp.h file and has the following fields:

- gr_name: Contains the name of the group.
- gr_passwd: Contains the password of the group. **Note:** This field is no longer used.
- gr_gid: Contains the ID of the group.
- gr_mem: Contains the members of the group.

If the Network Information Service (NIS) is enabled on the system, these subroutines attempt to retrieve the group information from the NIS authentication server.

Parameters

- **GID**: Specifies the group ID.
- **Name**: Specifies the group name.
- **Group**: Specifies the basic group information to enter into the user database.
Return Values
If successful, the getgrent, getgrnam, and getgrgid subroutines return a pointer to a valid group structure. Otherwise, a null pointer is returned.

Error Codes
These subroutines fail if one or more of the following are returned:

- **EIO** Indicates that an input/output (I/O) error has occurred.
- **EINTR** Indicates that a signal was caught during the getgrnam or getgrgid subroutine.
- **EMFILE** Indicates that the maximum number of file descriptors specified by the OPEN_MAX value are currently open in the calling process.
- **ENFILE** Indicates that the maximum allowable number of files is currently open in the system.

To check an application for error situations, set the errno global variable to a value of 0 before calling the getgrgid subroutine. If the errno global variable is set on return, an error occurred.

File
/etc/group Contains basic group attributes.

Related Information
"putgrent Subroutine" on page 1305
List of Security and Auditing Subroutines, Subroutines Overview in AIX 5L Version 5.3 General Programming Concepts: Writing and Debugging Programs.

getgrgid_r Subroutine

**Purpose**
Gets a group database entry for a group ID.

**Library**
Thread-Safe C Library (libc_r.a)

**Syntax**
```c
#include <sys/types.h>
#include <grp.h>

int getgrgid_r(gid_t gid,  
struct group *grp,  
char *buffer,  
size_t bufsize,  
struct group **result);
```

**Description**
The getgrgid_r subroutine updates the group structure pointed to by grp and stores a pointer to that structure at the location pointed to by result. The structure contains an entry from the group database with a matching gid. Storage referenced by the group structure is allocated from the memory provided with the buffer parameter, which is bufsize characters in size. The maximum size needed for this buffer can be determined with the (_SC_GETGR_R_SIZE_MAX) sysconf parameter. A NULL pointer is returned at the location pointed to by result on error or if the requested entry is not found.
Return Values
Upon successful completion, `getgrgid_r` returns a pointer to a `struct group` with the structure defined in `<grp.h>` with a matching entry if one is found. The `getgrgid_r` function returns a null pointer if either the requested entry was not found, or an error occurred. On error, `errno` will be set to indicate the error.

The return value points to a static area that is overwritten by a subsequent call to the `getgrent, getgrgid, or getgrnam` subroutine.

If successful, the `getgrgid_r` function returns zero. Otherwise, an error number is returned to indicate the error.

Error Codes
The `getgrgid_r` function fails if:

ERANGE Insufficient storage was supplied via `buffer` and `bufsize` to contain the data to be referenced by the resulting `group` structure.

Applications wishing to check for error situations should set `errno` to 0 before calling `getgrgid_r`. If `errno` is set on return, an error occurred.

Related Information
The `getgrent, getgrgid, getgrnam, setgrent, endgrent` subroutine.

The `<grp.h>`, `<limits.h>`, and `<sys/types.h>` header files.

---

getgrnam_r Subroutine

Purpose
Search a group database for a name.

Library
Thread-Safe C Library (`libc_r.a`)

Syntax
```c
#include <sys/types.h>
#include <grp.h>

int getgrnam_r (const char **name,
    struct group *grp,
    char *buffer,
    size_t bufsize,
    struct group **result);
```

Description
The `getgrnam_r` function updates the `group` structure pointed to by `grp` and stores pointer to that structure at the location pointed to by `result`. The structure contains an entry from the group database with a matching `gid` or `name`. Storage referenced by the group structure is allocated from the memory provided with the `buffer` parameter, which is `bufsize` characters in size. The maximum size needed for this buffer can be determined with the `_SC_GETGR_R_SIZE_MAX` `sysconf` parameter. A NULL pointer is returned at the location pointed to by `result` on error or if the requested entry is not found.
Return Values
The `getgrnam_r` function returns a pointer to a `struct group` with the structure defined in `<grp.h>` with a matching entry if one is found. The `getgrnam_r` function returns a null pointer if either the requested entry was not found, or an error occurred. On error, `errno` will be set to indicate the error.

The return value points to a static area that is overwritten by a subsequent call to the `getgrent`, `getgrgid`, or `getgrnam` subroutine.

If successful, the `getgrnam_r` function returns zero. Otherwise, an error number is returned to indicate the error.

Error Codes
The `getgrnam_r` function fails if:

- `ERANGE` Insufficient storage was supplied via `buffer` and `bufsize` to contain the data to be referenced by the resulting `group` structure.

Applications wishing to check for error situations should set `errno` to 0 before calling `getgrnam_r`. If `errno` is set on return, an error occurred.

Related Information
The `getgrent, getgrgid, getgrnam, setgrent, endgrent` subroutine.

The `<grp.h>`, `<limits.h>`, and `<sys/types.h>` header files.

---

textgroupattr, IDtogroup, nextgroup, or putgroupattr Subroutine

Purpose
Accesses the group information in the user database.

Library
Security Library (`libc.a`)

Syntax
```c
#include <usersec.h>

int getgroupattr (char *Group, char *Attribute, void *Value, int Type)
char *Group;
char *Attribute;
void *Value;
int Type;

int putgroupattr (char *Group, char *Attribute, void *Value, int Type)
char *Group;
char *Attribute;
void *Value;
int Type;

char *IDtogroup (gid_t GID)
```

370 Technical Reference, Volume 1: Base Operating System and Extensions
char *nextgroup (Mode, Argument);
int Mode, Argument;

Description
Attention: These subroutines and the setpwent and setgrent subroutines should not be used simultaneously. The results can be unpredictable.

These subroutines access group information. Because of their greater granularity and extensibility, you should use them instead of the getgrent, putgrent, getgrnam, getgrgid, setgrent, and endgrent subroutines.

The getgroupattr subroutine reads a specified attribute from the group database. If the database is not already open, the subroutine will do an implicit open for reading.

Similarly, the putgroupattr subroutine writes a specified attribute into the group database. If the database is not already open, the subroutine does an implicit open for reading and writing. Data changed by putgroupattr must be explicitly committed by calling the putgroupattr subroutine with a Type parameter specifying the SEC_COMMIT value. Until the data is committed, only get subroutine calls within the process will return the written data.

New entries in the user and group databases must first be created by invoking putgroupattr with the SEC_NEW type.

The IDtogroup subroutine translates a group ID into a group name.

The nextgroup subroutine returns the next group in a linear search of the group database. The consistency of consecutive searches depends upon the underlying storage-access mechanism and is not guaranteed by this subroutine.

The setuserdb and enduserdb subroutines should be used to open and close the user database.

Parameters
Argument Presently unused and must be specified as null.
Attribute Specifies which attribute is read. The following possible values are defined in the usersec.h file:
S_ID Group ID. The attribute type is SEC_INT.
S_USERS Members of the group. The attribute type is SEC_LIST.
S_ADMS Administrators of the group. The attribute type is SEC_LIST.
S_ADMIN Administrative status of a group. Type: SEC_BOOL.
S_GRPEXPORT Specifies if the DCE registry can overwrite the local group information with the DCE group information during a DCE export operation. The attribute type is SEC_BOOL.

Additional user-defined attributes may be used and will be stored in the format specified by the Type parameter.

GID Specifies the group ID to be translated into a group name.
Group Specifies the name of the group for which an attribute is to be read.
**Mode**

Specifies the search mode. Also can be used to delimit the search to one or more user credential databases. Specifying a non-null *Mode* value implicitly rewinds the search. A null mode continues the search sequentially through the database. This parameter specifies one of the following values as a bit mask (defined in the *usersec.h* file):

- **S_LOCAL**
  The local database of groups are included in the search.

- **S_SYSTEM**
  All credentials servers for the system are searched.

**Type**

Specifies the type of attribute expected. Valid values are defined in the *usersec.h* file and include:

- **SEC_INT**
  The format of the attribute is an integer. The buffer returned by the *getgroupattr* subroutine and the buffer supplied by the *putgroupattr* subroutine are defined to contain an integer.

- **SEC_CHAR**
  The format of the attribute is a null-terminated character string.

- **SEC_LIST**
  The format of the attribute is a series of concatenated strings, each null-terminated. The last string in the series is terminated by two successive null characters.

- **SEC_BOOL**
  A pointer to an integer (*int *) that has been cast to a null pointer.

- **SEC_COMMIT**
  For the *putgroupattr* subroutine, this value specified by itself indicates that changes to the named group are committed to permanent storage. The *Attribute* and *Value* parameters are ignored. If no group is specified, changes to all modified groups are committed to permanent storage.

- **SEC_DELETE**
  The corresponding attribute is deleted from the database.

- **SEC_NEW**
  If using the *putgroupattr* subroutine, updates all the group database files with the new group name.

**Value**

Specifies the address of a pointer for the *getgroupattr* subroutine. The *getgroupattr* subroutine will return the address of a buffer in the pointer. For the *putgroupattr* subroutine, the *Value* parameter specifies the address of a buffer in which the attribute is stored. See the *Type* parameter for more details.

**Security**

Files Accessed:

<table>
<thead>
<tr>
<th>Mode</th>
<th>File</th>
</tr>
</thead>
<tbody>
<tr>
<td>rw</td>
<td><em>/etc/group</em> (write access for <em>putgroupattr</em>)</td>
</tr>
<tr>
<td>rw</td>
<td><em>/etc/security/group</em> (write access for <em>putgroupattr</em>)</td>
</tr>
</tbody>
</table>

**Return Values**

The *getgroupattr* and *putgroupattr* subroutines, when successfully completed, return a value of 0. Otherwise, a value of -1 is returned and the *errno* global variable is set to indicate the error.

The *IDtogroup* and *nextgroup* subroutines return a character pointer to a buffer containing the requested group name, if successfully completed. Otherwise, a null pointer is returned and the *errno* global variable is set to indicate the error.
Error Codes

Note: All of these subroutines return errors from other subroutines.

These subroutines fail if the following is true:

EACCESS Access permission is denied for the data request.

The getgroupattr and putgroupattr subroutines fail if one or more of the following are true:

EINVAL The Value parameter does not point to a valid buffer or to valid data for this type of attribute. Limited testing is possible and all errors may not be detected.
EINVAL The Type parameter contains more than one of the SEC_INT, SEC_BOOL, SEC_CHAR, SEC_LIST, or SEC_COMMIT attributes.
EINVAL The Type parameter specifies that an individual attribute is to be committed, and the Group parameter is null.
ENOENT The specified Group parameter does not exist or the attribute is not defined for this group.
EPERM Operation is not permitted.

The IDtogroup subroutine fails if the following is true:

ENOENT The GID parameter could not be translated into a valid group name on the system.

The nextgroup subroutine fails if one or more of the following are true:

EINVAL The Mode parameter is not null, and does not specify either S_LOCAL or S_SYSTEM.
EINVAL The Argument parameter is not null.
ENOENT The end of the search was reached.

Related Information
The getuserattr, getuserpw, putuserattr, putuserpw, or putuserpwhist subroutine, getresuserattr, putresuserattr, or putresuserpwhist subroutine, setpwdb subroutine, setuserdb subroutine.

List of Security and Auditing Subroutines and Subroutines Overview in AIX 5L Version 5.3 General Programming Concepts: Writing and Debugging Programs.

getgroupattrs Subroutine

Purpose
Retrieves multiple group attributes in the group database.

Library
Security Library (libc.a)

Syntax
#include <usersec.h>
int getgroupattrs (Group, Attributes, Count)
char * Group;
dbattr_t * Attributes;
int Count

Description
Attention: Do not use this subroutine and the setpwent and setgrent subroutines simultaneously. The results can be unpredictable.

The getgroupattrs subroutine accesses group information. Because of its greater granularity and extensibility, use it instead of the getgrent routines.

The getgroupattrs subroutine reads one or more attributes from the group database. If the database is not already open, this subroutine does an implicit open for reading. A call to the getgroupattrs subroutine with an Attributes parameter of null and Count parameter of 0 for every new group verifies that the group exists.

The Attributes array contains information about each attribute that is to be read. The dbattr_t data structure contains the following fields:

- attr_name
  The name of the desired attribute.

- attr_idx
  Used internally by the getgroupattrs subroutine.

- attr_type
  The type of the desired attribute. The list of attribute types is defined in the usersec.h header file.

- attr_flag
  The results of the request to read the desired attribute.

- attr_un
  A union containing the returned values. Its union members that follow correspond to the definitions of the attr_char, attr_int, attr_long, and attr_llong macros, respectively:

  - un_char
    Attributes of type SEC_CHAR and SEC_LIST store a pointer to the returned attribute in this member when the requested attribute is successfully read. The caller is responsible for freeing this memory.

  - un_int
    Attributes of type SEC_INT and SEC_BOOL store the value of the attribute in this member when the requested attribute is successfully read.

  - un_long
    Attributes of type SEC_LONG store the value of the attribute in this member when the requested attribute is successfully read.

  - un_llong
    Attributes of type SEC_LLONG store the value of the attribute in this member when the requested attribute is successfully read.

- attr_domain
  The authentication domain containing the attribute. The getgroupattrs subroutine is responsible for managing the memory referenced by this pointer.

  If attr_domain is specified for an attribute, the get request is sent only to that domain.

  If attr_domain is not specified (that is, set to NULL), getgroupattrs searches the domains in a predetermined order. The search starts with the local file system and continues with the domains specified in the /usr/lib/security/methods.cfg file. This search space can be restricted with the setauthdb subroutine so that only the domain specified in the setauthdb call is searched.
If `attr_domain` is not specified, the `getgroupattrs` subroutine sets this field to the name of the domain from which the value is retrieved. If the request for a NULL domain was not satisfied, the request is tried from the local files using the default stanza.

Use the `setuserdb` and `enduserdb` subroutines to open and close the group database. Failure to explicitly open and close the group database can result in loss of memory and performance.

### Parameters

- **Group**
  - Specifies the name of the group for which the attributes are to be read.

- **Attributes**
  - A pointer to an array of 0 or more elements of type `dbattr_t`. The list of group attributes is defined in the `usersec.h` header file.

- **Count**
  - The number of array elements in `Attributes`. A `Count` parameter of 0 can be used to determine if the group exists.

### Security

Files accessed:

<table>
<thead>
<tr>
<th>Mode</th>
<th>File</th>
</tr>
</thead>
<tbody>
<tr>
<td>rw</td>
<td><code>/etc/group</code></td>
</tr>
<tr>
<td>rw</td>
<td><code>/etc/security/group</code></td>
</tr>
</tbody>
</table>

### Return Values

If `Group` exists, the `getgroupattrs` subroutine returns 0. Otherwise, a value of -1 is returned and the `errno` global variable is set to indicate the error. Each element in the `Attributes` array must be examined on a successful call to `getgroupattrs` to determine if the `Attributes` array entry was successfully retrieved.

### Error Codes

The `getgroupattrs` subroutine returns an error that indicates that the group does or does not exist. Additional errors can indicate an error querying the information databases for the requested attributes.

- **EINVAL**
  - The `Count` parameter is less than zero.

- **EINVAL**
  - The `Attributes` parameter is null and the `Count` parameter is greater than 0.

- **ENOENT**
  - The specified `Group` parameter does not exist.

If the `getgroupattrs` subroutine fails to query an attribute, one or more of the following errors is returned in the `attr_flag` field of the corresponding `Attributes` element:

- **EACCESS**
  - The user does not have access to the attribute specified in the `attr_name` field.

- **EINVAL**
  - The `attr_type` field in the `Attributes` entry contains an invalid type.

- **EINVAL**
  - The `attr_un` field in the `Attributes` entry does not point to a valid buffer or to valid data for this type of attribute. Limited testing is possible and all errors might not be detected.

- **ENOATTR**
  - The `attr_name` field in the `Attributes` entry specifies an attribute that is not defined for this user or group.

- **ENOMEM**
  - Memory could not be allocated to store the return value.

### Examples

The following sample test program displays the output to a call to `getgroupattrs`. In this example, the system has a user named `foo`.

```c
attribute name : id
attribute flag : 0
attribute domain : files
```
attribute value : 204
attribute name : admin
attribute flag : 0
attribute domain : files
attribute value : 0

attribute name : adms
attribute flag : 0
attribute domain : files
attribute value : 0

attribute name : registry
attribute flag : 0
attribute domain :
attribute value : compat

#include <stdio.h>
#include <usersec.h>

#define NATTR 4
#define GROUPNAME "bar"

char * ConvertToComma(char *); /* Convert from SEC_LIST to SEC_CHAR with '\0' replaced with ',' */

main() {

dbattr_t attributes[NATTR];
int i;
int rc;

memset(&attributes, 0, sizeof(attributes));

/*
* Fill in the attributes array with "id", "expires" and
* "SYSTEM" attributes.
*/

attributes[0].attr_name = $ID;
attributes[0].attr_type = SEC_INT;;

attributes[1].attr_name = $ADMIN;
attributes[1].attr_type = SEC_BOOL;

attributes[2].attr_name = $ADMS;
attributes[2].attr_type = SEC_LIST;

attributes[3].attr_name = $REGISTRY;
attributes[3].attr_type = SEC_CHAR;

/*
* Make a call to getuserattrs.
*/

setuserdb($READ);

rc = getgroupattrs(GROUPNAME, attributes, NATTR);

enduserdb();

if (rc) {
printf("getgroupattrs failed ....\n");
exit(-1);
}

for (i = 0; i < NATTR; i++) {
printf("attribute name : %s \n", attributes[i].attr_name);
printf("attribute flag : %d \n", attributes[i].attr_flag);

if (attributes[i].attr_flag) {
    /*
     * No attribute value. Continue.
     */
    printf("\n");
    continue;
}
    /*
     * We have a value.
     */
    printf("attribute domain : %s \n", attributes[i].attr_domain);
    printf("attribute value : ");

switch (attributes[i].attr_type) {
    case SEC_CHAR:
        if (attributes[i].attr_char) {
            printf("%s \n", attributes[i].attr_char);
            free(attributes[i].attr_char);
        }
        break;
    case SEC_LIST:
        if (attributes[i].attr_char) {
            printf("%s \n", ConvertToComma(attributes[i].attr_char));
            free(attributes[i].attr_char);
        }
        break;
    case SEC_INT:
    case SEC_BOOL:
        printf("%d \n", attributes[i].attr_int);
        break;
    default:
        break;
}
printf("\n");
}
exit(0);

The following output for the call is expected:

Base Operating System (BOS) Runtime Services (A-P) 377

printf("attribute flag : %d \n", attributes[i].attr_flag);

if (attributes[i].attr_flag) {
    /*
     * No attribute value. Continue.
     */
    printf("\n");
    continue;
}
    /*
     * We have a value.
     */
    printf("attribute domain : %s \n", attributes[i].attr_domain);
    printf("attribute value : ");

switch (attributes[i].attr_type) {
    case SEC_CHAR:
        if (attributes[i].attr_char) {
            printf("%s \n", attributes[i].attr_char);
            free(attributes[i].attr_char);
        }
        break;
    case SEC_LIST:
        if (attributes[i].attr_char) {
            printf("%s \n", ConvertToComma(attributes[i].attr_char));
            free(attributes[i].attr_char);
        }
        break;
    case SEC_INT:
    case SEC_BOOL:
        printf("%d \n", attributes[i].attr_int);
        break;
    default:
        break;
}
printf("\n");
}
exit(0);

The following output for the call is expected:
attribute name : id
attribute flag : 0
attribute domain : files
attribute value : 204

attribute name : admin
attribute flag : 0
attribute domain : files
attribute value : 0

attribute name : adms
attribute flag : 0
attribute domain : files
attribute value :

attribute name : registry
attribute flag : 0
attribute domain :
attribute value : compat

Files

/etc/group
Contains group IDs.

Related Information
The "getuserattrs Subroutine" on page 455, setuserdb Subroutine
List of Security and Auditing Subroutines, Subroutines Overview in AIX 5L Version 5.3 General Programming Concepts: Writing and Debugging Programs.

getgroups Subroutine

Purpose
Gets the supplementary group ID of the current process.

Library
Standard C Library (libc.a)

Syntax
#include <sys/types.h>
#include <unistd.h>

int getgroups (NGroups, GIDSet)
int NGroups;
gid_t GIDSet[];

Description
The getgroups subroutine gets the supplementary group ID of the process. The list is stored in the array pointed to by the GIDSet parameter. The NGroups parameter indicates the number of entries that can be stored in this array. The getgroups subroutine never returns more than the number of entries specified by the NGROUPS_MAX constant. (The NGROUPS_MAX constant is defined in the limits.h file.) If the value in the NGroups parameter is 0, the getgroups subroutine returns the number of groups in the supplementary group.
Parameters

- **GIDSet**
  - Points to the array in which the supplementary group ID of the user's process is stored.

- **NGroups**
  - Indicates the number of entries that can be stored in the array pointed to by the **GIDSet** parameter.

Return Values

Upon successful completion, the **getgroups** subroutine returns the number of elements stored into the array pointed to by the **GIDSet** parameter. If the **getgroups** subroutine is unsuccessful, a value of -1 is returned and the **errno** global variable is set to indicate the error.

Error Codes

The **getgroups** subroutine is unsuccessful if either of the following error codes is true:

- **EFAULT**
  - The **NGroups** and **GIDSet** parameters specify an array that is partially or completely outside of the allocated address space of the process.

- **EINVAL**
  - The **NGroups** parameter is smaller than the number of groups in the supplementary group.

Related Information

The **getgid** subroutine, **initgroups** subroutine, **setgid** subroutine, **setgroups** subroutine.

The **groups** command, **setgroups** command.

List of Security and Auditing Subroutines and Subroutines Overview in AIX 5L Version 5.3 General Programming Concepts: Writing and Debugging Programs.

---

**getgrpaclattr, nextgrpacl, or putgrpaclattr Subroutine**

**Purpose**

Accesses the group screen information in the SMIT ACL database.

**Library**

Security Library (**libc.a**)

**Syntax**

```c
#include <usersec.h>

int getgrpaclattr (Group, Attribute, Value, Type)
char *User;
char *Attribute;
void *Value;
int Type;
char *nextgrpacl(void)

int putgrpaclattr (Group, Attribute, Value, Type)
char *User;
char *Attribute;
void *Value;
int Type;
```
### Description

The `getgrpaclattr` subroutine reads a specified group attribute from the SMIT ACL database. If the database is not already open, this subroutine does an implicit open for reading.

Similarly, the `putgrpaclattr` subroutine writes a specified attribute into the user SMIT ACL database. If the database is not already open, this subroutine does an implicit open for reading and writing. Data changed by the `putgrpaclattr` subroutine must be explicitly committed by calling the `putgrpaclattr` subroutine with a `Type` parameter specifying `SEC_COMMIT`. Until all the data is committed, only the `getgrpaclattr` subroutine within the process returns written data.

The `nextgrpact` subroutine returns the next group in a linear search of the group SMIT ACL database. The consistency of consecutive searches depends upon the underlying storage-access mechanism and is not guaranteed by this subroutine.

The `setacldb` and `endacldb` subroutines should be used to open and close the database.

### Parameters

- **Attribute**
  Specifies which attribute is read. The following possible attributes are defined in the `usersec.h` file:
  - `S_SCREENS`
    String of SMIT screens. The attribute type is `SEC_LIST`.

- **Type**
  Specifies the type of attribute expected. Valid types are defined in the `usersec.h` file and include:
  - `SEC_LIST`
    The format of the attribute is a series of concatenated strings, each null-terminated. The last string in the series must be an empty (zero character count) string.
    For the `getgrpaclattr` subroutine, the user should supply a pointer to a defined character pointer variable. For the `putgrpaclattr` subroutine, the user should supply a character pointer.
  - `SEC_COMMIT`
    For the `putgrpaclattr` subroutine, this value specified by itself indicates that changes to the named group are to be committed to permanent storage. The `Attribute` and `Value` parameters are ignored. If no group is specified, the changes to all modified groups are committed to permanent storage.
  - `SEC_DELETE`
    The corresponding attribute is deleted from the group SMIT ACL database.
  - `SEC_NEW`
    Updates the group SMIT ACL database file with the new group name when using the `putgrpaclattr` subroutine.

- **Value**
  Specifies a buffer, a pointer to a buffer, or a pointer to a pointer depending on the `Attribute` and `Type` parameters. See the `Type` parameter for more details.

### Return Values

If successful, the `getgrpaclattr` returns 0. Otherwise, a value of -1 is returned and the `errno` global variable is set to indicate the error.

### Error Codes

Possible return codes are:

- **EACCES**
  Access permission is denied for the data request.
- **ENOENT**
  The specified `Group` parameter does not exist or the attribute is not defined for this group.
- **ENOATTR**
  The specified user attribute does not exist for this group.
- **EINVAL**
  The `Attribute` parameter does not contain one of the defined attributes or null.
EINVAL
The Value parameter does not point to a valid buffer or to valid data for this type of attribute.

EPERM
Operation is not permitted.

Related Information
The getgrpaclattr, nextgrpacl, or putgrpaclattr subroutine, setacldb, or endacldb subroutine.

getgrset Subroutine

Purpose
Accesses the concurrent group set information in the user database.

Library
Standard C Library (libc.a)

Syntax
char *getgrset (User)
const char *User;

Description
The getgrset subroutine returns a pointer to the comma separated list of concurrent group identifiers for the named user.

If the Network Information Service (NIS) is enabled on the system, these subroutines attempt to retrieve the user information from the NIS authentication server.

Parameters

User Specifies the user name.

Return Values
If successful, the getgrset subroutine returns a pointer to a list of supplementary groups. This pointer must be freed by the user.

Error Codes
A NULL pointer is returned on error. The value of the errno global variable is undefined on error.

File
/etc/group Contains basic group attributes.

Related Information
List of Security and Auditing Subroutines, Subroutines Overview in AIX 5L Version 5.3 General Programming Concepts: Writing and Debugging Programs.
getinterval, incinterval, absinterval, resinc, resabs, alarm, ualarm, getitimer or setitimer Subroutine

Purpose
Manipulates the expiration time of interval timers.

Library
Standard C Library (libc.a)

Syntax
#include <sys/time.h>

int getinterval (TimerID, Value)
timer_t TimerID;
struct itimerstruc_t *Value;

int incinterval (TimerID, Value, OValue)
timer_t TimerID;
struct itimerstruc_t *Value, *OValue;

int absinterval (TimerID, Value, OValue)
timer_t TimerID;
struct itimerstruc_t *Value, *OValue;

int resabs (TimerID, Resolution, Maximum)
timer_t TimerID;
struct timestruc_t *Resolution, *Maximum;

int resinc (TimerID, Resolution, Maximum)
timer_t TimerID;
struct timestruc_t *Resolution, *Maximum;

#include <unistd.h>

unsigned int alarm (Seconds)
unsigned int Seconds;

useconds_t ualarm (Value, Interval)
useconds_t Value, Interval;

int setitimer (Which, Value, OValue)
int Which;
struct itimerval *Value, *OValue;

int getitimer (Which, Value)
int Which;
struct itimerval *Value;

Description
The getinterval, incinterval, and absinterval subroutines manipulate the expiration time of interval timers. These functions use a timer value defined by the struct itimerstruc_t structure, which includes the following fields:

struct timestruc_t it_interval; /* timer interval period */
struct timestruc_t it_value; /* timer interval expiration */
If the \textit{it\_value} field is nonzero, it indicates the time to the next timer expiration. If \textit{it\_value} is 0, the
per-process timer is disabled. If the \textit{it\_interval} member is nonzero, it specifies a value to be used in
reloading the \textit{it\_value} field when the timer expires. If \textit{it\_interval} is 0, the timer is to be disabled after its
next expiration (assuming \textit{it\_value} is nonzero).

The \texttt{getinterval} subroutine returns a value from the \texttt{struct itimerstruc\_t} structure to the \textit{Value} parameter.
The \textit{it\_value} field of this structure represents the amount of time in the current interval before the timer expires, should one exist for the per-process timer specified in the \textit{TimerID} parameter. The \textit{it\_interval} field has the value last set by the \texttt{incinterval} or \texttt{absinterval} subroutine. The fields of the \textit{Value} parameter
are subject to the resolution of the timer.

The \texttt{incinterval} subroutine sets the value of a per-process timer to a given offset from the current timer setting. The \texttt{absinterval} subroutine sets the value of the per-process timer to a given absolute value. If the specified absolute time has already expired, the \texttt{absinterval} subroutine will succeed and the expiration notification will be made. Both subroutines update the interval timer period. Time values smaller than the resolution of the specified timer are rounded up to this resolution. Time values larger than the maximum value of the specified timer are rounded down to the maximum value.

The \texttt{resinc} and \texttt{resabs} subroutines return the resolution and maximum value of the interval timer
contained in the \textit{TimerID} parameter. The resolution of the interval timer is contained in the \textit{Resolution}
parameter, and the maximum value is contained in the \textit{Maximum} parameter. These values might not be
the same as the values returned by the corresponding system timer, the \texttt{gettimer} subroutine. In addition, it
is likely that the maximum values returned by the \texttt{resinc} and \texttt{resabs} subroutines will be different.

\textbf{Note:} If a nonprivileged user attempts to submit a fine granularity timer (that is, a timer request of less
than 10 milliseconds), the timer request is raised to 10 milliseconds.

The \texttt{alarm} subroutine causes the system to send the calling thread’s process a \texttt{SIGALRM} signal after the
number of real-time seconds specified by the \textit{Seconds} parameter have elapsed. Since the signal is sent to
the process, in a multi-threaded process another thread than the one that called the \texttt{alarm} subroutine may
receive the \texttt{SIGALRM} signal. Processor scheduling delays may prevent the process from handling the
signal as soon as it is generated. If the value of the \textit{Seconds} parameter is 0, a pending alarm request, if any, is canceled. Alarm requests are not stacked. Only one \texttt{SIGALRM} generation can be scheduled in this
manner. If the \texttt{SIGALRM} signal has not yet been generated, the call results in rescheduling the time at
which the \texttt{SIGALRM} signal is generated. If several threads in a process call the \texttt{alarm} subroutine, only the
last call will be effective.

The \texttt{ualarm} subroutine sends a \texttt{SIGALRM} signal to the invoking process in a specified number of
seconds. The \texttt{getitimer} subroutine gets the value of an interval timer. The \texttt{setitimer} subroutine sets the
value of an interval timer.

\textbf{Parameters}

\begin{itemize}
  \item \texttt{TimerID} Specifies the ID of the interval timer.
  \item \texttt{Value} Points to a \texttt{struct itimerstruc\_t} structure.
  \item \texttt{OValue} Represents the previous time-out period.
  \item \texttt{Resolution} Indicates the maximum value of the interval timer.
  \item \texttt{Maximum} Indicates the maximum value of the interval timer.
  \item \texttt{Seconds} Specifies the number of real-time seconds to elapse before the first \texttt{SIGALRM} signal.
  \item \texttt{Interval} Specifies the number of microseconds between subsequent periodic \texttt{SIGALRM} signals. If a
    nonprivileged user attempts to submit a fine granularity timer (that is, a timer request of
    less than 10 milliseconds), the timer request interval is automatically raised to 10
    milliseconds.
\end{itemize}
Which Identifies the type of timer. Valid values are:

**ITIMER_REAL**
Decrement in real time. A **SIGALRM** signal occurs when this timer expires.

**ITIMER_VIRTUAL**
Decrement in process virtual time. It runs only during process execution. A **SIGVTALRM** signal occurs when it expires.

**ITIMER_PROF**
Decrement in process virtual time and when the system runs on behalf of the process. It is designed for use by interpreters in statistically profiling the execution of interpreted programs. Each time the **ITIMER_PROF** timer expires, the **SIGPROF** signal occurs. Because this signal may interrupt in-progress system calls, programs using this timer must be prepared to restart interrupted system calls.

**Return Values**
If these subroutines are successful, a value of 0 is returned. If an error occurs, a value of -1 is returned and the **errno** global variable is set.

The **alarm** subroutine returns the amount of time (in seconds) remaining before the system is scheduled to generate the **SIGALRM** signal from the previous call to **alarm**. It returns a 0 if there was no previous **alarm** request.

The **ualarm** subroutine returns the number of microseconds previously remaining in the alarm clock.

**Error Codes**
If the **getinterval**, **incinterval**, **absinterval**, **resinc**, **resabs**, **settimer**, **gettimer**, or **settimer** subroutine is unsuccessful, a value of -1 is returned and the **errno** global variable is set to one of the following error codes:

- **EINVAL** Indicates that the TimerID parameter does not correspond to an ID returned by the **gettimerid** subroutine, or a value structure specified a nanosecond value less than 0 or greater than or equal to one thousand million (1,000,000,000).
- **EIO** Indicates that an error occurred while accessing the timer device.
- **EFAULT** Indicates that a parameter address has referenced invalid memory.

The **alarm** subroutine is always successful. No return value is reserved to indicate an error for it.

**Related Information**
The **gettimer** (*gettimer, settimer, restimer, stime, or time Subroutine* on page 441) subroutine, **gettimerid** (*gettimerid Subroutine* on page 444) subroutine, **sigaction**, **sigvec**, or **signal** subroutine.

*Time data manipulation services* in Operating system and device management.

[Subroutines Overview](#) in AIX 5L Version 5.3 General Programming Concepts: Writing and Debugging Programs.

[Signal Management](#) in AIX 5L Version 5.3 General Programming Concepts: Writing and Debugging Programs provides more information about signal management in multi-threaded processes.
getiopri Subroutine

Purpose
Enables the getting of a process I/O priority.

Syntax
short getiopri (ProcessID);
pid_t ProcessID;

Description
The getiopri subroutine returns the I/O scheduling priority of a process. If the target process ID does not match the process ID of the caller, the caller must either be running as root, or have an effective and real user ID that matches the target process.

Parameters
ProcessID Specifies the process ID. If this value is -1, the current process scheduling priority is returned.

Return Values
Upon successful completion, the getiopri subroutine returns the I/O scheduling priority of a thread in the process. A returned value of IOPRIORITY_UNSET indicates that the I/O priority was not set. Otherwise, a value of -1 is returned and the errno global variable is set to indicate the error.

Errors
EPERM The calling process is not root. It does not have the same process ID as the target process, and does not have the same real effective user ID as the target process.
ESRCH No process can be found corresponding to the specified ProcessID.

Implementation Specifics
1. Implementation requires an additional field in the proc structure.
2. The default setting for process I/O priority is IOPRIORITY_UNSET.
3. Once set, process I/O priorities should be inherited across a fork. I/O priorities should not be inherited across an exec.
4. The setiopri system call generates an auditing event using audit_svcstart if auditing is enabled on the system (audit_flag is true).

Related Information
The "getpri Subroutine" on page 406, setpri subroutine, setpri subroutine

getipnodebyaddr Subroutine

Purpose
Address-to-nodename translation.
Library
Standard C Library (libc.a)
(libaixinet)

Syntax

#include <sys/socket.h>
#include <netdb.h>
struct hostent *getipnodebyaddr [src, len, af, error_num];
const void *src;
size_t len;
int af;
int *error_num;

Description

The getipnodebyaddr subroutine has the same arguments as the gethostbyaddr subroutine but adds an error number. It is thread-safe.

The getipnodebyaddr subroutine is similar in its name query to the gethostbyaddr subroutine except in one case. If af equals AF_INET6 and the IPv6 address is an IPv4-mapped IPv6 address or an IPv4-compatible address, then the first 12 bytes are skipped over and the last 4 bytes are used as an IPv4 address with af equal to AF_INET to lookup the name.

If the getipnodebyaddr subroutine is returning success, then the single address that is returned in the hostent structure is a copy of the first argument to the function with the same address family and length that was passed as arguments to this function.

All of the information returned by getipnodebyaddr is dynamically allocated: the hostent structure and the data areas pointed to by the h_name, h_addr_list, and h_aliases members of the hostent structure. To return this information to the system the function freehostent is called.

Parameters

<table>
<thead>
<tr>
<th>src</th>
<th>Specifies a node address. It is a pointer to either a 4-byte (IPv4) or 16-byte (IPv6) binary format address.</th>
</tr>
</thead>
<tbody>
<tr>
<td>af</td>
<td>Specifies the address family which is either AF_INET or AF_INET6.</td>
</tr>
<tr>
<td>len</td>
<td>Specifies the length of the node binary format address.</td>
</tr>
<tr>
<td>error_num</td>
<td>Returns argument to the caller with the appropriate error code.</td>
</tr>
</tbody>
</table>

Return Values

The getipnodebyaddr subroutine returns a pointer to a hostent structure on success.

The getipnodebyaddr subroutine returns a null pointer if an error occurs. The error_num parameter is set to indicate the error.

Error Codes

<table>
<thead>
<tr>
<th>HOST_NOT_FOUND</th>
<th>The host specified by the name parameter was not found.</th>
</tr>
</thead>
<tbody>
<tr>
<td>TRY_AGAIN</td>
<td>The local server did not receive a response from an authoritative server. Try again later.</td>
</tr>
<tr>
<td>NO_RECOVERY</td>
<td>This error code indicates an unrecoverable error.</td>
</tr>
</tbody>
</table>
The requested name is valid but does not have an Internet address at the name server.

**Related Information**
The `freehostent` subroutine and `getipnodebyname` subroutine.

### getipnodebyname Subroutine

#### Purpose
Nodename-to-address translation.

#### Library
Standard C Library (`libc.a`)

Application (libaixinet)

#### Syntax
```c
#include <libc.a>
#include <netdb.h>
struct hostent *getipnodebyname(const char *name, int af, int flags, int *error_num);
```

#### Description
The commonly used functions `gethostbyname` and `gethostbyname2` are inadequate for many applications. You could not specify the type of addresses desired in `gethostbyname`. In `gethostbyname2`, a global option (`RES_USE_INET6`) is required when IPv6 addresses are used. Also, `gethostbyname2` needed more control over the type of addresses required.

The `getipnodebyname` subroutine gives the caller more control over the types of addresses required and is thread safe. It also does not need a global option like `RES_USE_INET6`.

The name argument can be either a node name or a numeric (either a dotted-decimal IPv4 or colon-separated IPv6) address.

The `flags` parameter values include `AI_DEFAULT`, `AI_V4MAPPED`, `AI_ALL`, and `AI_ADDRCONFIG`. The `res_flags` value `AI_DEFAULT` is designed to handle most applications. Its definition is:

```c
#define AI_DEFAULT (AI_V4MAPPED | AI_ADDRCONFIG)
```

When porting simple applications to use IPv6, simply replace the call:
```c
hp = gethostbyname(name);
```

with
```c
hp = getipnodebyname(name, AF_INET6, AI_DEFAULT, &error_num);
```

To modify the behavior of the `getipnodebyname` subroutine, constant values can be logically-ORed into the `flags` parameter.
A `flags` value of 0 implies a strict interpretation of the `af` parameter. If `af` is AF_INET then only IPv4 addresses are searched for and returned. If `af` is AF_INET6 then only IPv6 addresses are searched for and returned.

If the AI_V4MAPPED flag is specified along with an `af` of AF_INET6, then the caller accepts IPv4-mapped IPv6 addresses. That is, if a query for IPv6 addresses fails, then a query for IPv4 addresses is made and if any are found, then they are returned as IPv4-mapped IPv6 addresses. The AI_V4MAPPED flag is only valid with an `af` of AF_INET6.

If the AI_ALL flag is used in conjunction the AI_V4MAPPED flag and `af` is AF_INET6, then the caller wants all addresses. The addresses returned are IPv6 addresses and/or IPv4-mapped IPv6 addresses. Only if both queries (IPv6 and IPv4) fail does `getipnodebyname` return NULL. Again, the AI_ALL flag is only valid with an `af` of AF_INET6.

The AI_ADDRCONFIG flag is used to specify that a query for IPv6 addresses should only occur if the node has at least one IPv6 source address configured and a query for IPv4 addresses should only occur if the node has at least one IPv4 source address configured. For example, if the node only has IPv4 addresses configured, `af` equals AF_INET6, and the node name being looked up has both IPv4 and IPv6 addresses, then if only the AI_ADDRCONFIG flag is specified, `getipnodebyname` will return NULL. If the AI_V4MAPPED flag is specified with the AI_ADDRCONFIG flag (AI_DEFAULT), then any IPv4 addresses found will be returned as IPv4-mapped IPv6 addresses.

There are 4 different situations when the name argument is a literal address string:

1. `name` is a dotted-decimal IPv4 address and `af` is AF_INET. If the query is successful, then `h_name` points to a copy of `name`, `h_addrtype` is the `af` argument, `h_length` is 4, `h_aliases` is a NULL pointer, `h_addr_list[0]` points to the 4-byte binary address and `h_addr_list[1]` is a NULL pointer.
2. `name` is a colon-separated IPv6 address and `af` is AF_INET6. If the query is successful, then `h_name` points to a copy of `name`, `h_addrtype` is the `af` parameter, `h_length` is 16, `h_aliases` is a NULL pointer, `h_addr_list[0]` points to the 16-byte binary address and `h_addr_list[1]` is a NULL pointer.
3. `name` is a dotted-decimal IPv4 address and `af` is AF_INET6. If the AI_V4MAPPED flag is specified and the query is successful, then `h_name` points to an IPv4-mapped IPv6 address string, `h_addrtype` is the `af` argument, `h_length` is 16, `h_aliases` is a NULL pointer, `h_addr_list[0]` points to the 16-byte binary address and `h_addr_list[1]` is a NULL pointer.
4. `name` is a colon-separated IPv6 address and `af` is AF_INET. This is an error, `getipnodebyname` returns a NULL pointer and `error_num` equals HOST_NOT_FOUND.

### Parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>name</code></td>
<td>Specifies either a node name or a numeric (either a dotted-decimal IPv4 or colon-separated IPv6) address.</td>
</tr>
<tr>
<td><code>af</code></td>
<td>Specifies the address family which is either AF_INET or AF_INET6.</td>
</tr>
<tr>
<td><code>flags</code></td>
<td>Controls the types of addresses searched for and the types of addresses returned.</td>
</tr>
<tr>
<td><code>error_num</code></td>
<td>Returns argument to the caller with the appropriate error code.</td>
</tr>
</tbody>
</table>

### Return Values

The `getipnodebyname` subroutine returns a pointer to a `hostent` structure on success.

The `getipnodebyname` subroutine returns a null pointer if an error occurs. The `error_num` parameter is set to indicate the error.
Error Codes

<table>
<thead>
<tr>
<th>Error Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>HOST_NOT_FOUND</td>
<td>The host specified by the name parameter was not found.</td>
</tr>
<tr>
<td>TRY_AGAIN</td>
<td>The local server did not receive a response from an authoritative server.</td>
</tr>
<tr>
<td></td>
<td>Try again later.</td>
</tr>
<tr>
<td>NO_RECOVERY</td>
<td>The host specified by the name parameter was not found. This error code</td>
</tr>
<tr>
<td></td>
<td>indicates an unrecoverable error.</td>
</tr>
<tr>
<td>NO_ADDRESS</td>
<td>The requested name is valid but does not have an Internet address at the</td>
</tr>
<tr>
<td></td>
<td>name server.</td>
</tr>
</tbody>
</table>

Related Information

The `freehostent` subroutine and `getnodebyaddr` subroutine.

getlogin Subroutine

Purpose

Gets a user’s login name.

Library

Standard C Library (`libc.a`)

Syntax

```
#include <sys/types.h>
#include <unistd.h>
#include <limits.h>
char *getlogin (void)
```

Description

**Attention:** Do not use the `getlogin` subroutine in a multithreaded environment. To access the thread-safe version of this subroutine, see the `getlogin_r` subroutine.

**Attention:** The `getlogin` subroutine returns a pointer to an area that may be overwritten by successive calls.

The `getlogin` subroutine returns a pointer to the login name in the `/etc/utmp` file. You can use the `getpwnam` subroutine to locate the correct password file entry when the same user ID is shared by several login names.

If the `getlogin` subroutine cannot find the login name in the `/etc/utmp` file, it returns the process `LOGNAME` environment variable. If the `getlogin` subroutine is called within a process that is not attached to a terminal, it returns the value of the `LOGNAME` environment variable. If the `LOGNAME` environment variable does not exist, a null pointer is returned.

Return Values

The return value can point to static data whose content is overwritten by each call. If the login name is not found, the `getlogin` subroutine returns a null pointer.
Error Codes
If the getlogin function is unsuccessful, it returns one or more of the following error codes:

- **EMFILE**
  Indicates that the OPEN_MAX file descriptors are currently open in the calling process.

- **ENFILE**
  Indicates that the maximum allowable number of files is currently open in the system.

- **ENXIO**
  Indicates that the calling process has no controlling terminal.

Files
/\etc/utmp
Contains a record of users logged into the system.

Related Information
The getgrent, getgrgid, getgrnam, putgrent, setgrent, or endgrent subroutine, getlogin_r subroutine, getpwent, getpwpuid, setpwent, or endpwent subroutine, getpwnam subroutine. List of Security and Auditing Subroutines, Subroutines Overview in AIX 5L Version 5.3 General Programming Concepts: Writing and Debugging Programs.

getlogin_r Subroutine

Purpose
Gets a user’s login name.

Library
Thread-Safe C Library (libc_r.a)

Syntax
```c
int getlogin_r (Name, Length)
char * Name;
size_t Length;
```

Description
The getlogin_r subroutine gets a user’s login name from the /etc/utmp file and places it in the Name parameter. Only the number of bytes specified by the Length parameter (including the ending null value) are placed in the Name parameter.

Applications that call the getlogin_r subroutine must allocate memory for the login name before calling the subroutine. The name buffer must be the length of the Name parameter plus an ending null value.

If the getlogin_r subroutine cannot find the login name in the utmp file or the process is not attached to a terminal, it places the LOGNAME environment variable in the name buffer. If the LOGNAME environment variable does not exist, the Name parameter is set to null and the getlogin_r subroutine returns a -1.

Parameters

- **Name**
  Specifies a buffer for the login name. This buffer should be the length of the Length parameter plus an ending null value.
**Length** Specifies the total length in bytes of the Name parameter. No more bytes than the number specified by the Length parameter are placed in the Name parameter, including the ending null value.

**Return Values**
If successful, the getlogin_r function returns 0. Otherwise, an error number is returned to indicate the error.

**Error Codes**
If the getlogin_r subroutine does not succeed, it returns one of the following error codes:
- **EINVAL** Indicates that the Name parameter is not valid.
- **EMFILE** Indicates that the OPEN_MAX file descriptors are currently open in the calling process.
- **ENFILE** Indicates that the maximum allowable number of files are currently open in the system.
- **ENXIO** Indicates that the calling process has no controlling terminal.
- **ERANGE** Indicates that the value of Length is smaller than the length of the string to be returned, including the terminating null character.

**File**
/etc/utmp
Contains a record of users logged into the system.

**Related Information**
The getgrent_r, getgrgid_r, getgrnam_r, setgrent_r, or endgrent_r subroutine, getlogin subroutine, getpwnam_r, getpwuid_r, setpwnam_r, or setpwuid_r subroutine, getpwent_r subroutine, getpwuid, getgruid, getgrnam, getgrent, getpwnam, putpwent, setpwent, or endpwent subroutine.

List of Security and Auditing Subroutines, List of Multithread Subroutines, and Subroutines Overview in AIX 5L Version 5.3 General Programming Concepts: Writing and Debugging Programs.

---

**getnextprojdb Subroutine**

**Purpose**
Retrieves the next project from the specified project database.

**Library**
The libaacct.a library.

**Syntax**
```c
<sys/aacct.h>

getnextprojdb(void *handle, struct projdb *project, char *comm)
```

**Description**
The getnextprojdb subroutine retrieves the next project definitions from the project database named through the handle parameter. The caller must initialize the project database prior to calling this routine with the projdballoc routine. Upon successful completion, the project information is copied to the project.
structure specified by the caller. In addition, the associated project comment, if present, is copied to the buffer pointed to by the `comm` parameter. The comment buffer is allocated by the caller and must have a length of 1024 bytes.

There is an internal state (that is, the current project) associated with the project database. When the project database is initialized, the current project is the first project in the database. The `getnextprojdb` subroutine returns the current project and advances the current project assignment to the next project in the database so that successive calls read each project entry in the database. When the last project is read, the current project assignment is advanced to the end of the database. Any attempt to read beyond the end of the project database results in a failure.

**Parameters**

- `handle`  
  Pointer to the `projdb` handle.
- `project`  
  Pointer to project structure where the retrieved data is stored.
- `comm`  
  Comment associated with the project in the database.

**Security**

No restriction. Any user can call this function.

**Return Values**

- **0**  
  Success
- **-1**  
  Failure

**Error Codes**

- **EINVAL**  
  Invalid arguments, if passed pointer is NULL.
- **ENOENT**  
  End of the project database.
- **ENOENT**  
  No projects available.

**Related Information**


---

**getopt Subroutine**

**Purpose**

Returns the next flag letter specified on the command line.

**Library**

Standard C Library (`libc.a`)

**Syntax**

```
#include <unistd.h>
```
int getopt (ArgumentC, ArgumentV, OptionString);
int ArgumentC;
char *const ArgumentV[];
const char *OptionString;

extern int optind;
extern int optopt;
extern int opterr;
extern char *optarg;

Description
The optind parameter indexes the next element of the ArgumentV parameter to be processed. It is initialized to 1 and the getopt subroutine updates it after calling each element of the ArgumentV parameter.

The getopt subroutine returns the next flag letter in the ArgumentV parameter list that matches a letter in the OptionString parameter. If the flag takes an argument, the getopt subroutine sets the optarg parameter to point to the argument as follows:

- If the flag was the last letter in the string pointed to by an element of the ArgumentV parameter, the optarg parameter contains the next element of the ArgumentV parameter and the optind parameter is incremented by 2. If the resulting value of the optind parameter is not less than the ArgumentC parameter, this indicates a missing flag argument, and the getopt subroutine returns an error message.
- Otherwise, the optarg parameter points to the string following the flag letter in that element of the ArgumentV parameter and the optind parameter is incremented by 1.

Parameters
ArgumentC Specifies the number of parameters passed to the routine.
ArgumentV Specifies the list of parameters passed to the routine.
OptionString Specifies a string of recognized flag letters. If a letter is followed by a : (colon), the flag is expected to take a parameter that may or may not be separated from it by white space.
optind Specifies the next element of the ArgumentV array to be processed.
optopt Specifies any erroneous character in the OptionString parameter.
opterr Indicates that an error has occurred when set to a value other than 0.
optarg Points to the next option flag argument.

Return Values
The getopt subroutine returns the next flag letter specified on the command line. A value of -1 is returned when all command line flags have been parsed. When the value of the ArgumentV [optind] parameter is null, *ArgumentV [optind] is not the - (minus) character, or ArgumentV [optind] points to the "-" (minus) string, the getopt subroutine returns a value of -1 without changing the value. If ArgumentV [optind] points to the "- -" (double minus) string, the getopt subroutine returns a value of -1 after incrementing the value of the optind parameter.

Error Codes
If the getopt subroutine encounters an option character that is not specified by the OptionString parameter, a ? (question mark) character is returned. If it detects a missing option argument and the first character of OptionString is a : (colon), then a : (colon) character is returned. If this subroutine detects a missing option argument and the first character of OptionString is not a colon, it returns a ? (question mark). In either case, the getopt subroutine sets the optopt parameter to the option character that caused the error. If the application has not set the opterr parameter to 0 and the first character of OptionString is not a : (colon), the getopt subroutine also prints a diagnostic message to standard error.
Examples

The following code fragment processes the flags for a command that can take the mutually exclusive flags a and b, and the flags f and o, both of which require parameters.

```c
#include <unistd.h>  /*Needed for access subroutine constants*/
main(argc, argv)
int argc;
char **argv;
{
    int c;
    extern int optind;
    extern char *optarg;
    .
    .
    .
    while ((c = getopt(argc, argv, "abf:o:")) != EOF)
    {
        switch (c)
        {
            case 'a':
                if (bflg)
                    errflg++;
                else
                    aflg++;
                break;
            case 'b':
                if (aflg)
                    errflg++;
                else
                    bflg++;
                break;
            case 'f':
                ifile = optarg;
                break;
            case 'o':
                ofile = optarg;
                break;
            case '?':
                errflg++;
        }  /* case */
        if (errflg)
        {
            fprintf(stderr, "usage: . . ");
            exit(2);
        }  /* while */
        for ( ; optind < argc; optind++)
        {
            if (access(argv[optind], R_OK))
            {
                .
                .
            }  /* for */
        }  /* main */
```

Related Information

The `getopt` command.

List of Executable Program Creation Subroutines, Subroutines Overview, and List of Multithread Subroutines in AIX 5L Version 5.3 General Programming Concepts: Writing and Debugging Programs.
getpagesize Subroutine

Purpose
Gets the system page size.

Library
Standard C Library (libc.a)

Syntax
#include <unistd.h>
int getpagesize( )

Description
The getpagesize subroutine returns the number of bytes in a page. Page granularity is the granularity for many of the memory management calls.

The page size is determined by the system and may not be the same as the underlying hardware page size.

Related Information
The brk or sbrk subroutine.

The size command.

Program Address Space Overview and Subroutines Overview in AIX 5L Version 5.3 General Programming Concepts: Writing and Debugging Programs.

getpaginfo Subroutine

Purpose
Retrieves a Process Authentication Group (PAG) flags for a given PAG type.

Library
Security Library (libc.a)

Syntax
#include <pag.h>
int getpaginfo ( name, infop, infosz )
char * name;
struct paginfo * infop;
int infosz;

Description
The getpaginfo subroutine retrieves the PAG flags for a given PAG name. For this function to succeed, the PAG name must be registered with the operating system before this subroutine is called. The infop parameter must be a valid, referenced PAG info structure of the size specified by infosz.
Parameters

name A 1-character to 4-character, NULL-terminated name for the PAG type. Typical values include afs, dfs, pki, and krb5.
infop Points to a paginfo struct where the operating system returns the PAG flags.
infosz Indicates the size of the PAG info structure.

Return Values
A value of 0 is returned upon successful completion. If the getpaginfo subroutine fails a value of -1 is returned and the errno global variable is set to indicate the error.

Error Codes
The getpaginfo subroutine fails if the following condition is true:

EINVAL The named PAG type does not exist as part of the table.

Other errors might be set by subroutines invoked by the getpaginfo subroutine.

Related Information
pag_getid System Call, pag_getname System Call, pag_getvalue System Call, pag_setname System Call, pag_setvalue System Call, kcred_genpagvalue Kernel Service, kcred_getpagid Kernel Service, and kcred_getpagname Kernel Service.

List of Security and Auditing Subroutines in AIX 5L Version 5.3 General Programming Concepts.

getpagvalue or getpagvalue64 Subroutine

Purpose
Returns the Process Authentication Group (PAG) value for a given PAG type.

Library
Security Library (libc.a)

Syntax
#include <pag.h>

int getpagvalue ( name )
char * name;

uint64_t getpagvalue64( name );
char * name;

Description
The getpagvalue and getpagvalue64 subroutines retrieve the PAG value for a given PAG name. For these functions to succeed, the PAG name must be registered with the operating system before these subroutines are called.

Parameters
name A 1-character to 4-character, NULL-terminated name for the PAG type. Typical values include afs, dfs, pki, and krb5.
Return Values
The getpagvalue and getpagvalue64 subroutines return a PAG value upon successful completion. Upon a failure, a value of -1 is returned and the errno global variable is set to indicate the error.

Error Codes
The getpagvalue and getpagvalue64 subroutines fail if the following condition is true:

EINVAL
The named PAG type does not exist as part of the table.

Other errors might be set by subroutines invoked by the getpagvalue and getpagvalue64 subroutines.

Related Information

List of Security and Auditing Subroutines in AIX 5L Version 5.3 General Programming Concepts.

getpass Subroutine

Purpose
Reads a password.

Library
Standard C Library (libc.a)

Syntax
#include <stdlib.h>

char *getpass (Promt)
char *Promt;

Description
Attention: The characters are returned in a static data area. Subsequent calls to this subroutine overwrite the static data area.

The getpass subroutine does the following:
• Opens the controlling terminal of the current process.
• Writes the characters specified by the Prompt parameter to that device.
• Reads from that device the number of characters up to the value of the PASS_MAX constant until a new-line or end-of-file (EOF) character is detected.
• Restores the terminal state and closes the controlling terminal.

During the read operation, character echoing is disabled.

The getpass subroutine is not safe in a multithreaded environment. To use the getpass subroutine in a threaded application, the application must keep the integrity of each thread.
Parameters

Prompt Specifies a prompt to display on the terminal.

Return Values
If this subroutine is successful, it returns a pointer to the string. If an error occurs, the subroutine returns a null pointer and sets the_errno global variable to indicate the error.

Error Codes
If the getpass subroutine is unsuccessful, it returns one or more of the following error codes:

EINTR Indicates that an interrupt occurred while the getpass subroutine was reading the terminal device. If a SIGINT or SIGQUIT signal is received, the getpass subroutine terminates input and sends the signal to the calling process.
ENXIO Indicates that the process does not have a controlling terminal.

Note: Any subroutines called by the getpass subroutine may set other error codes.

Related Information
The getuserpw (“getuserpw, putuserpw, or putuserpwhist Subroutine” on page 463) subroutine, newpass (“newpass Subroutine” on page 891) subroutine.

Note: Any subroutines called by the getpass subroutine may set other error codes.

Related Information
The getuserpw (“getuserpw, putuserpw, or putuserpwhist Subroutine” on page 463) subroutine, newpass (“newpass Subroutine” on page 891) subroutine.

List of Security and Auditing Subroutines Subroutines Overview in AIX SL Version 5.3 General Programming Concepts: Writing and Debugging Programs.

getpcred Subroutine

Purpose
Reads the current process credentials.

Library
Security Library (libc.a)

Syntax

```
#include <usersec.h>

char **getpcred (Which);
int Which;
```

Description
The getpcred subroutine reads the specified process security credentials and returns a pointer to a NULL terminated array of pointers in allocated memory. Each pointer in the array points to a string containing an attribute/value pair in allocated memory. It's the responsibility of the caller to free each individual string as well as the array of pointers.
Parameters

Which Specifies which credentials are read. This parameter is a bit mask and can contain one or more of the following values, as defined in the usersec.h file:

- **CRED_RUID**
  - Real user name
- **CRED_LUID**
  - Login user name
- **CRED_RGID**
  - Real group name
- **CRED_GROUPS**
  - Supplementary group ID
- **CRED_AUDIT**
  - Audit class of the current process
  - **Note:** A process must have root user authority to retrieve this credential. Otherwise, the getpcred subroutine returns a null pointer and the errno global variable is set to EPERM.
- **CRED_RLIMITS**
  - BSD resource limits
  - **Note:** Use the `getrlimit` subroutine to control resource consumption.
- **CRED_UMASK**
  - The umask.

If the Which parameter is null, all credentials are returned.

Return Values

When successful, the getpcred subroutine returns a pointer to a NULL terminated array of string pointers containing the requested values. If the getpcred subroutine is unsuccessful, a NULL pointer is returned and the errno global variable is set to indicate the error.

Error Codes

The getpcred subroutine fails if either of the following are true:

- **EINVAL** The Which parameter contains invalid credentials requests.
- **EPERM** The process does not have the proper authority to retrieve the requested credentials.

Other errors can also be set by any subroutines invoked by the getpcred subroutine.

Related Information

The ckuseracct subroutine, ckuserID subroutine, getpenv subroutine, setpcred subroutine.

[List of Security and Auditing Subroutines][1] [Subroutines Overview][2] in AIX 5L Version 5.3 General Programming Concepts: Writing and Debugging Programs.
getpeereid Subroutine

Purpose
Gets the effective user ID and effective group ID of a peer on a connected UNIX domain socket.

Syntax
```
#include <sys/types.h>
int getpeereid(int socket, uid_t *euid, gid_t *egid);
```

Description
The `getpeereid` subroutine returns the effective user and group IDs of the peer connected to a stream socket in the UNIX domain. The effective user and group IDs are saved in the socket, to be returned, when the peer calls `connect` or `listen`.

Parameters
- `socket` Specifies the descriptor number of a connected socket.
- `euid` The effective user ID of the peer socket.
- `egid` The effective group ID of the peer socket.

Return Values
When the `getpeereid` subroutine successfully completes, a value of 0 is returned and the `euid` and `egid` parameters hold the effective user ID and group ID, respectively.

If the `getpeereid` subroutine is unsuccessful, the system handler returns a value of -1 to the calling program and sets the `errno` global variable to an error code that indicates the specific error.

Error Codes
The `getpeereid` subroutine is unsuccessful if any of the following errors occurs:

- `EBADF` The `socket` parameter is not valid.
- `ENOTSOCK` The `socket` parameter refers to a file, not a socket.
- `ENOTCONN` The socket is not connected.
- `ENOBUFFS` Insufficient resources were available in the system to complete the call.
- `EFAULT` The `address` parameter is not in a writable part of the user address space.

Note: The `getpeerid` technology used to support this function in AIX was originally published by D. J. Bernstein, Associate Professor, Department of Mathematics, Statistics, and Computer Science, University of Illinois at Chicago. In addition, the specific `getpeerid` syntax reflected originated with William Erik Baxter. All the aforementioned are used by AIX with permission.

getpenv Subroutine

Purpose
Reads the current process environment.

Library
Security Library (`libc.a`)
Syntax
#include <usersec.h>

char **getpenv ( int Which);

Description
The getpenv subroutine reads the specified environment variables and returns them in a character buffer.

Parameters

Which Specifies which environment variables are to be returned. This parameter is a bit mask and may contain one or more of the following values, as defined in the usersec.h file:

PENV_USR The normal user-state environment. Typically, the shell variables are contained here.
PENV_SYS The system-state environment. This data is located in system space and protected from unauthorized access.

All variables are returned by setting the Which parameter to logically OR the PENV_USER and PENV_SYSTEM values.

The variables are returned in a null-terminated array of character pointers in the form var=val. The user-state environment variables are prefaced by the string USRENVIRON:, and the system-state variables are prefaced with SYSENVIRON:. If a user-state environment is requested, the current directory is always returned in the PWD variable. If this variable is not present in the existing environment, the getpenv subroutine adds it to the returned string.

Return Values
Upon successful return, the getpenv subroutine returns the environment values. If the getpenv subroutine fails, a null value is returned and the errno global variable is set to indicate the error.

Note: This subroutine can partially succeed, returning only the values that the process permits it to read.

Error Codes
The getpenv subroutine fails if one or more of the following are true:

EINVAL The Which parameter contains values other than PENV_USR or PENV_SYS.

Other errors can also be set by subroutines invoked by the getpenv subroutine.

Related Information
The ckuseracct subroutine, ckuserID subroutine, getpcred subroutine, setpenv subroutine.

List of Security and Auditing Subroutines, Subroutines Overview in AIX 5L Version 5.3 General Programming Concepts: Writing and Debugging Programs.
getpgid Subroutine

Purpose
Returns the process group ID of the calling process.

Library
Standard C Library (libc.a)

Syntax
#include <unistd.h>

pid_t getpgid (Pid)
(pid_ Pid)

Description
The getpgid subroutine returns the process group ID of the process whose process ID is equal to that specified by the Pid parameter. If the value of the Pid parameter is equal to (pid_t)0, the getpgid subroutine returns the process group ID of the calling process.

Parameter
Pid
The process ID of the process to return the process group ID for.

Return Values
id
The process group ID of the requested process
-1
Not successful and errno set to one of the following.

Error Code
ESRCH
There is no process with a process ID equal to Pid.

EPERM
The process whose process ID is equal to Pid is not in the same session as the calling process.

EINVAL
The value of the Pid argument is invalid.

Related Information
The exec ("exec: execl, execlp, execv, execve, execvp, or exect Subroutine" on page 235) subroutine, fork ("fork, f_fork, or vfork Subroutine" on page 287) subroutine, getpid ("getpid, getpgid, or getpgrp Subroutine" on page 430) subroutine, getsid ("getsid Subroutine" on page 431) subroutine, setpgid subroutine, setsid subroutine.

g getpid, getpgrp, or getppid Subroutine

Purpose
Returns the process ID, process group ID, and parent process ID.
Syntax

```c
#include <unistd.h>
pid_t getpid (void)
pid_t getpgrp (void)
pid_t getppid (void)
```

Description

The `getpid` subroutine returns the process ID of the calling process.

The `getpgrp` subroutine returns the process group ID of the calling process.

The `getppid` subroutine returns the process ID of the calling process' parent process.

Related Information

The `exec` subroutine, `fork`, `setpgrp`, `setpgid`, `sigaction`, `sigvec`, or `signal` subroutine.

**getportattr or putportattr Subroutine**

**Purpose**

Accesses the port information in the port database.

**Library**

Security Library (libc.a)

**Syntax**

```c
#include <usersec.h>

int getportattr (Port, Attribute, Value, Type)
char *Port;
char *Attribute;
void *Value;
int Type;

int putportattr (Port, Attribute, Value, Type)
char *Port;
char *Attribute;
void *Value;
int Type;
```

**Description**

The `getportattr` or `putportattr` subroutine accesses port information. The `getportattr` subroutine reads a specified attribute from the port database. If the database is not already open, the `getportattr` subroutine implicitly opens the database for reading. The `putportattr` subroutine writes a specified attribute into the port database. If the database is not already open, the `putportattr` subroutine implicitly opens the database for reading and writing. The data changed by the `putportattr` subroutine must be explicitly committed by calling the `putportattr` subroutine with a `Type` parameter equal to the `SEC_COMMIT` value. Until all the data is committed, only these subroutines within the process return the written data.
Values returned by these subroutines are in dynamically allocated buffers. You do not need to move the values prior to the next call.

Use the `setuserdb` or `enduserdb` subroutine to open and close the port database.

**Parameters**

- **Port**
  Specifies the name of the port for which an attribute is read.

- **Attribute**
  Specifies the name of the attribute read. This attribute can be one of the following values defined in the `usersec.h` file:

  - **S_HERALD**: Defines the initial message printed when the `getty` or `login` command prompts for a login name. This value is of the type `SEC_CHAR`.
  
  - **S_SAKENABLED**: Indicates whether or not trusted path processing is allowed on this port. This value is of the type `SEC_BOOL`.
  
  - **S_SYNONYM**: Defines the set of ports that are synonyms attributes for the given port. This value is of the type `SEC_LIST`.
  
  - **S_LOGTIMES**: Defines when the user can access the port. This value is of the type `SEC_LIST`.
  
  - **S_LOGDISABLE**: Defines the number of unsuccessful login attempts that result in the system locking the port. This value is of the type `SEC_INT`.
  
  - **S_LOGINTERVAL**: Defines the time interval in seconds within which `S_LOGDISABLE` number of unsuccessful login attempts must occur before the system locks the port. This value is of the type `SEC_INT`.
  
  - **S_LOGREENABLE**: Defines the time interval in minutes after which a system-locked port is unlocked. This value is of the type `SEC_INT`.
  
  - **S_LOGDELAY**: Defines the delay factor in seconds between unsuccessful login attempts. This value is of the type `SEC_INT`.
  
  - **S_LOCKTIME**: Defines the time in seconds since the epoch (zero time, January 1, 1970) that the port was locked. This value is of the type `SEC_INT`.
  
  - **S_ULOGTIMES**: Lists the times in seconds since the epoch (midnight, January 1, 1970) when unsuccessful login attempts occurred. This value is of the type `SEC_LIST`.
  
  - **S_USERNAMEECHO**: Indicates whether user name input echo and user name masking is enabled for the port. This value is of the type `SEC_BOOL`.
  
  - **S_PWD_PROMPT**: Defines the password prompt message printed when requesting password input. This value is of the type `SEC_CHAR`.

- **Value**
  Specifies the address of a buffer in which the attribute is stored with `putportattr` or is to be read with `getportattr`.
Type  Specifies the type of attribute expected. The following types are valid and defined in the usersec.h file:

SEC_INT  Indicates the format of the attribute is an integer. The buffer returned by the getportattr subroutine and the buffer supplied by the putportattr subroutine are defined to contain an integer.

SEC_CHAR  Indicates the format of the attribute is a null-terminated character string.

SEC_LIST  Indicates the format of the attribute is a list of null-terminated character strings. The list itself is null terminated.

SEC_BOOL  An integer with a value of either 0 or 1, or a pointer to a character pointing to one of the following strings:
  • True
  • Yes
  • Always
  • False
  • No
  • Never

SEC_COMMIT  Indicates that changes to the specified port are committed to permanent storage if specified alone for the putportattr subroutine. The Attribute and Value parameters are ignored. If no port is specified, changes to all modified ports are committed.

SEC_DELETE  Deletes the corresponding attribute from the database.

SEC_NEW  Updates all of the port database files with the new port name when using the putportattr subroutine.

Security
Access Control: The calling process must have access to the port information in the port database.

File Accessed:

\[\text{rw} \quad /\text{etc/security/login.cfg} \]
\[\text{rw} \quad /\text{etc/security/portlog} \]

Return Values
The getportattr and putportattr subroutines return a value of 0 if completed successfully. Otherwise, a value of -1 is returned and the errno global value is set to indicate the error.

Error Codes
These subroutines are unsuccessful if the following values are true:

EACCES  Indicates that access permission is denied for the data requested.
ENOENT  Indicates that the Port parameter does not exist or the attribute is not defined for the specified port.
ENOATTR  Indicates that the specified port attribute does not exist for the specified port.
EINVAL Indicates that the *Attribute* parameter does not contain one of the defined attributes or is a null value.

EINVAL Indicates that the *Value* parameter does not point to a valid buffer or to valid data for this type of attribute.

EPERM Operation is not permitted.

Related Information
The `setuserdb` or `enduserdb` subroutine.

List of Security and Auditing Services in *AIX 5L Version 5.3 General Programming Concepts: Writing and Debugging Programs.*

Subroutines Overview in *AIX 5L Version 5.3 General Programming Concepts: Writing and Debugging Programs.*

---

**getpri Subroutine**

**Purpose**
Returns the scheduling priority of a process.

**Library**
Standard C Library (*libc.a*)

**Syntax**

```c
int getpri (ProcessID);
pid_t ProcessID;
```

**Description**
The `getpri` subroutine returns the scheduling priority of a process.

**Parameters**

- **ProcessID**
  Specifies the process ID. If this value is 0, the current process scheduling priority is returned.

**Return Values**
Upon successful completion, the `getpri` subroutine returns the scheduling priority of a thread in the process. Otherwise, a value of -1 is returned and the `errno` global variable is set to indicate the error.

**Error Codes**
The `getpri` subroutine is unsuccessful if one of the following is true:

- EPERM A process was located, but its effective and real user ID did not match those of the process executing the `getpri` subroutine, and the calling process did not have root user authority.
- ESRCH No process can be found corresponding to that specified by the `ProcessID` parameter.
getpriority, setpriority, or nice Subroutine

Purpose
Gets or sets the nice value.

Libraries
getpriority, setpriority: Standard C Library (libc.a)
nice: Standard C Library (libc.a)

Berkeley Compatibility Library (libbsd.a)

Syntax
```
#include <sys/resource.h>

int getpriority(int Which, int Who);

int setpriority(int Which, int Who, int Priority);

#include <unistd.h>

int nice(int Increment);
```

Description
The nice value of the process, process group, or user, as indicated by the Which and Who parameters is obtained with the getpriority subroutine and set with the setpriority subroutine.

The getpriority subroutine returns the highest priority nice value (lowest numerical value) pertaining to any of the specified processes. The setpriority subroutine sets the nice values of all of the specified processes to the specified value. If the specified value is less than -20, a value of -20 is used; if it is greater than 20, a value of 20 is used. Only processes that have root user authority can lower nice values.

The nice subroutine increments the nice value by the value of the Increment parameter.

Note: Nice values are only used for the scheduling policy SCHED_OTHER, where they are combined with a calculation of recent cpu usage to determine the priority value.

To provide upward compatibility with older programs, the nice interface, originally found in AT&T System V, is supported.
Note: Process priorities in AT&T System V are defined in the range of 0 to 39, rather than -20 to 20 as in BSD, and the nice library routine is supported by both. Accordingly, two versions of the nice are supported by AIX Version 3. The default version behaves like the AT&T System V version, with the Increment parameter treated as the modifier of a value in the range of 0 to 39 (0 corresponds to -20, 39 corresponds to 9, and priority 20 is not reachable with this interface).

If the behavior of the BSD version is desired, compile with the Berkeley Compatibility Library (libbsd.a). The Increment parameter is treated as the modifier of a value in the range -20 to 20.

Parameters

Which Specifies one of PRIO_PROCESS, PRIO_PGRP, or PRIO_USER.
Who Interpreted relative to the Which parameter (a process identifier, process group identifier, and a user ID, respectively). A zero value for the Who parameter denotes the current process, process group, or user.
Priority Specifies a value in the range -20 to 20. Negative nice values cause more favorable scheduling.
Increment Specifies a value that is added to the current process nice value. Negative values can be specified, although values exceeding either the high or low limit are truncated.

Return Values

On successful completion, the getpriority subroutine returns an integer in the range -20 to 20. A return value of -1 can also indicate an error, and in this case the errno global variable is set.

On successful completion, the setpriority subroutine returns 0. Otherwise, -1 is returned and the global variable errno is set to indicate the error.

On successful completion, the nice subroutine returns the new nice value minus NZERO. Otherwise, a value of -1 is returned and the errno global variable is set to indicate the error.

Note: A value of -1 can also be returned. In that case, the calling process should also check the errno global variable.

Error Codes

The getpriority and setpriority subroutines are unsuccessful if one of the following is true:

ESRCH No process was located using the Which and Who parameter values specified.
EINVAL The Which parameter was not recognized.

In addition to the errors indicated above, the setpriority subroutine is unsuccessful if one of the following is true:

EPERM A process was located, but neither the effective nor real user ID of the caller of the process executing the setpriority subroutine has root user authority.
EACCESS The call to setpriority would have changed the priority of a process to a value lower than its current value, and the effective user ID of the process executing the call did not have root user authority.

The nice subroutine is unsuccessful if the following is true:

EPERM The Increment parameter is negative or greater than 2 * {NZERO} and the calling process does not have appropriate privileges.
Related Information
The exec subroutine.

Subroutines Overview in AIX 5L Version 5.3 General Programming Concepts: Writing and Debugging Programs.

getproclist, getlparlist, or getarmlist Subroutine

Purpose
Retrieve the transaction records from the advanced accounting data file.

Library
The libaacct.a library.

Syntax
```
#include <sys/aacct.h>
int getproclist(char *filename, long long begin_time, long long end_time, struct aacct_tran **p_list);
int getlparlist(char *filename, long long begin_time, long long end_time, struct aacct_tran **l_list);
int getarmlist(char *filename, long long begin_time, long long end_time, struct aacct_tran **t_list);
```

Description
The getproclist, getlparlist, and getarmlist subroutines parse the specified advanced accounting data file and retrieve the process, LPAR, and ARM transaction records, respectively. The retrieved transaction records are returned in the form of a linked list of type struct aacct_tran_rec.

These APIs can be called multiple times with different accounting data file names in order to generate a consolidated list of transaction records from multiple data files. They append the new file data to the end of the linked list pointed to by the p_list, l_list, and t_list arguments. They also internally sort the transaction records based on the time of transaction, which gives users a time-sorted list of transaction records from these routines.

The getproclist, getlparlist, and getarmlist subroutines can also be used to retrieve the intended transaction records for a particular interval of time by passing the begin and end times of the interval as arguments to these routines. If these interval arguments are specified as -1, transaction records for all the intervals are retrieved.

Parameters
- **begin_time**: Specifies the start timestamp for collecting records in a particular intervals. The input is in seconds since EPOCH. Specifying -1 retrieves all the records.
- **end_time**: Specifies the end timestamp for collecting records in a particular intervals. The input is in seconds since EPOCH. Specifying -1 retrieves all the records.
- **filename**: Name of the advanced accounting data file.
- **l_list**: Pointers to the linked list of aacct_tran_rec structures, which hold the retrieved LPAR records.
- **p_list**: Pointers to the linked list of aacct_tran_rec structures, which hold the retrieved process records.
- **t_list**: Pointers to the linked list of aacct_tran_rec structures, which hold the retrieved ARM records.
Security
No restrictions. Any user can call this function.

Return Values

<table>
<thead>
<tr>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>The call to the subroutine was successful.</td>
</tr>
<tr>
<td>-1</td>
<td>The call to the subroutine failed.</td>
</tr>
</tbody>
</table>

Error Codes

- **EINVAL**: The passed pointer is NULL.
- **ENOENT**: Specified data file does not exist.
- **EPERM**: Permission denied. Unable to read the data file.
- **ENOMEM**: Insufficient memory.

Related Information

The "agg_proc_stat, agg_lpar_stat, agg_arm_stat, or free_agg_list Subroutine" on page 36, "buildproclist Subroutine" on page 125, "buildtranlist or freetranlist Subroutine" on page 126.

Understanding the Advanced Accounting Subsystem

getprocs Subroutine

Purpose
Gets process table entries.

Library
Standard C library (libc.a)
Syntax

```c
#include <procinfo.h>
#include <sys/types.h>

int getprocs (ProcessBuffer, ProcessSize, FileBuffer, FileSize, IndexPointer, Count)
struct procsinfo *ProcessBuffer;
or struct procsinfo64 *ProcessBuffer;
int ProcessSize;
struct fdsinfo *FileBuffer;
int FileSize;
pid_t *IndexPointer;
int Count;

int getprocs64 (ProcessBuffer, ProcessSize, FileBuffer, FileSize, IndexPointer, Count)
struct procentry64 *ProcessBuffer;
int ProcessSize;
struct fdsinfo64 *FileBuffer;
int FileSize;
pid_t *IndexPointer;
int Count;
```

Description

The `getprocs` subroutine returns information about processes, including process table information defined by the `procsinfo` structure, and information about the per-process file descriptors defined by the `fdsinfo` structure.

The `getprocs` subroutine retrieves up to `Count` process table entries, starting with the process table entry corresponding to the process identifier indicated by `IndexPointer`, and places them in the array of `procsinfo` structures indicated by the `ProcessBuffer` parameter. File descriptor information corresponding to the retrieved processes are stored in the array of `fdsinfo` structures indicated by the `FileBuffer` parameter.

On return, the process identifier referenced by `IndexPointer` is updated to indicate the next process table entry to be retrieved. The `getprocs` subroutine returns the number of process table entries retrieved.

The `getprocs` subroutine is normally called repeatedly in a loop, starting with a process identifier of zero, and looping until the return value is less than `Count`, indicating that there are no more entries to retrieve.

**Note:** The process table may change while the `getprocs` subroutine is accessing it. Returned entries will always be consistent, but since processes can be created or destroyed while the `getprocs` subroutine is running, there is no guarantee that retrieved entries will still exist, or that all existing processes have been retrieved.

When used in 32-bit mode, limits larger than can be represented in 32 bits are truncated to `RLIM_INFINITY`. Large `rusage` and other values are truncated to `INT_MAX`. Alternatively, the `struct procsinfo64` and `sizeof (struct procsinfo64)` can be used by 32-bit `getprocs` to return full 64-bit process information. Note that the `procsinfo` structure not only increases certain `procsinfo` fields from 32 to 64 bits, but that it contains additional information not present in `procsinfo`. The `struct procsinfo64` contains the same data as `struct procsinfo` when compiled in a 64-bit program.
In AIX 5.1 and later, 64-bit applications are required to use getprocs64() and procentry64. Note that struct procentry64 contains the same information as struct procsinfo64, with the addition of support for the 64-bit time_t and dev_t, and the 256-bit sigset_t. The procentry64 structure also contains a new version of struct ucred (struct ucred_ext) and a new, expanded struct rusage (struct trusage64) as described in <sys/cred.h> and <sys/resource.h> respectively. Application developers are also encouraged to use getprocs64() in 32-bit applications to obtain 64-bit process information as this interface provides the new, larger types. The getprocs() interface will still be supported for 32-bit applications using struct procsinfo or struct procsinfo64 but will not be available to 64-bit applications.

Parameters

ProcessBuffer
Specifies the starting address of an array of procsinfo, procsinfo64, or procentry64 structures to be filled in with process table entries. If a value of NULL is passed for this parameter, the getprocs subroutine scans the process table and sets return values as normal, but no process entries are retrieved.

Note: The ProcessBuffer parameter of getprocs subroutine contains two struct rusage fields named pi_ru and pi_cru. Each of these fields contains two struct timeval fields named ru_ultime and ru_stime. The tv_usec field in both of the struct timeval contain nanoseconds instead of microseconds. These values come from the struct user fields named U_ru and U_cru.

ProcessSize
Specifies the size of a single procsinfo, procsinfo64, or procentry64 structure.

FileBuffer
Specifies the starting address of an array of fdsinfo, or fdsinfo64 structures to be filled in with per-process file descriptor information. If a value of NULL is passed for this parameter, the getprocs subroutine scans the process table and sets return values as normal, but no file descriptor entries are retrieved.

FileSize
Specifies the size of a single fdsinfo, or fdsinfo64 structure.

IndexPointer
Specifies the address of a process identifier which indicates the required process table entry. A process identifier of zero selects the first entry in the table. The process identifier is updated to indicate the next entry to be retrieved.

Note: The IndexPointer does not have to correspond to an existing process, and may in fact correspond to a different process than the one you expect. There is no guarantee that the process slot pointed to by IndexPointer will contain the same process between successive calls to getprocs() or getprocs64().

Count
Specifies the number of process table entries requested.

Return Values
If successful, the getprocs subroutine returns the number of process table entries retrieved; if this is less than the number requested, the end of the process table has been reached. A value of 0 is returned when the end of the process table has been reached. Otherwise, a value of -1 is returned, and the errno global variable is set to indicate the error.

Error Codes
The getprocs subroutine does not succeed if the following are true:

EINVAL
The ProcessSize or FileSize parameters are invalid, or the IndexPointer parameter does not point to a valid process identifier, or the Count parameter is not greater than zero.
The copy operation to one of the buffers was not successful.

Related Information
The `getpid` ("getpid, getpgrp, or getppid Subroutine" on page 402), `getpgrp` ("getpid, getpgrp, or getppid Subroutine" on page 402), or `getppid` ("getpid, getpgrp, or getppid Subroutine" on page 402) subroutines, the `getthrds` ("getthrds Subroutine" on page 438) subroutine.

The `ps` command.

getproj Subroutine

Purpose
Retrieves the project definition from the kernel project registry for the requested project name.

Library
The `libaacct.a` library.

Syntax
```
<sys/aacct.h>

getproj(struct project *, int flag)
```

Description
The `getproj` subroutine functions similar to the `getprojs` subroutine with the exception that the `getproj` subroutine retrieves the definition only for the project name or number, which is passed as its argument. The `flag` parameter indicates what is passed. The `flag` parameter has the following values:

- PROJ_NAME — Indicates that the supplied project definition only has the project name. The `getproj` subroutine queries the kernel to obtain a match for the supplied project name and returns the matching entry.
- PROJ_NUM — Indicates that the supplied project definition only has the project number. The `getproj` subroutine queries the kernel to obtain a match for the supplied project number and returns the matching entry.

Generally, the projects are loaded from the system project definition file or LDAP, or from both. When more than one of these project repositories are used, project name and project ID collisions are possible. These projects are differentiated by the kernel using an origin flag. This origin flag designates the project repository from where the project definition is obtained. If the caller wants to retrieve the project definition that belongs to a specific project repository, the specific origin value should be passed in the flags field of the project structure. Valid project origins values that can be passed are defined in the `sys/aacct.h` file. If the projects are currently loaded from the project repository represented by the origin value, `getproj` returns the specified project if it exists. If the origin value is not passed, the first project reference found in the kernel registry is returned. Regardless of whether the origin is passed or not, `getproj` always returns the project origin flags in the output project structure.

Parameters

- `project` Pointer holding the project whose information is required.
- `flag` An integer flag that indicates whether the match needs to be performed on the supplied project name or number.
Security
There are no restrictions. Any user can call this function.

Return Values

<table>
<thead>
<tr>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Success</td>
</tr>
<tr>
<td>-1</td>
<td>Failure</td>
</tr>
</tbody>
</table>

Error Codes

- **EINVAL**: Invalid argument. The flag parameter is not valid or the passed pointer is NULL.
- **ENOENT**: Project not found.

Related Information

The addproj Subroutine on page 31, chprojattr Subroutine on page 158, getprojdb Subroutine, getprojs Subroutine on page 415, rmproj Subroutine.

getprojdb Subroutine

Purpose
Retrieves the specified project record from the project database.

Library
The libaacct.a library.

Syntax

```c
#include <sys/aacct.h>

getprojdb(void *handle, struct project *project, int flag)
```

Description
The getprojdb subroutine searches the project database associated with the handle parameter for the specified project. The project database must be initialized before calling this subroutine. The routines projdballoc and projdbfinit are provided for this purpose. The flag parameter indicates the type of search. The following flags are defined:

- PROJ_NAME — Search by product name. The getprojdb subroutine scans the file to obtain a match for the supplied project name and returns the matching entry.
- PROJ_NUM — Search by product number. The getprojdb subroutine scans the file to obtain a match for the supplied project number and returns the matching entry.

The entire database is searched. If the specified record is found, the getprojdb subroutine stores the relevant project information into the struct project buffer, which is passed as an argument to this subroutine. The specified project is then made the current project in the database. If the specified project is not found, the database is reset so that the first project in the database is the current project.

Parameters

- **handle**: Pointer to the handle allocated for the project database.
- **project**: Pointer holding the project name whose information is required.
flag Integer flag indicating what type of information is sent for matching; that is, whether the match needs to be performed by project name or number.

Security
No restrictions. Any user can call this function.

Return Values

<table>
<thead>
<tr>
<th>Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Success</td>
</tr>
<tr>
<td>-1</td>
<td>Failure</td>
</tr>
</tbody>
</table>

Error Codes

<table>
<thead>
<tr>
<th>Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ENOENT</td>
<td>Project definition not found.</td>
</tr>
<tr>
<td>EINVAL</td>
<td>Invalid arguments if flag is not valid or passed pointer is NULL.</td>
</tr>
</tbody>
</table>

Related Information
The [addprojdb Subroutine](#) on page 32, [chprojattdb Subroutine](#) on page 159, [getfirstprojdb Subroutine](#) on page 363, [getnextprojdb Subroutine](#) on page 391, [getproj Subroutine](#) on page 413, [projdballoc Subroutine](#) on page 1158, [projdbfinit Subroutine](#) on page 1159, [projdbfree Subroutine](#) on page 1160, [rmprojdb Subroutine](#)

getprojs Subroutine

Purpose
Retrieves the project details from the kernel project registry.

Library
The `libaacct.a` library.

Syntax
```
<sys/aacct.h>

getprojs(struct project *, int *)
```

Description
The `getprojs` subroutine retrieves the specified number of project definitions from the kernel project registry. The number of definitions to be retrieved is passed as an argument to this subroutine, and it is also passed with a buffer of type `struct project`, where the retrieved project definitions are stored.

When the `getprojs` subroutine is called with a NULL value passed instead of a pointer to a `struct project`, the `getprojs` subroutine returns the total number of defined projects in the kernel project registry. This number can be used by any subsequent calls to retrieve the project details.

If the integer value passed is smaller than the number of project definitions available, then the project buffer will be filled with as many entries as requested. If the value is greater than the number of available definitions, then the available records are filled in the structure and the integer value is updated with the number of records actually retrieved.
Generally, the projects are loaded from the system project definition file or LDAP, or from both. When more than one of these project repositories are used, project name and project ID collisions are possible. These projects are differentiated by the kernel using an origin flag. This origin flag designates the project repository from where the project definition is obtained. Valid project origins values that can be passed are defined in the `sys/aacct.h` file. The `getproj` subroutine also returns this origin information in the `flags` field of the output project structures.

**Parameters**

- `pointer` Points to a project structure where the retrieved data is stored.
- `int` An integer that indicates the number of elements to be retrieved.

**Security**

There are no restrictions. Any user can call this function.

**Return Values**

- 0 Success
- -1 Failure

**Error Codes**

- `EINVAL` Invalid arguments if passed `int` pointer is NULL
- `ENOENT` No projects available.

**Related Information**

- The “addproj Subroutine” on page 31, “chprojattr Subroutine” on page 158, “getproj Subroutine” on page 413, “rmproj Subroutine”

---

### getpw Subroutine

**Purpose**

Retrieves a user’s `/etc/passwd` file entry.

**Library**

Standard C Library (`libc.a`)

**Syntax**

```c
int getpw (UserID, Buffer)

uid_t UserID
char *Buffer
```

**Description**

The `getpw` subroutine opens the `/etc/passwd` file and returns, in the `Buffer` parameter, the `/etc/passwd` file entry of the user specified by the `UserID` parameter.

**Parameters**

- `Buffer` Specifies a character buffer large enough to hold any `/etc/passwd` entry.
**UserID**

Specifies the ID of the user for which the entry is desired.

**Return Values**
The `getpw` subroutine returns:

<table>
<thead>
<tr>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Successful completion</td>
</tr>
<tr>
<td>-1</td>
<td>Not successful</td>
</tr>
</tbody>
</table>

**getpwent, getpwuid, getpwnam, putpwent, setpwent, or endpwent**

**Subroutine**

Accesses the basic user information in the user database.

**Library**

Standard C Library (`libc.a`)

**Syntax**

```c
#include <sys/types.h>
#include <pwd.h>
struct passwd *getpwent ()

struct passwd *getpwuid (UserID)
uid_t UserID;

struct passwd *getpwnam (Name)
char *Name;

int putpwent (Password, File)
struct passwd *Password;
FILE *File;

void setpwent ()
void endpwent ()
```

**Description**

**Attention:** All information generated by the `getpwent`, `getpwnam`, and `getpwuid` subroutines is stored in a static area. Subsequent calls to these subroutines overwrite this static area. To save the information in the static area, applications should copy it.

These subroutines access the basic user attributes.

The `setpwent` subroutine opens the user database if it is not already open. Then, this subroutine sets the cursor to point to the first user entry in the database. The `endpwent` subroutine closes the user database.

The `getpwent`, `getpwnam`, and `getpwuid` subroutines return information about a user. These subroutines do the following:

- `getpwent` Returns the next user entry in the sequential search.
- `getpwnam` Returns the first user entry in the database whose name matches the `Name` parameter.
- `getpwuid` Returns the first user entry in the database whose ID matches the `UserID` parameter.
The \texttt{putpwent} subroutine writes a password entry into a file in the colon-separated format of the \\
\texttt{/etc/passwd} file.

**The user Structure**

The \texttt{getpwent}, \texttt{getpwnam}, and \texttt{getpwuid} subroutines return a \texttt{user} structure. This structure The \texttt{user} structure is defined in the \texttt{pwd.h} file and has the following fields:

- \texttt{pw\_name}: Contains the name of the user name.
- \texttt{pw\_passwd}: Contains the user's encrypted password.\texttt{\textbf{Note: If the password is not stored in the /etc/passwd file and the invoker does not have access to the shadow file that contains passwords, this field contains an undecryptable string, usually an * (asterisk).}}
- \texttt{pw\_uid}: Contains the user's ID.
- \texttt{pw\_gid}: Identifies the user's principal group ID.
- \texttt{pw\_gecos}: Contains general user information.
- \texttt{pw\_dir}: Identifies the user's home directory.
- \texttt{pw\_shell}: Identifies the user's login shell.

**Note:** If Network Information Services (NIS) is enabled on the system, these subroutines attempt to retrieve the information from the NIS authentication server before attempting to retrieve the information locally.

**Parameters**

- **File**: Points to an open file whose format is similar to the \texttt{/etc/passwd} file format.
- **Name**: Specifies the user name.
- **Password**: Points to a password structure. This structure contains user attributes.
- **UserID**: Specifies the user ID.

**Security**

Files Accessed:

- **Mode**
  - \texttt{rw} \texttt{/etc/passwd} (write access for the \texttt{putpwent} subroutine only)
  - \texttt{r} \texttt{/etc/security/passwd} (if the password is desired)

**Return Values**

The \texttt{getpwent}, \texttt{getpwnam}, and \texttt{getpwuid} subroutines return a pointer to a valid password structure if successful. Otherwise, a null pointer is returned.

The \texttt{getpwent} subroutine will return a null pointer and an \texttt{errno} value of \texttt{ENOATTR} when it detects a corrupt entry. To get subsequent entries following the corrupt entry, call the \texttt{getpwent} subroutine again.

**Files**

- \texttt{/etc/passwd}: Contains user IDs and their passwords

**Related Information**

The \texttt{getgrent} subroutine, \texttt{getgroupttr} subroutine, \texttt{getgroupttr, IDtogroup, nextgroup, or putgroupttr Subroutine} on page 366, \texttt{setgrent}, \texttt{setgroupttr, IDtogroup, nextgroup, or putgroupttr Subroutine} on page 370.
getuserattr subroutine, getuserpw, putuserpw, or putuserpwhist subroutine, setuserdb subroutine.

List of Security and Auditing Subroutines, Subroutines, Example Programs, and Libraries in AIX 5L Version 5.3 General Programming Concepts: Writing and Debugging Programs.

getrlimit, getrlimit64, setrlimit, setrlimit64, or vlimit Subroutine

Purpose
Controls maximum system resource consumption.

Library
Standard C Library (libc.a)

Syntax
#include <sys/time.h>
#include <sys/resource.h>

int setrlimit(int Resource1, struct rlimit *RLP);
int getrlimit(int Resource1, struct rlimit *RLP);

int setrlimit64(int Resource1, struct rlimit64 *RLP);
int getrlimit64(int Resource1, struct rlimit64 *RLP);

int setrlimit64(int Resource1, struct rlimit64 *RLP);

#include <sys/vlimit.h>

vlimit(int Resource2, int Value);

Description
The getrlimit subroutine returns the values of limits on system resources used by the current process and its children processes. The setrlimit subroutine sets these limits. The vlimit subroutine is also supported, but the getrlimit subroutine replaces it.

A resource limit is specified as either a soft (current) or hard limit. A calling process can raise or lower its own soft limits, but it cannot raise its soft limits above its hard limits. A calling process must have root user authority to raise a hard limit.

Note: The initial values returned by the getrlimit subroutine are the ulimit values in effect when the process was started. For maxdata programs the initial soft limit for data is set to the lower of data ulimit value or a value corresponding to the number of data segments reserved for data segments. When a program is executing using the large address-space model, the operating system attempts...
to modify the soft limit on data size to match the *maxdata* value. If the *maxdata* value is larger than the current hard limit on data size, either the program will not execute if the XPG_SUS_ENV environment variable has the value set to ON, or the soft limit will be set to the current hard limit. If the *maxdata* value is smaller than the size of the program's static data, the program will not execute.

The *rlimit* structure specifies the hard and soft limits for a resource, as defined in the *sys/resource.h* file. The *RLIM_INFINITY* value defines an infinite value for a limit.

When compiled in 32-bit mode, *RLIM_INFINITY* is a 32-bit value; when compiled in 64-bit mode, it is a 64-bit value. 32-bit routines should use *RLIM64_INFINITY* when setting 64-bit limits with the *setrlimit64* routine, and recognize this value when returned by *getrlimit64*.

This information is stored as per-process information. This subroutine must be executed directly by the shell if it is to affect all future processes created by the shell.

**Note:** Raising the data limit does not raise the program break value. Use the *brk/sbrk* subroutines to raise the break value. If the proper memory segments are not initialized at program load time, raising your memory limit will not allow access to this memory. Use the `-bmaxdata` flag of the *ld* command to set up these segments at load time.

When compiled in 32-bit mode, the *struct rlimit* values may be returned as *RLIM_SAVED_MAX* or *RLIM_SAVED_CUR* when the actual resource limit is too large to represent as a 32-bit *rlim_t*.

These values can be used by library routines which set their own *rlimits* to save off potentially 64-bit *rlimit* values (and prevent them from being truncated by the 32-bit *struct rlimit*). Unless the library routine intends to permanently change the *rlimits*, the *RLIM_SAVED_MAX* and *RLIM_SAVED_CUR* values can be used to restore the 64-bit *rlimits*.

Application limits may be further constrained by available memory or implementation defined constants such as *OPEN_MAX* (maximum available open files).
Parameters

Resource1

Can be one of the following values:

**RLIMIT_AS**
The maximum size of a process’ total available memory, in bytes. This limit is not enforced.

**RLIMIT_CORE**
The largest size, in bytes, of a core file that can be created. This limit is enforced by the kernel. If the value of the **RLIMITFSIZE** limit is less than the value of the **RLIMITCORE** limit, the system uses the **RLIMITFSIZE** limit value as the soft limit.

**RLIMIT_CPU**
The maximum amount of central processing unit (CPU) time, in seconds, to be used by each process. If a process exceeds its soft CPU limit, the kernel will send a **SIGXCPU** signal to the process. After the hard limit is reached, the process will be killed with **SIGXCPU**, even if it handles, blocks, or ignores that signal.

**RLIMIT_DATA**
The maximum size, in bytes, of the data region for a process. This limit defines how far a program can extend its break value with the **sbrk** subroutine. This limit is enforced by the kernel. If the **XPG_SUS_ENV=ON** environment variable is set in the user’s environment before the process is executed and a process attempts to set the limit lower than current usage, the operation fails with **errno** set to **EINVAL**. If the **XPG_SUS_ENV** environment variable is not set, the operation fails with **errno** set to **EFAULT**.

**RLIMITFSIZE**
The largest size, in bytes, of any single file that can be created. When a process attempts to write, truncate, or clear beyond its soft **RLIMITFSIZE** limit, the operation will fail with **errno** set to **EFBIG**. If the environment variable **XPG_SUS_ENV=ON** is set in the user’s environment before the process is executed, then the **SIGXFSZ** signal is also generated.

**RLIMITNOFILE**
This is a number one greater than the maximum value that the system may assign to a newly-created descriptor.

**RLIMITSTACK**
The maximum size, in bytes, of the stack region for a process. This limit defines how far a program stack region can be extended. Stack extension is performed automatically by the system. This limit is enforced by the kernel. When the stack limit is reached, the process receives a **SIGSEGV** signal. If this signal is not caught by a handler using the signal stack, the signal ends the process.

**RLIMITRSS**
The maximum size, in bytes, to which the resident set size of a process can grow. This limit is not enforced by the kernel. A process may exceed its soft limit size without being ended.

**RLP**
Points to the **rlimit** or **rlimit64** structure, which contains the soft (current) and hard limits. For the **getrlimit** subroutine, the requested limits are returned in this structure. For the **setrlimit** subroutine, the desired new limits are specified here.

Resource2

The flags for this parameter are defined in the **sys/vlimit.h**, and are mapped to corresponding flags for the **setrlimit** subroutine.

Value
Specifies an integer used as a soft-limit parameter to the **vlimit** subroutine.

Return Values

On successful completion, a return value of 0 is returned, changing or returning the resource limit. Otherwise, a value of -1 is returned and the **errno** global variable is set to indicate the error. If the current limit specified is beyond the hard limit, the **setrlimit** subroutine sets the limit to to max limit and returns successfully.
Error Codes

The getrlimit, getrlimit64, setrlimit, setrlimit64, or vlimit subroutine is unsuccessful if one of the following is true:

- **EFAULT**
  - The address specified for the RLP parameter is not valid.

- **EINVAL**
  - The Resource1 parameter is not a valid resource, or the limit specified in the RLP parameter is invalid.

- **EPERM**
  - The limit specified to the setrlimit subroutine would have raised the maximum limit value, and the caller does not have root user authority.

Related Information

The sigaction, sigvec, or signal subroutines, sigstack subroutine, ulimit subroutine.

getrpcent, getrpcbyname, getrpcbynumber, setrpcent, or endrpcent Subroutine

Purpose

Accesses the /etc/rpc file.

Library

Standard C Library (libc.a)

Syntax

```c
#include <netdb.h>

struct rpcent *getrpcent ()
struct rpcent *getrpcbyname (Name)
char *Name;
struct rpcent *getrpcbynumber (Number)
int Number;
void setrpcent (StayOpen)
int StayOpen;
void endrpcent
```

Description

**Attention:** Do not use the getrpcent, getrpcbyname, getrpcbynumber, setrpcent, or endrpcent subroutine in a multithreaded environment.

**Attention:** The information returned by the getrpcbyname, and getrpcbynumber subroutines is stored in a static area and is overwritten on subsequent calls. Copy the information to save it.

The getrpcbyname and getrpcbynumber subroutines each return a pointer to an object with the rpcent structure. This structure contains the broken-out fields of a line from the /etc/rpc file. The getrpcbyname and getrpcbynumber subroutines searches the rpc file sequentially from the beginning of the file until it finds a matching RPC program name or number, or until it reaches the end of the file. The getrpcent subroutine reads the next line of the file, opening the file if necessary.

The setrpcent subroutine opens and rewinds the /etc/rpc file. If the StayOpen parameter does not equal 0, the rpc file is not closed after a call to the getrpcent subroutine.

The setrpcent subroutine rewinds the rpc file. The endrpcent subroutine closes it.
The `rpc` file contains information about Remote Procedure Call (RPC) programs. The `rpcent` structure is in the `/usr/include/netdb.h` file and contains the following fields:

- `r_name`: Contains the name of the server for an RPC program.
- `r_aliases`: Contains an alternate list of names for RPC programs. This list ends with a 0.
- `r_number`: Contains a number associated with an RPC program.

### Parameters

- **Name**: Specifies the name of a server for `rpc` program.
- **Number**: Specifies the `rpc` program number for service.
- **StayOpen**: Contains a value used to indicate whether to close the `rpc` file.

### Return Values

These subroutines return a null pointer when they encounter the end of a file or an error.

### Files

- `/etc/rpc`: Contains information about Remote Procedure Call (RPC) programs.

### Related Information

[Remote Procedure Call (RPC) for Programming in AIX 5L Version 5.3 General Programming Concepts: Writing and Debugging Programs](#)

### getrusage, getrusage64, times, or vtimes Subroutine

#### Purpose
Displays information about resource use.

#### Libraries

- **getrusage, getrusage64, times**: Standard C Library (`libc.a`)
- **vtimes**: Berkeley Compatibility Library (`libbsd.a`)

#### Syntax

```c
#include <sys/times.h>
#include <sys/resource.h>

int getrusage (Who, RUsage);
int Who;
struct rusage *RUsage;

int getrusage64 (Who, RUsage);
int Who;
struct rusage64 *RUsage;

#include <sys/types.h>
#include <sys/times.h>

clock_t times (Buffer);
struct tms *Buffer;
```
#include <sys/times.h>

vtimes (ParentVM, ChildVM)
struct vtimes *ParentVm, ChildVm;

Description
The getrusage subroutine displays information about how resources are used by the current process or all completed child processes.

When compiled in 64-bit mode, rusage counters are 64 bits. If getrusage is compiled in 32-bit mode, rusage counters are 32 bits. If the kernel's value of a usage counter has exceeded the capacity of the corresponding 32-bit rusage value being returned, the rusage value is set to INT_MAX.

The getrusage64 subroutine can be called to make 64-bit rusage counters explicitly available in a 32-bit environment.

In AIX 5.1 and later, 64-bit quantities are also available to 64-bit applications through the getrusage() interface in the ru_utime and ru_stime fields of struct rusage.

The times subroutine fills the structure pointed to by the Buffer parameter with time-accounting information. All time values reported by the times subroutine are measured in terms of the number of clock ticks used. Applications should use sysconf (_SC_CLK_TCK) to determine the number of clock ticks per second.

The tms structure defined in the /usr/include/sys/times.h file contains the following fields:

time_t tms_utime;
time_t tms_stime;
time_t tms_cutime;
time_t tms_cstime;

This information is read from the calling process as well as from each completed child process for which the calling process executed a wait subroutine.

tms_utime The CPU time used for executing instructions in the user space of the calling process

tms_stime The CPU time used by the system on behalf of the calling process.

tms_cutime The sum of the tms_utime and the tms_cutime values for all the child processes.

tms_cstime The sum of the tms_stime and the tms_cstime values for all the child processes.

Note: The system measures time by counting clock interrupts. The precision of the values reported by the times subroutine depends on the rate at which the clock interrupts occur.

The vtimes subroutine is supported to provide compatibility with earlier programs.

The vtimes subroutine returns accounting information for the current process and for the completed child processes of the current process. Either the ParentVm parameter, the ChildVm parameter, or both may be 0. In that case, only the information for the nonzero pointers is returned.

After a call to the vtimes subroutine, each buffer contains information as defined by the contents of the /usr/include/sys/vtimes.h file.

Parameters

Who Specifies a value of RUSAGE_THREAD, RUSAGE_SELF, or RUSAGE_CHILDREN.
**RUsage**
Points to a buffer described in the `/usr/include/sys/resource.h` file. The fields are interpreted as follows:

- **ru_utime**
  The total amount of time running in user mode.

- **ru_stime**
  The total amount of time spent in the system executing on behalf of the processes.

- **ru_maxrss**
  The maximum size, in kilobytes, of the used resident set size.

- **ru_ixrss**
  An integral value indicating the amount of memory used by the text segment that was also shared among other processes. This value is expressed in units of kilobytes * seconds-of-execution and is calculated by adding the number of shared memory pages in use each time the internal system clock ticks, and then averaging over one-second intervals.

- **ru_idrss**
  An integral value of the amount of unshared memory in the data segment of a process (expressed in units of kilobytes * seconds-of-execution).

- **ru_minflt**
  The number of page faults serviced without any I/O activity. In this case, I/O activity is avoided by reclaiming a page frame from the list of pages awaiting reallocation.

- **ru_majflt**
  The number of page faults serviced that required I/O activity.

- **ru_nswap**
  The number of times a process was swapped out of main memory.

- **ru_inblock**
  The number of times the file system performed input.

- **ru_oublock**
  The number of times the file system performed output.  
  **Note:** The numbers that the ru_inblock and ru_oublock fields display account for real I/O only; data supplied by the caching mechanism is charged only to the first process to read or write the data.

- **ru_msgsnd**
  The number of IPC messages sent.

- **ru_msgrcv**
  The number of IPC messages received.

- **ru_nsignals**
  The number of signals delivered.

- **ru_nvcsw**
  The number of times a context switch resulted because a process voluntarily gave up the processor before its time slice was completed. This usually occurs while the process waits for availability of a resource.

- **ru_nivcsw**
  The number of times a context switch resulted because a higher priority process ran or because the current process exceeded its time slice.

**Buffer**
Points to a `tms` structure.

**ParentVm**
Points to a `vtimes` structure that contains the accounting information for the current process.

**ChildVm**
Points to a `vtimes` structure that contains the accounting information for the terminated child processes of the current process.
Return Values
Upon successful completion, the `getrusage` and `getrusage64` subroutines return a value of 0. Otherwise, a value of -1 is returned and the `errno` global variable is set to indicate the error.

Upon successful completion, the `times` subroutine returns the elapsed real time in units of ticks, whether profiling is enabled or disabled. This reference time does not change from one call of the `times` subroutine to another. If the `times` subroutine fails, it returns a value of -1 and sets the `errno` global variable to indicate the error.

Error Codes
The `getrusage` and `getrusage64` subroutines do not run successfully if either of the following is true:
- **EINVAL**: The `Who` parameter is not a valid value.
- **EFAULT**: The address specified for `RUsage` is not valid.

The `times` subroutine does not run successfully if the following is true:
- **EFAULT**: The address specified by the `buffer` parameter is not valid.

Error Codes

Related Information
The `gettimer, settimer, restimer, stime, or time` subroutine, `wait, waitpid, or wait3` subroutine.

**getroleattr, nextrole or putroleattr Subroutine**

Purpose
Accesses the role information in the roles database.

Library
Security Library (libc.a)

Syntax
```c
#include <usersec.h>

int getroleattr(Role, Attribute, Value, Type);
char *Role;
char *Attribute;
void *Value;
int Type;
char *nextrole(void)

int putroleattr(Role, Attribute, Value, Type)
char *Role;
char *Attribute;
void *Value;
int Type;
```

Description
The `getroleattr` subroutine reads a specified attribute from the role database. If the database is not already open, this subroutine does an implicit open for reading.

Similarly, the `putroleattr` subroutine writes a specified attribute into the role database. If the database is not already open, this subroutine does an implicit open for reading and writing. Data changed by the
**putroleattr** subroutine must be explicitly committed by calling the **putroleattr** subroutine with a Type parameter specifying SEC_COMMIT. Until all the data is committed, only the **getroleattr** subroutine within the process returns written data.

The **nextrole** subroutine returns the next role in a linear search of the role database. The consistency of consecutive searches depends upon the underlying storage-access mechanism and is not guaranteed by this subroutine.

The **setroledb** and **endroledb** subroutines should be used to open and close the role database.

**Parameters**

*Attribute* Specifies which attribute is read. The following possible attributes are defined in the **usersec.h** file:

- **S_ROLELIST**
  List of roles included by this role. The attribute type is **SEC_LIST**.

- **S_AUTHORIZATIONS**
  List of authorizations included by this role. The attribute type is **SEC_LIST**.

- **S_GROUPS**
  List of groups required for this role. The attribute type is **SEC_LIST**.

- **S_SCREENS**
  List of SMIT screens required for this role. The attribute type is **SEC_LIST**.

- **S_VISIBILITY**
  Number value stating the visibility of the role. The attribute type is **SEC_INT**.

- **S_MSGCAT**
  Message catalog file name. The attribute type is **SEC_CHAR**.

- **S_MSGNUMBER**
  Message number within the catalog. The attribute type is **SEC_INT**.
**Type** Specifies the type of attribute expected. Valid types are defined in the `usersec.h` file and include:

- **SEC_INT**
  The format of the attribute is an integer.
  For the `getroleattr` subroutine, the user should supply a pointer to a defined integer variable.
  For the `putroleattr` subroutine, the user should supply an integer.

- **SEC_CHAR**
  The format of the attribute is a null-terminated character string.
  For the `getroleattr` subroutine, the user should supply a pointer to a defined character pointer variable.
  For the `putroleattr` subroutine, the user should supply a character pointer.

- **SEC_LIST**
  The format of the attribute is a series of concatenated strings, each null-terminated. The last string in the series must be an empty (zero character count) string.
  For the `getroleattr` subroutine, the user should supply a pointer to a defined character pointer variable.
  For the `putroleattr` subroutine, the user should supply a character pointer.

- **SEC_COMMIT**
  For the `putroleattr` subroutine, this value specified by itself indicates that changes to the named role are to be committed to permanent storage. The `Attribute` and `Value` parameters are ignored. If no role is specified, the changes to all modified roles are committed to permanent storage.

- **SEC_DELETE**
  The corresponding attribute is deleted from the database.

- **SEC_NEW**
  Updates the role database file with the new role name when using the `putroleattr` subroutine.

**Value** Specifies a buffer, a pointer to a buffer, or a pointer to a pointer depending on the `Attribute` and `Type` parameters. See the `Type` parameter for more details.

**Return Values**
If successful, the `getroleattr` returns 0. Otherwise, a value of -1 is returned and the `errno` global variables is set to indicate the error.

**Error Codes**
Possible return codes are:

- **EACCES** Access permission is denied for the data request.
- **ENOENT** The specified `Role` parameter does not exist.
- **ENOATTR** The specified role attribute does not exist for this role.
- **EINVAL** The `Attribute` parameter does not contain one of the defined attributes or null.
- **EINVAL** The `Value` parameter does not point to a valid buffer or to valid data for this type of attribute.
- **EPERM** Operation is not permitted.

**Related Information**
The `getuserattr`, `nextusracl`, or `putusraclattr` ("getuserattr, IDtouser, nextuser, or putuserattr Subroutine" on page 449) subroutine, `setroledb`, or `endacldb` subroutine.
gets or fgets Subroutine

Purpose
Gets a string from a stream.

Library
Standard I/O Library (libc.a)

Syntax
```c
#include <stdio.h>
char *gets (String);
char *String;

char *fgets (String, Number, Stream);
char *String;
int Number;
FILE *Stream;
```

Description
The `gets` subroutine reads bytes from the standard input stream, `stdin`, into the array pointed to by the `String` parameter. It reads data until it reaches a new-line character or an end-of-file condition. If a new-line character stops the reading process, the `gets` subroutine discards the new-line character and terminates the string with a null character.

The `fgets` subroutine reads bytes from the data pointed to by the `Stream` parameter into the array pointed to by the `String` parameter. The `fgets` subroutine reads data up to the number of bytes specified by the `Number` parameter minus 1, or until it reads a new-line character and transfers that character to the `String` parameter, or until it encounters an end-of-file condition. The `fgets` subroutine then terminates the data string with a null character.

The first successful run of the `fgetc` ("getc, getchar, fgetc, or getw Subroutine" on page 343), `fgets`, `fgetwc` ("getwc, fgetwc, or getwchar Subroutine" on page 472), `fgetws` ("getws or fgetws Subroutine" on page 475), `fread` ("fread or fwrite Subroutine" on page 307), `fscanf` `getc` ("getc, getchar, fgetc, or getw Subroutine" on page 343), `getchar` ("getc, getchar, fgetc, or getw Subroutine" on page 343), `gets` or `scanf` subroutine using a stream that returns data not supplied by a prior call to the `ungetc` or `ungetwc` subroutine marks the `st_atime` field for update.

Parameters
- `String` Points to a string to receive bytes.
- `Stream` Points to the `FILE` structure of an open file.
- `Number` Specifies the upper bound on the number of bytes to read.

Return Values
If the `gets` or `fgets` subroutine encounters the end of the file without reading any bytes, it transfers no bytes to the `String` parameter and returns a null pointer. If a read error occurs, the `gets` or `fgets` subroutine returns a null pointer and sets the `errno` global variable (errors are the same as for the `fgetc` ("getc, getchar, fgetc, or getw Subroutine" on page 343) subroutine). Otherwise, the `gets` or `fgets` subroutine returns the value of the `String` parameter.

Note: Depending upon which library routine the application binds to, this subroutine may return `EINTR`. Refer to the `signal` subroutine regarding the `SA_RESTART` value.
Related Information

The feof, ferror, clearerr, or fileno macro, fopen, freopen, or fdopen subroutine, fread subroutine, getc, getchar, fgetc, or getw subroutine, getwc, fgetwc, or getwchar subroutine, putw or fputw subroutine, getfsent_r, getfsspec_r, getfsfile_r, getfstype_r, setfsent_r, or endfsent_r subroutine.

List of String Manipulation Services Subroutines Overview in AIX 5L Version 5.3 General Programming Concepts: Writing and Debugging Programs.

getfsent_r, getfsspec_r, getfsfile_r, getfstype_r, setfsent_r, or endfsent_r Subroutine

Purpose

Gets information about a file system.

Library

Thread-Safe C Library (libc_r.a)

Syntax

```c
#include <fstab.h>

int getfsent_r (FSSent, FSFile, PassNo)
struct fstab *FSSent;
AFILE_t *FSFile;
int *PassNo;

int getfsspec_r (Special, FSSent, FSFile, PassNo)
const char *Special;
struct fstab *FSSent;
AFILE_t *FSFile;
int *PassNo;

int getfsfile_r (File, FSSent, FSFile, PassNo)
const char *File;
struct fstab *FSSent;
AFILE_t *FSFile;
int *PassNo;

int getfstype_r (Type, FSSent, FSFile, PassNo)
const char *Type;
struct fstab *FSSent;
AFILE_t *FSFile;
int *PassNo;

int setfsent_r (FSFile, PassNo)
AFILE_t *FSFile;
int *PassNo;

int endfsent_r (FSFile)
AFILE_t *FSFile;
```
Description

The `getfsent_r` subroutine reads the next line of the `/etc/filesystems` file, opening it necessary.

The `setfsent_r` subroutine opens the `filesystems` file and positions to the first record.

The `endfsent_r` subroutine closes the `filesystems` file.

The `getfsspec_r` and `getfsfile_r` subroutines search sequentially from the beginning of the file until a matching special file name or file-system file name is found, or until the end of the file is encountered. The `getfstype_r` subroutine behaves similarly, matching on the file-system type field.

Programs using this subroutine must link to the `libpthread.a` library.

Parameters

- **FSSent**: Points to a structure containing information about the file system. The `FSSent` parameter must be allocated by the caller. It cannot be a null value.
- **FSFile**: Points to an attribute structure. The `FSFile` parameter is used to pass values between subroutines.
- **PassNo**: Points to an integer. The `setfsent_r` subroutine initializes the `PassNo` parameter.
- **Special**: Specifies a special file name to search for in the `filesystems` file.
- **File**: Specifies a file name to search for in the `filesystems` file.
- **Type**: Specifies a type to search for in the `filesystems` file.

Return Values

- 0: Indicates that the subroutine was successful.
- -1: Indicates that the subroutine was not successful.

Files

`/etc/filesystems` Centralizes file-system characteristic.

Related Information

The `getvfsent`, `getvfsbytype`, `getvfsbyname`, `getvfsbyflag`, `setvfsent`, or `endvfsent` subroutine.

The `filesystems` file in `AIX 5L Version 5.3 Files Reference`.

List of Multithread Subroutines in `AIX 5L Version 5.3 General Programming Concepts: Writing and Debugging Programs`.

**getsid Subroutine**

Purpose

Returns the session ID of the calling process.

Library

(libc.a)
Syntax
#include <unistd.h>

pid_t getsid (pid_t pid)

Description
The getsid subroutine returns the process group ID of the process that is the session leader of the process specified by pid. If pid is equal to pid_t subroutine, it specifies the calling process.

Parameters
pid
A process ID of the process being queried.

Return Values
Upon successful completion, getsid subroutine returns the process group ID of the session leader of the specified process. Otherwise, it returns (pid_t)-1 and set errno to indicate the error.

id
The session ID of the requested process.
-1
Not successful and the errno global variable is set to one of the following error codes.

Error Codes
ESRCH
There is no process with a process ID equal to pid.

EPERM
The process specified by pid is not in the same session as the calling process.
ESRCH
There is no process with a process ID equal to pid.

Related Information
The exec subroutine, getsid subroutine, fork subroutine, getpid subroutine, setpgid subroutine

getssys Subroutine

Purpose
Reads a subsystem record.

Library
System Resource Controller Library (libsrc.a)

Syntax
#include <sys/srcobj.h>
#include <spc.h>

int getssys (SubsystemName, SRCSubsystem)
char *SubsystemName;
struct SRCsubsys *SRCSubsystem;
Description
The \texttt{getssys} subroutine reads a subsystem record associated with the specified subsystem and returns the ODM record in the \texttt{SRCsubsys} structure.

The \texttt{SRCsubsys} structure is defined in the \texttt{sys/srcobj.h} file.

Parameters
\begin{itemize}
\item \texttt{SRCSubsystem} \quad Points to the \texttt{SRCsubsys} structure.
\item \texttt{SubsystemName} \quad Specifies the name of the subsystem to be read.
\end{itemize}

Return Values
Upon successful completion, the \texttt{getssys} subroutine returns a value of 0. Otherwise, it returns a value of -1 and the \texttt{odmerrno} variable is set to indicate the error, or an SRC error code is returned.

Error Codes
If the \texttt{getssys} subroutine fails, the following is returned:
\begin{itemize}
\item \texttt{SRC_NOREC} \quad Subsystem name does not exist.
\end{itemize}

Files
\begin{itemize}
\item /etc/objrepos/SRCsubsys \quad SRC Subsystem Configuration object class.
\end{itemize}

Related Information
The \texttt{addssys} \texttt{("addssys Subroutine" on page 33)} subroutine, \texttt{delssys} \texttt{("delssys Subroutine" on page 211)} subroutine, \texttt{getsubsvr} \texttt{("getsubsvr Subroutine" on page 434)} subroutine.

\texttt{Defining Your Subsystem to the SRC} | \texttt{List of SRC Subroutines} | \texttt{System Resource Controller (SRC)}
\texttt{Overview for Programmers in AIX 5L Version 5.3} | \texttt{General Programming Concepts: Writing and Debugging Programs}.

getsubopt Subroutine

Purpose
Parse suboptions from a string.

Library
Standard C Library (\texttt{libc.a})

Syntax
\begin{verbatim}
#include <stdlib.h>

int getsubopt (char **optionp,
               char * const * tokens,
               char ** valuep)
\end{verbatim}

Description
The \texttt{getsubopt} subroutine parses suboptions in a flag parameter that were initially parsed by the \texttt{getopt} subroutine. These suboptions are separated by commas and may consist of either a single token, or a
token-value pair separated by an equal sign. Because commas delimit suboptions in the option string, they are not allowed to be part of the suboption or the value of a suboption. Similarly, because the equal sign separates a token from its value, a token must not contain an equal sign.

The `getsubopt` subroutine takes the address of a pointer to the option string, a vector of possible tokens, and the address of a value string pointer. It returns the index of the token that matched the suboption in the input string or -1 if there was no match. If the option string at `*optionp` contains only one suboption, the `getsubopt` subroutine updates `*optionp` to point to the start of the next suboption. If the suboption has an associated value, the `getsubopt` subroutine updates `*valuep` to point to the value's first character. Otherwise it sets `*valuep` to a NULL pointer.

The token vector is organized as a series of pointers to strings. The end of the token vector is identified by a NULL pointer.

When the `getsubopt` subroutine returns, if `*valuep` is not a NULL pointer then the suboption processed included a value. The calling program may use this information to determine if the presence or lack of a value for this suboption is an error.

Additionally, when the `getsubopt` subroutine fails to match the suboption with the tokens in the `tokens` array, the calling program should decide if this is an error, or if the unrecognized option should be passed on to another program.

Return Values

The `getsubopt` subroutine returns the index of the matched token string, or -1 if no token strings were matched.

Related Information

The `getopt` subroutine.

---

**getsubsvr Subroutine**

**Purpose**

Reads a subsystem record.

**Library**

System Resource Controller Library (`libsrc.a`)

**Syntax**

```c
#include <sys/srcobj.h>
#include <spc.h>

int getsubsvr(SubserverName, SRCSubserver);
char *SubserverName;
struct SRCSubsvr *SRCSubserver;
```

**Description**

The `getsubsvr` subroutine reads a subsystem record associated with the specified subserver and returns the ODM record in the `SRCsubsvr` structure.

The `SRCsubsvr` structure is defined in the `sys/srcobj.h` file and includes the following fields:

```c
char sub_type[30];
```
Parameters

SRCSubserver Points to the SRCsubsvr structure.

SubserverName Specifies the subserver to be read.

Return Values

Upon successful completion, the getsubsvr subroutine returns a value of 0. Otherwise, it returns a value of -1 and the odmerrno variable is set to indicate the error, or an SRC error code is returned.

Error Codes

If the getsubsvr subroutine fails, the following is returned:

SRC_NOREC The specified SRCsubsvr record does not exist.

Files

/etc/objrepos/SRCsubsvr SRC Subserver Configuration object class.

Related Information

The getsys subroutine. "getsys Subroutine" on page 432

Defining Your Subsystem to the SRC List of SRC Subroutines System Resource Controller (SRC) Overview for Programmers in AIX 5L Version 5.3 General Programming Concepts: Writing and Debugging Programs.

getchbattr or putchbattr Subroutine

Purpose

Accesses the TCB information in the user database.

Library

Security Library (libc.a)

Syntax

#include <usersec.h>

int getchbattr (Entry, Attribute, Value, Type)
char *Entry;
char *Attribute;
void *Value;
int Type;

int putchbattr (Entry, Attribute, Value, Type)
char *Entry;
char *Attribute;
void *Value;
int Type;
Description
These subroutines access Trusted Computing Base (TCB) information.

The `gettcbattr` subroutine reads a specified attribute from the `tcbck` database. If the database is not already open, the subroutine will do an implicit open for reading.

Similarly, the `puttcbattr` subroutine writes a specified attribute into the `tcbck` database. If the database is not already open, the subroutine does an implicit open for reading and writing. Data changed by `puttcbattr` must be explicitly committed by calling the `puttcbattr` subroutine with a `Type` parameter specifying the SEC_COMMIT value. Until the data is committed, only `get` subroutine calls within the process will return the written data.

New entries in the `tcbck` databases must first be created by invoking `puttcbattr` with the SEC_NEW type.

The `tcbck` database usually defines all the files and programs that are part of the TCB, but the root user or a member of the security group can choose to define only those files considered to be security-relevant.

Parameters

`Attribute` Specifies which attribute is read. The following possible values are defined in the `sysck.h` file:

- **S_ACL** The access control list for the file. Type: SEC_CHAR.
- **S_CHECKSUM** The checksum of the file. Type: SEC_CHAR.
- **S_CLASS** The logical group of the file. The attribute type is SEC_LIST.
- **S_GROUP** The file group. The attribute type is SEC_CHAR.
- **S_LINKS** The hard links to this file. Type: SEC_LIST.
- **S_MODE** The File mode. Type: SEC_LIST.
- **S_OWNER** The file owner. Type: SEC_CHAR.
- **S_PROGRAM** The associated checking program for the file. Type: SEC_CHAR.
- **S_SIZE** The size of the file in bytes. Type: SEC_LONG.
- **S_SOURCE** The source for the file. Type: SEC_CHAR.
- **S_SYMLINKS** The symbolic links to the file. Type: SEC_LIST.
- **S_TARGET** The target file (if file is a symbolic link). Type: SEC_CHAR.
- **S_TCB** The Trusted Computer Base. The attribute type is SEC_BOOL.
- **S_TYPE** The type of file. The attribute type is SEC_CHAR.

Additional user-defined attributes may be used and will be stored in the format specified by the `Type` parameter.

`Entry` Specifies the name of the file for which an attribute is to be read or written.
**Type**  
Specifies the type of attribute expected. Valid values are defined in the `usersec.h` file and include:

- **SEC_BOOL**
  A pointer to an integer (`int *`) that has been cast to a null pointer.

- **SEC_CHAR**
  The format of the attribute is a null-terminated character string.

- **SEC_LIST**
  The format of the attribute is a series of concatenated strings, each null-terminated.  
  The last string in the series is terminated by two successive null characters.

- **SEC_LONG**
  The format of the attribute is a 32-bit integer.

**Value**  
Specifies the address of a pointer for the `gettcbattr` subroutine. The `gettcbattr` subroutine will return the address of a buffer in the pointer. For the `puttcbattr` subroutine, the `Value` parameter specifies the address of a buffer in which the attribute is stored. See the `Type` parameter for more details.

**Security**

Files Accessed:

<table>
<thead>
<tr>
<th>Mode</th>
<th>File</th>
</tr>
</thead>
<tbody>
<tr>
<td>rw</td>
<td>/etc/security/sysck.cfg (write access for <code>puttcbattr</code>)</td>
</tr>
</tbody>
</table>

**Return Values**

The `gettcbattr` and `puttcbattr` subroutines, when successfully completed, return a value of 0. Otherwise, a value of -1 is returned and the `errno` global variable is set to indicate the error.

**Error Codes**

*Note:* These subroutines return errors from other subroutines.

These subroutines fail if the following is true:

- **EACCESS**
  Access permission is denied for the data request.

The `gettcbattr` and `puttcbattr` subroutines fail if one or more of the following are true:

- **EINVAL**
  The `Value` parameter does not point to a valid buffer or to valid data for this type of attribute.  
  Limited testing is possible and all errors may not be detected.

- **EINVAL**
  The `Entry` parameter is null or contains a pointer to a null string.

- **EINVAL**
  The `Type` parameter contains more than one of the **SEC_BOOL**, **SEC_CHAR**, **SEC_LIST**, or **SEC_LONG** attributes.

- **EINVAL**
  The `Type` parameter specifies that an individual attribute is to be committed, and the `Entry` parameter is null.

- **ENOENT**
  The specified `Entry` parameter does not exist or the attribute is not defined for this entry.

- **EPERM**
  Operation is not permitted.
Related Information

The `getuserattr` subroutine, `getuserpw`, `putuserpw`, or `putuserpwhist` subroutine, `setpwdb`, or `setuserdb` subroutine.

List of Security and Auditing Subroutines and Subroutines Overview in AIX 5L Version 5.3 General Programming Concepts: Writing and Debugging Programs.

---

getthrds Subroutine

**Purpose**

Gets kernel thread table entries.

**Library**

Standard C library (`libc.a`)

**Syntax**

```c
#include <procinfo.h>
#include <sys/types.h>

int getthrds (ProcessIdentifier, ThreadBuffer, ThreadSize, IndexPointer, Count);

pid_t ProcessIdentifier;
struct thrdsinfo *ThreadBuffer;
or struct thrdsinfo64 *ThreadBuffer;
int ThreadSize;
tid_t *IndexPointer;
int Count;

int getthrds64 (ProcessIdentifier, ThreadBuffer, ThreadSize, IndexPointer, Count);

pid_t ProcessIdentifier;
struct thrdentry64 *ThreadBuffer;
int ThreadSize;
tid64_t *IndexPointer;
int Count;
```

**Description**

The `getthrds` subroutine returns information about kernel threads, including kernel thread table information defined by the `thrdsinfo` or `thrdsinfo64` structure.

The `getthrds` subroutine retrieves up to `Count` kernel thread table entries, starting with the entry corresponding to the thread identifier indicated by `IndexPointer`, and places them in the array of `thrdsinfo` or `thrdsinfo64`, or `thrdentry64` structures indicated by the `ThreadBuffer` parameter.

On return, the kernel thread identifier referenced by `IndexPointer` is updated to indicate the next kernel thread table entry to be retrieved. The `getthrds` subroutine returns the number of kernel thread table entries retrieved.

If the `ProcessIdentifier` parameter indicates a process identifier, only kernel threads belonging to that process are considered. If this parameter is set to -1, all kernel threads are considered.
The getthrds subroutine is normally called repeatedly in a loop, starting with a kernel thread identifier of zero, and looping until the return value is less than Count, indicating that there are no more entries to retrieve.

1. Do not use information from the procsinfo structure (see the getprocs subroutine) to determine the value of the Count parameter; a process may create or destroy kernel threads in the interval between a call to getprocs and a subsequent call to getthrds.

2. The kernel thread table may change while the getthrds subroutine is accessing it. Returned entries will always be consistent, but since kernel threads can be created or destroyed while the getthrds subroutine is running, there is no guarantee that retrieved entries will still exist, or that all existing kernel threads have been retrieved.

When used in 32-bit mode, limits larger than can be represented in 32 bits are truncated to RLIM_INFINITY. Large values are truncated to INT_MAX. 64-bit applications are required to use getthrds64() and struct thrdentry64. Note that struct thrdentry64 contains the same information as struct thrsinfo64 with the only difference being support for the 64-bit tid_t and the 256-bit sigset_t. Application developers are also encouraged to use getthrds64() in 32-bit applications to obtain 64-bit thread information as this interface provides the new, larger types. The getthrds() interface will still be supported for 32-bit applications using struct thrsinfo or struct thrsinfo64, but will not be available to 64-bit applications.

Parameters

ProcessIdentifier
Specifies the process identifier of the process whose kernel threads are to be retrieved. If this parameter is set to -1, all kernel threads in the kernel thread table are retrieved.

ThreadBuffer
Specifies the starting address of an array of thrsinfo, thrsinfo64, or thrdentry64 structures which will be filled in with kernel thread table entries. If a value of NULL is passed for this parameter, the getthrds subroutine scans the kernel thread table and sets return values as normal, but no kernel thread table entries are retrieved.

ThreadSize
Specifies the size of a single thrsinfo, thrsinfo64, or thrdentry64 structure.

IndexPointer
Specifies the address of a kernel thread identifier which indicates the required kernel thread table entry (this does not have to correspond to an existing kernel thread). A kernel thread identifier of zero selects the first entry in the table. The kernel thread identifier is updated to indicate the next entry to be retrieved.

Count
Specifies the number of kernel thread table entries requested.

Return Value
If successful, the getthrds subroutine returns the number of kernel thread table entries retrieved; if this is less than the number requested, the end of the kernel thread table has been reached. A value of 0 is returned when the end of the kernel thread table has been reached. Otherwise, a value of -1 is returned, and the errno global variable is set to indicate the error.

Error Codes
The getthrds subroutine fails if the following are true:

EINVAL
The ThreadSize is invalid, or the IndexPointer parameter does not point to a valid kernel thread identifier, or the Count parameter is not greater than zero.

ESRCH
The process specified by the ProcessIdentifier parameter does not exist.

EFAULT
The copy operation to one of the buffers failed.
Related Information

The `getpid` ("getpid, getpgrp, or getppid Subroutine" on page 402), `getpgrp` ("getpid, getpgrp, or getppid Subroutine" on page 402), or `getppid` ("getpid, getpgrp, or getppid Subroutine" on page 402) subroutines, the `getprocs` ("getprocs Subroutine" on page 410) subroutine.

The `ps` command.

gettimeofday, settimeofday, or ftime Subroutine

Purpose

Displays, gets and sets date and time.

Libraries

`gettimeofday`, `settimeofday`: Standard C Library (libc.a)

`ftime`: Berkeley Compatibility Library (libbsd.a)

Syntax

```c
#include <sys/time.h>
int gettimeofday (Tp, Tzp)
struct timeval *Tp;
void *Tzp;

int settimeofday (Tp, Tzp)
struct timeval *Tp;
struct timezone *Tzp;
```

Description

Current Greenwich time and the current time zone are displayed with the `gettimeofday` subroutine, and set with the `settimeofday` subroutine. The time is expressed in seconds and microseconds since midnight (0 hour), January 1, 1970. The resolution of the system clock is hardware-dependent, and the time may be updated either continuously or in “ticks.” If the `Tzp` parameter has a value of 0, the time zone information is not returned or set.

If a recent `adjtime` subroutine call is causing the clock to be adjusted backwards, it is possible that two closely spaced `gettimeofday` calls will observe that time has moved backwards. You can set the `GETTOD_ADJ_MONOTONIC` environment value to cause the returned value to never decrease. After this environment variable is set, the returned value briefly remains constant as necessary to always report a nondecreasing time of day. This extra processing adds significant pathlength to `gettimeofday`. Although any setting of this environment variable requires this extra processing, setting it to 1 is recommended for future compatibility.

The `Tp` parameter returns a pointer to a `timeval` structure that contains the time since the epoch began in seconds and microseconds.

The `timezone` structure indicates both the local time zone (measured in minutes of time westward from Greenwich) and a flag that, if nonzero, indicates that daylight saving time applies locally during the appropriate part of the year.
In addition to the difference in timer granularity, the timezone structure distinguishes these subroutines from the POSIX gettimer and settimer subroutines, which deal strictly with Greenwich Mean Time.

The ftime subroutine fills in a structure pointed to by its argument, as defined by \textless sys/timeb.h\textgreater. The structure contains the time in seconds since 00:00:00 UTC (Coordinated Universal Time), January 1, 1970, up to 1000 milliseconds of more-precise interval, the local timezone (measured in minutes of time westward from UTC), and a flag that, if nonzero, indicates that Daylight Saving time is in effect, and the values stored in the timeb structure have been adjusted accordingly.

**Parameters**

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>(Tp)</td>
<td>Pointer to a timeval structure, defined in the sys/time.h file.</td>
</tr>
<tr>
<td>(Tzp)</td>
<td>Pointer to a timezone structure, defined in the sys/time.h file.</td>
</tr>
</tbody>
</table>

**Return Values**

If the subroutine succeeds, a value of 0 is returned. If an error occurs, a value of -1 is returned and errno is set to indicate the error.

**Error Codes**

If the settimeofday subroutine is unsuccessful, the errno value is set to EPERM to indicate that the process’s effective user ID does not have root user authority.

No errors are defined for the gettimeofday or ftime subroutine.

---

**gettimer, settimer, restimer, stime, or time Subroutine**

**Purpose**

Gets or sets the current value for the specified systemwide timer.

**Library**

Standard C Library (libc.a)

**Syntax**

```c
#include <sys/time.h>
#include <sys/types.h>

int gettimer(TimerType, Value)
timer_t TimerType;
struct timestruc_t * Value;
#include <sys/timers.h>
#include <sys/types.h>

int gettimer(TimerType, Value)
timer_t TimerType;
struct itimerspec * Value;

int settimer(TimerType, TimePointer)
int TimerType;
const struct timestruc_t *TimePointer;
```
int restimer(TimerType, Resolution, MaximumValue);
int TimerType;
struct timestruc_t *Resolution, *MaximumValue;

int stime(Tp)
long *Tp;
#include <sys/types.h>
time_t time(Tp)
time_t *Tp;

Description
The settimer subroutine is used to set the current value of the TimePointer parameter for the systemwide timer, specified by the TimerType parameter.

When the gettimer subroutine is used with the function prototype in sys/timers.h, then except for the parameters, the gettimer subroutine is identical to the getinterval subroutine. Use of the getinterval subroutine is recommended, unless the gettimer subroutine is required for a standards-conformant application. The description and semantics of the gettimer subroutine are subject to change between releases, pending changes in the draft standard upon which the current gettimer subroutine description is based.

When the gettimer subroutine is used with the function prototype in /sys/timers.h, the gettimer subroutine returns an itimerspec structure to the pointer specified by the Value parameter. The it_value member of the itimerspec structure represents the amount of time in the current interval before the timer expires, or a zero interval if the timer is disabled. The members of the pointer specified by the Value parameter are subject to the resolution of the timer.

When the gettimer subroutine is used with the function prototype in sys/time.h, the gettimer subroutine returns a timestruc structure to the pointer specified by the Value parameter. This structure holds the current value of the system wide timer specified by the Value parameter.

The resolution of any timer can be obtained by the restimer subroutine. The Resolution parameter represents the resolution of the specified timer. The MaximumValue parameter represents the maximum possible timer value. The value of these parameters are the resolution accepted by the settimer subroutine.

Note: If a nonprivileged user attempts to submit a fine granularity timer (that is, a timer request of less than 10 milliseconds), the timer request is raised to 10 milliseconds.

The time subroutine returns the time in seconds since the Epoch (that is, 00:00:00 GMT, January 1, 1970). The Tp parameter points to an area where the return value is also stored. If the Tp parameter is a null pointer, no value is stored.

The stime subroutine is implemented to provide compatibility with older AIX, AT&T System V, and BSD systems. It calls the settimer subroutine using the TIMEOFDAY timer.

Parameters
Value Points to a structure of type itimerspec.
TimerType

Specifies the systemwide timer:

**TIMEOFDAY**

(POSIX system clock timer) This timer represents the time-of-day clock for the system. For this timer, the values returned by the `gettimer` subroutine and specified by the `settimer` subroutine represent the amount of time since 00:00:00 GMT, January 1, 1970.

TimePointer

Points to a structure of type `struct timestruc_t`.

Resolution

The resolution of a specified timer.

MaximumValue

The maximum possible timer value.

Tp

Points to a structure containing the time in seconds.

**Return Values**

The `gettimer`, `settimer`, `restimer`, and `stime` subroutines return a value of 0 (zero) if the call is successful. A return value of -1 indicates an error occurred, and `errno` is set.

The `time` subroutine returns the value of time in seconds since Epoch. Otherwise, a value of `((time_t) - 1)` is returned and the `errno` global variable is set to indicate the error.

**Error Codes**

If an error occurs in the `gettimer`, `settimer`, `restimer`, or `stime` subroutine, a return value of -1 is received and the `errno` global variable is set to one of the following error codes:

- **EINVAL**
  The `TimerType` parameter does not specify a known systemwide timer, or the `TimePointer` parameter of the `settimer` subroutine is outside the range for the specified systemwide timer.

- **EFAULT**
  A parameter address referenced memory that was not valid.

- **EIO**
  An error occurred while accessing the timer device.

- **EPERM**
  The requesting process does not have the appropriate privilege to set the specified timer.

If the `time` subroutine is unsuccessful, a return value of -1 is received and the `errno` global variable is set to the following:

- **EFAULT**
  A parameter address referenced memory that was not valid.

**Related Information**

The `asctime` subroutine, `clock` subroutine, `ctime` subroutine, `difftime` subroutine, `gmtime` subroutine, `localtime` subroutine, `mktime` subroutine, `strftime` subroutine, `strptime` subroutine, and `utime` subroutine.

Time data manipulation services in Operating system and device management.

Subroutines Overview in AIX 5L Version 5.3 General Programming Concepts: Writing and Debugging Programs.
gettimerid Subroutine

Purpose
Allocates a per-process interval timer.

Library
Standard C Library (libc.a)

Syntax
#include <sys/time.h>
#include <sys/events.h>

timer_t gettimerid(
    TimerType, NotifyType);

Description
The gettimerid subroutine is used to allocate a per-process interval timer based on the timer with the given timer type. The unique ID is used to identify the interval timer in interval timer requests. (See getinterval subroutine). The particular timer type, the TimerType parameter, is defined in the sys/time.h file and can identify either a systemwide timer or a per-process timer. The mechanism by which the process is to be notified of the expiration of the timer event is the NotifyType parameter, which is defined in the sys/events.h file.

The TimerType parameter represents one of the following timer types:

- **TIMEOFDAY** (POSIX system clock timer) This timer represents the time-of-day clock for the system. For this timer, the values returned by the gettimer subroutine and specified by the settimer subroutine represent the amount of time since 00:00:00 GMT, January 1, 1970, in nanoseconds.
- **TIMERID_ALRM** (Alarm timer) This timer schedules the delivery of a SIGALRM signal at a timer specified in the call to the settimer subroutine.
- **TIMERID_REAL** (Real-time timer) The real-time timer decrements in real time. A SIGALRM signal is delivered when this timer expires.
- **TIMERID_VIRTUAL** (Virtual timer) The virtual timer decrements in process virtual time, it runs only when the process is executing in user mode. A SIGVTALRM signal is delivered when it expires.
- **TIMERID_PROF** (Profiling timer) The profiling timer decrements both when running in user mode and when the system is running for the process. It is designed to be used by processes to profile their execution statistically. A SIGPROF signal is delivered when the profiling timer expires.

Interval timers with a notification value of DELIVERY_SIGNAL are inherited across an exec subroutine.

Parameters

- **NotifyType** Notifies the process of the expiration of the timer event.
- **TimerType** Identifies either a systemwide timer or a per-process timer.
Return Values
If the `gettimerid` subroutine succeeds, it returns a `timer_t` structure that can be passed to the per-process interval timer subroutines, such as the `getinterval` subroutine. If an error occurs, the value -1 is returned and `errno` is set.

Error Codes
If the `gettimerid` subroutine fails, the value -1 is returned and `errno` is set to one of the following error codes:

**EAGAIN**
- The calling process has already allocated all of the interval timers associated with the specified timer type for this implementation.

**EINVAL**
- The specified timer type is not defined.

Related Information
The `exec` subroutine, `fork` subroutine, `getinterval`, `incinterval`, `absinterval`, `resinc`, or `resabs` subroutine, `gettimer`, `settimer`, or `restimer` subroutine, `reltimerid` subroutine.

Time data manipulation services in *Operating system and device management.*

**gettyent, getttynam, setttyent, or endttyent Subroutine**

**Purpose**
Gets a tty description file entry.

**Library**
Standard C Library (`libc.a`)

**Syntax**

```c
#include <ttyent.h>

struct ttyent *getttyent()
struct ttyent *getttynam(Name)
char *Name;
void setttyent()
void endttyent()
```

**Description**

**Attention:** Do not use the `getttyent`, `getttynam`, `setttyent`, or `endttyent` subroutine in a multithreaded environment.

The `getttyent` and `getttynam` subroutines each return a pointer to an object with the `ttyent` structure. This structure contains the broken-out fields of a line from the tty description file. The `ttyent` structure is in the `/usr/include/sys/ttyent.h` file and contains the following fields:
tty_name
The name of the character special file in the /dev directory. The character special file must reside in the /dev directory.

ty_getty
The command that is called by the init process to initialize tty line characteristics. This is usually the getty command, but any arbitrary command can be used. A typical use is to initiate a terminal emulator in a window system.

ty_type
The name of the default terminal type connected to this tty line. This is typically a name from the termcap database. The TERM environment variable is initialized with this name by the getty or login command.

ty_status
A mask of bit fields that indicate various actions to be allowed on this tty line. The following is a description of each flag:

TTY_ON
Enables logins (that is, the init process starts the specified getty command on this entry).

TTY_SECURE
Allows a user with root user authority to log in to this terminal. The TTY_ON flag must be included.

ty_window
The command to execute for a window system associated with the line. The window system is started before the command specified in the ty_getty field is executed. If none is specified, this is null.

ty_comment
The trailing comment field. A leading delimiter and white space is removed.

The gettyent subroutine reads the next line from the tty file, opening the file if necessary. The settyent subroutine rewinds the file. The endttyent subroutine closes it.

The gettynam subroutine searches from the beginning of the file until a matching name (specified by the Name parameter) is found (or until the EOF is encountered).

Parameters

Name Specifies the name of a tty description file.

Return Values
These subroutines return a null pointer when they encounter an EOF (end-of-file) character or an error.

Files

/usr/lib/libodm.a Specifies the ODM (Object Data Manager) library.
/usr/lib/libcfg.a Archives device configuration subroutines.
/etc/termcap Defines terminal capabilities.

Related Information
The ttyslot subroutine.
The getty command, init command, login command.

List of Files and Directories Subroutines in AIX 5L Version 5.3 General Programming Concepts: Writing and Debugging Programs.
getuid, geteuid, or getuidx Subroutine

Purpose
Gets the real or effective user ID of the current process.

Library
Standard C Library (libc.a)

Syntax
#include <sys/types.h>
#include <unistd.h>
uid_t getuid(void)
uid_t geteuid(void)
#include <id.h>
uid_t getuidx (int type);

Description
The getuid subroutine returns the real user ID of the current process. The geteuid subroutine returns the effective user ID of the current process.

The getuidx subroutine returns the user ID indicated by the type parameter of the calling process.

These subroutines are part of Base Operating System (BOS) Runtime.

Return Values
The getuid, geteuid and getuidx subroutines return the corresponding user ID. The getuid and geteuid subroutines always succeed.

The getuidx subroutine will return -1 and set the global errno variable to EINVAL if the type parameter is not one of ID_REAL, ID_EFFECTIVE, ID_SAVED or ID_LOGIN.

Parameters

type Specifies the user ID to get. Must be one of ID_REAL (real user ID), ID_EFFECTIVE (effective user ID), ID_SAVED (saved set-user ID) or ID_LOGIN (login user ID).

Error Codes
If the getuidx subroutine fails the following is returned:

EINVAL Indicates the value of the type parameter is invalid.

Related Information
The setuid subroutine.

List of Security and Auditing Subroutines  Subroutines Overview  in AIX 5L Version 5.3 General Programming Concepts: Writing and Debugging Programs.
getuinfo Subroutine

Purpose
Finds a value associated with a user.

Library
Standard C Library (libc.a)

Syntax
char *getuinfo (Name)
char *Name;

Description
The getuinfo subroutine finds a value associated with a user. This subroutine searches a user information
buffer for a string of the form Name=Value and returns a pointer to the Value substring if the Name value
is found. A null value is returned if the Name value is not found.

The INuibp global variable points to the user information buffer:
extern char *INuibp;

This variable is initialized to a null value.

If the INuibp global variable is null when the getuinfo subroutine is called, the usrinfo subroutine is called
to read user information from the kernel into a local buffer. The INUuibp is set to the address of the local
buffer. If the INuibp external variable is not set, the usrinfo subroutine is automatically called the first time
the getuinfo subroutine is called.

Parameter
Name Specifies a user name.

Related Information
List of Security and Auditing Subroutines, Subroutines Overview in AIX 5L Version 5.3 General
Programming Concepts: Writing and Debugging Programs.

getuinfox Subroutine

Purpose
Finds a value associated with a user.

Library
Standard C Library (libc.a)

Syntax
char *getuinfox (Name)
char *Name;
Description
The `getuinfox` subroutine finds a value associated with a user. This subroutine searches a privileged kernel buffer for a string of the form `Name=Value` and returns a pointer to the `Value` substring if the `Name` value is found. A Null value is returned if the `Name` value is not found. The caller is responsible for freeing the memory returned by the `getuinfox` subroutine.

Parameters
`Name` Specifies a name.

Return Values
Upon success, the `getuinfox` subroutine returns a pointer to the `Value` substring.

Error Codes
A Null value is returned if the `Name` value is not found.

getuserattr, IDtouser, nextuser, or putuserattr Subroutine

Purpose
Accesses the user information in the user database.

Library
Security Library (`libc.a`)

Syntax
```c
#include <usersec.h>

int getuserattr (User, Attribute, Value, Type)
char *User;
char *Attribute;
void *Value;
int Type;

char *IDtouser (uid__t UID);

char *nextuser (Mode, Argument)
int Mode, Argument;

int putuserattr (User, Attribute, Value, Type)
char *User;
char *Attribute;
void *Value;
int Type;
```

Description
**Attention:** These subroutines and the `setpwent` and `setgrent` subroutines should not be used simultaneously. The results can be unpredictable.

These subroutines access user information. Because of their greater granularity and extensibility, you should use them instead of the `getpwent` routines.
The **getuserattr** subroutine reads a specified attribute from the user database. If the database is not already open, this subroutine does an implicit open for reading. A call to the **getuserattr** subroutine for every new user verifies that the user exists.

Similarly, the **putuserattr** subroutine writes a specified attribute into the user database. If the database is not already open, this subroutine does an implicit open for reading and writing. Data changed by the **putuserattr** subroutine must be explicitly committed by calling the **putuserattr** subroutine with a **Type** parameter specifying **SEC_COMMIT**. Until all the data is committed, only these subroutines within the process return written data.

New entries in the user and group databases must first be created by invoking **putuserattr** with the **SEC_NEW** type.

The **IDtouser** subroutine translates a user ID into a user name.

The **nextuser** subroutine returns the next user in a linear search of the user database. The consistency of consecutive searches depends upon the underlying storage-access mechanism and is not guaranteed by this subroutine.

The **setuserdb** and **enduserdb** subroutines should be used to open and close the user database.

The **enduserdb** subroutine frees all memory allocated by the **getuserattr** subroutine.

**Parameters**

**Argument**

Presently unused and must be specified as null.

**Attribute**

Specifies which attribute is read. The following possible attributes are defined in the **usersec.h** file:

- **S_CORECOMP**
  - Core compression status. The attribute type is **SEC_CHAR**.

- **S_COREPATH**
  - Core path specification status. The attribute type is **SEC_CHAR**.

- **S_COREPNAME**
  - Core path specification location. The attribute type is **SEC_CHAR**.

- **S_CORENAMING**
  - Core naming status. The attribute type is **SEC_CHAR**.

- **S_ID**
  - User ID. The attribute type is **SEC_INT**.

- **S_PGID**
  - Principle group ID. The attribute type is **SEC_INT**.

- **S_PGRP**
  - Principle group name. The attribute type is **SEC_CHAR**.

- **S_GROUPS**
  - Groups to which the user belongs. The attribute type is **SEC_LIST**.

- **S_ADMGROUPS**
  - Groups for which the user is an administrator. The attribute type is **SEC_LIST**.

- **S_ADMIN**
  - Administrative status of a user. The attribute type is **SEC_BOOL**.

- **S_AUDITCLASSES**
  - Audit classes to which the user belongs. The attribute type is **SEC_LIST**.
S_AUTHSYSTEM
Defines the user’s authentication method. The attribute type is SEC_CHAR.

S_HOME
Home directory. The attribute type is SEC_CHAR.

S_SHELL
Initial program run by a user. The attribute type is SEC_CHAR.

S_GECOS
Personal information for a user. The attribute type is SEC_CHAR.

S_USRENV
User-state environment variables. The attribute type is SEC_LIST.

S_SYSENV
Protected-state environment variables. The attribute type is SEC_LIST.

S_LOGINCHK
Specifies whether the user account can be used for local logins. The attribute type is SEC_BOOL.

S_HISTEXPIRE
Defines the period of time (in weeks) that a user cannot reuse a password. The attribute type is SEC_INT.

S_HISTSIZE
Specifies the number of previous passwords that the user cannot reuse. The attribute type is SEC_INT.

S_MAXREPEAT
Defines the maximum number of times a user can repeat a character in a new password. The attribute type is SEC_INT.

S_MINAGE
Defines the minimum age in weeks that the user’s password must exist before the user can change it. The attribute type is SEC_INT.

S_PWDCHECKS
Defines the password restriction methods for this account. The attribute type is SEC_LIST.

S_MINALPHA
Defines the minimum number of alphabetic characters required in a new user’s password. The attribute type is SEC_INT.

S_MINDIFF
Defines the minimum number of characters required in a new password that were not in the old password. The attribute type is SEC_INT.

S_MINLEN
Defines the minimum length of a user’s password. The attribute type is SEC_INT.

S_MINOTHER
Defines the minimum number of non-alphabetic characters required in a new user’s password. The attribute type is SEC_INT.

S_DICTIONLIST
Defines the password dictionaries for this account. The attribute type is SEC_LIST.

S_SUCHK
Specifies whether the user account can be accessed with the su command. Type SEC_BOOL.

S_REGISTRY
Defines the user’s authentication registry. The attribute type is SEC_CHAR.
S_RLOGINCHK
Specifies whether the user account can be used for remote logins using the telnet or rlogin commands. The attribute type is SEC_BOOL.

S_DAEMONCHK
Specifies whether the user account can be used for daemon execution of programs and subsystems using the cron daemon or src. The attribute type is SEC_BOOL.

S_TPATH
Defines how the account may be used on the trusted path. The attribute type is SEC_CHAR. This attribute must be one of the following values:
- nosak  The secure attention key is not enabled for this account.
- notsh  The trusted shell cannot be accessed from this account.
- always This account may only run trusted programs.
- on     Normal trusted-path processing applies.

S_TTYS
List of ttys that can or cannot be used to access this account. The attribute type is SEC_LIST.

S_SUGROUPS
Groups that can or cannot access this account. The attribute type is SEC_LIST.

S_EXPIRATION
Expiration date for this account is a string in the form MMDDhhmmyy, where MM is the month, DD is the day, hh is the hour in 0 to 24 hour notation, mm is the minutes past the hour, and yy is the last two digits of the year. The attribute type is SEC_CHAR.

S_AUTH1
Primary authentication methods for this account. The attribute type is SEC_LIST.

S_AUTH2
Secondary authentication methods for this account. The attribute type is SEC_LIST.

S_UFSIZE
Process file size soft limit. The attribute type is SEC_INT.

S_UCPU
Process CPU time soft limit. The attribute type is SEC_INT.

S_UDATA
Process data segment size soft limit. The attribute type is SEC_INT.

S_USTACK
Process stack segment size soft limit. Type: SEC_INT.

S_URSS
Process real memory size soft limit. Type: SEC_INT.

S_UCORE
Process core file size soft limit. The attribute type is SEC_INT.

S_UNOFILE
Process file descriptor table size soft limit. The attribute type is SEC_INT.

S_PWD
Specifies the value of the passwd field in the /etc/passwd file. The attribute type is SEC_CHAR.
S_UMASK
File creation mask for a user. The attribute type is SEC_INT.

S_LOCKED
Specifies whether the user’s account can be logged into. The attribute type is SEC_BOOL.

S_ROLES
Defines the administrative roles for this account. The attribute type is SEC_LIST.

S_UFSIZE_HARD
Process file size hard limit. The attribute type is SEC_INT.

S_UCPU_HARD
Process CPU time hard limit. The attribute type is SEC_INT.

S_UDATA_HARD
Process data segment size hard limit. The attribute type is SEC_INT.

S_USREXPORT
Specifies if the DCE registry can overwrite the local user information with the DCE user information during a DCE export operation. The attribute type is SEC_BOOL.

S_USTACK_HARD
Process stack segment size hard limit. Type: SEC_INT.

S URSS_HARD
Process real memory size hard limit. Type: SEC_INT.

S_UCORE_HARD
Process core file size hard limit. The attribute type is SEC_INT.

S_UNOFILE_HARD
Process file descriptor table size hard limit. The attribute type is SEC_INT.

Note: These values are string constants that should be used by applications both for convenience and to permit optimization in latter implementations. Additional user-defined attributes may be used and will be stored in the format specified by the Type parameter.

Mode
Specifies the search mode. This parameter can be used to delimit the search to one or more user credentials databases. Specifying a non-null Mode value also implicitly rewinds the search. A null Mode value continues the search sequentially through the database. This parameter must include one of the following values specified as a bit mask; these are defined in the usersec.h file:

S_LOCAL
Locally defined users are included in the search.

S_SYSTEM
All credentials servers for the system are searched.

Type
Specifies the type of attribute expected. Valid types are defined in the usersec.h file and include:

SEC_INT
The format of the attribute is an integer.

SEC_CHAR
The format of the attribute is a null-terminated character string.
For the `getuserattr` subroutine, the user should supply a pointer to a defined character pointer variable. For the `putuserattr` subroutine, the user should supply a character pointer.

SEC_LIST
The format of the attribute is a series of concatenated strings, each null-terminated. The last string in the series is terminated by two successive null characters.

For the `getuserattr` subroutine, the user should supply a pointer to a defined character pointer variable. For the `putuserattr` subroutine, the user should supply a character pointer.

SEC_BOOL
The format of the attribute from `getuserattr` is an integer with the value of either 0 (false) or 1 (true). The format of the attribute for `putuserattr` is a null-terminated string containing one of the following strings: true, false, yes, no, always, or never.

For the `getuserattr` subroutine, the user should supply a pointer to a defined integer variable. For the `putuserattr` subroutine, the user should supply a character pointer.

SEC_COMMIT
For the `putuserattr` subroutine, this value specified by itself indicates that changes to the named user are to be committed to permanent storage. The `Attribute` and `Value` parameters are ignored. If no user is specified, the changes to all modified users are committed to permanent storage.

SEC_DELETE
The corresponding attribute is deleted from the database.

SEC_NEW
Updates all the user database files with the new user name when using the `putuserattr` subroutine.

`UID` Specifies the user ID to be translated into a user name.

`User` Specifies the name of the user for which an attribute is to be read.

`Value` Specifies a buffer, a pointer to a buffer, or a pointer to a pointer depending on the `Attribute` and `Type` parameters. See the `Type` parameter for more details.

**Security**

Files Accessed:

<table>
<thead>
<tr>
<th>Mode</th>
<th>File</th>
</tr>
</thead>
<tbody>
<tr>
<td>rw</td>
<td>/etc/passwd</td>
</tr>
<tr>
<td>rw</td>
<td>/etc/group</td>
</tr>
<tr>
<td>rw</td>
<td>/etc/security/user</td>
</tr>
<tr>
<td>rw</td>
<td>/etc/security/limits</td>
</tr>
<tr>
<td>rw</td>
<td>/etc/security/group</td>
</tr>
<tr>
<td>rw</td>
<td>/etc/security/environ</td>
</tr>
</tbody>
</table>

**Return Values**

If successful, the `getuserattr` subroutine with the `S_LOGINCHK` or `S_RLOGINCHK` attribute specified and the `putuserattr` subroutine return 0. Otherwise, a value of -1 is returned and the `errno` global variable is set to indicate the error. For all other attributes, the `getuserattr` subroutine returns 0.
If successful, the **IDtouser** and **nextuser** subroutines return a character pointer to a buffer containing the requested user name. Otherwise, a null pointer is returned and the **errno** global variable is set to indicate the error.

### Error Codes

If any of these subroutines fail, the following is returned:

- **EACCES** Access permission is denied for the data request.

If the **getuserattr** and **putuserattr** subroutines fail, one or more of the following is returned:

- **ENOENT** The specified **User** parameter does not exist.
- **EINVAL** The **Attribute** parameter does not contain one of the defined attributes or null.
- **EINVAL** The **Value** parameter does not point to a valid buffer or to valid data for this type of attribute. Limited testing is possible and all errors may not be detected.
- **EPERM** Operation is not permitted.
- **ENOATTR** The specified attribute is not defined for this user.

If the **IDtouser** subroutine fails, one or more of the following is returned:

- **ENOENT** The specified **User** parameter does not exist

If the **nextuser** subroutine fails, one or more of the following is returned:

- **EINVAL** The **Mode** parameter is not one of null, **S_LOCAL**, or **S_SYSTEM**.
- **EINVAL** The **Argument** parameter is not null.
- **ENOENT** The end of the search was reached.

### Files

`/etc/passwd` Contains user IDs.

### Related Information

The ["getgroupattr, IDtogroup, nextgroup, or putgroupattr Subroutine" on page 370]([#](#)), ["getuserpw, putuserpw, or putuserpwhist Subroutine" on page 463]([#](#)), **setpwdb** subroutine, **setuserdb** subroutine.

List of Security and Auditing Subroutines, Subroutines Overview in AIX 5L Version 5.3 General Programming Concepts: Writing and Debugging Programs.

---

### getuserattrs Subroutine

**Purpose**

Retrieves multiple user attributes in the user database.

**Library**

Security Library (**libc.a**)

**Syntax**

```c
#include <usersec.h>
```
int getuserattrs (User, Attributes, Count)
char * User;
dbattr_t * Attributes;
int Count

Description

Attention: Do not use this subroutine and the setpwent and setgrent subroutines simultaneously. The results can be unpredictable.

The getuserattrs subroutine accesses user information. Because of its greater granularity and extensibility, use it instead of the getpwent routines.

The getuserattrs subroutine reads one or more attributes from the user database. If the database is not already open, this subroutine does an implicit open for reading. A call to the getuserattrs subroutine with an Attributes parameter of null and Count parameter of 0 for every new user verifies that the user exists.

The Attributes array contains information about each attribute that is to be read. The dbattr_t data structure contains the following fields:

attr_name
The name of the desired attribute.

attr_idx
Used internally by the getuserattrs subroutine.

attr_type
The type of the desired attribute. The following possible attributes are defined in the usersec.h file:

S_CORECOMP
Core compression status. The attribute type is SEC_CHAR.

S_COREPATH
Core path specification status. The attribute type is SEC_CHAR.

S_COREPNAME
Core path specification location. The attribute type is SEC_CHAR.

S_CORENAMING
Core naming status. The attribute type is SEC_CHAR.

S_ID
User ID. The attribute type is SEC_INT.

S_Pgid
Principle group ID. The attribute type is SEC_INT.

S_PGRP
Principle group name. The attribute type is SEC_CHAR.

S_GROUPS
Groups to which the user belongs. The attribute type is SEC_LIST.

S_ADMGROUPS
Groups for which the user is an administrator. The attribute type is SEC_LIST.

S_ADMIN
Administrative status of a user. The attribute type is SEC_BOOL.

S_AUDITCLASSES
Audit classes to which the user belongs. The attribute type is SEC_LIST.

S_AUTHSYSTEM
Defines the user’s authentication method. The attribute type is SEC_CHAR.
**S_HOME**
Home directory. The attribute type is **SEC_CHAR**.

**S_SHELL**
Initial program run by a user. The attribute type is **SEC_CHAR**.

**S_GECOS**
Personal information for a user. The attribute type is **SEC_CHAR**.

**S_USRENV**
User-state environment variables. The attribute type is **SEC_LIST**.

**S_SYSENV**
Protected-state environment variables. The attribute type is **SEC_LIST**.

**S_LOGINCHK**
Specifies whether the user account can be used for local logins. The attribute type is **SEC_BOOL**.

**S_HISTEXPIRE**
Defines the period of time (in weeks) that a user cannot reuse a password. The attribute type is **SEC_INT**.

**S_HISTSIZE**
Specifies the number of previous passwords that the user cannot reuse. The attribute type is **SEC_INT**.

**S_MAXREPEAT**
Defines the maximum number of times a user can repeat a character in a new password. The attribute type is **SEC_INT**.

**S_MINAGE**
Defines the minimum age in weeks that the user’s password must exist before the user can change it. The attribute type is **SEC_INT**.

**S_PWDCHECKS**
Defines the password restriction methods for this account. The attribute type is **SEC_LIST**.

**S_MINALPHA**
Defines the minimum number of alphabetic characters required in a new user’s password. The attribute type is **SEC_INT**.

**S_MINDIFF**
Defines the minimum number of characters required in a new password that were not in the old password. The attribute type is **SEC_INT**.

**S_MINLEN**
Defines the minimum length of a user’s password. The attribute type is **SEC_INT**.

**S_MINOTHER**
Defines the minimum number of non-alphabetic characters required in a new user’s password. The attribute type is **SEC_INT**.

**S_DICTIONLIST**
Defines the password dictionaries for this account. The attribute type is **SEC_LIST**.

**S_SUCHK**
Specifies whether the user account can be accessed with the `su` command. Type **SEC_BOOL**.

**S_REGISTRY**
Defines the user’s authentication registry. The attribute type is **SEC_CHAR**.
S_RLOGINCHK
Specifies whether the user account can be used for remote logins using the telnet or rlogin commands. The attribute type is SEC_BOOL.

S_DAEMONCHK
Specifies whether the user account can be used for daemon execution of programs and subsystems using the cron daemon or src. The attribute type is SEC_BOOL.

S_TPATH
Defines how the account may be used on the trusted path. The attribute type is SEC_CHAR. This attribute must be one of the following values:

- nosak  The secure attention key is not enabled for this account.
- notsh  The trusted shell cannot be accessed from this account.
- always This account may only run trusted programs.
- on     Normal trusted-path processing applies.

S_TTYS
List of ttys that can or cannot be used to access this account. The attribute type is SEC_LIST.

S_SUGROUPS
Groups that can or cannot access this account. The attribute type is SEC_LIST.

S_EXPIRATION
Expiration date for this account is a string in the form MMDDhhmmyy, where MM is the month, DD is the day, hh is the hour in 0 to 24 hour notation, mm is the minutes past the hour, and yy is the last two digits of the year. The attribute type is SEC_CHAR.

S_AUTH1
Primary authentication methods for this account. The attribute type is SEC_LIST.

S_AUTH2
Secondary authentication methods for this account. The attribute type is SEC_LIST.

S_UFSIZE
Process file size soft limit. The attribute type is SEC_INT.

S_UCPU
Process CPU time soft limit. The attribute type is SEC_INT.

S_UDATA
Process data segment size soft limit. The attribute type is SEC_INT.

S_USTACK
Process stack segment size soft limit. Type: SEC_INT.

S_URSS
Process real memory size soft limit. Type: SEC_INT.

S_UCORE
Process core file size soft limit. The attribute type is SEC_INT.

S_UNOFILE
Process file descriptor table size soft limit. The attribute type is SEC_INT.

S_PWD
Specifies the value of the passwd field in the /etc/passwd file. The attribute type is SEC_CHAR.
S_UMASK
File creation mask for a user. The attribute type is SEC_INT.

S_LOCKED
Specifies whether the user’s account can be logged into. The attribute type is SEC_BOOL.

S.Roles
Defines the administrative roles for this account. The attribute type is SEC_LIST.

S_UFSIZE_HARD
Process file size hard limit. The attribute type is SEC_INT.

S_UCPU_HARD
Process CPU time hard limit. The attribute type is SEC_INT.

S_UDATA_HARD
Process data segment size hard limit. The attribute type is SEC_INT.

S_USREXPORT
Specifies if the DCE registry can overwrite the local user information with the DCE user information during a DCE export operation. The attribute type is SEC_BOOL.

S_USTACK_HARD
Process stack segment size hard limit. Type: SEC_INT.

S_URSS_HARD
Process real memory size hard limit. Type: SEC_INT.

S_UCORE_HARD
Process core file size hard limit. The attribute type is SEC_INT.

S_UNOFILE_HARD
Process file descriptor table size hard limit. The attribute type is SEC_INT.

attr_flag
The results of the request to read the desired attribute.

attr_un
A union containing the returned values. Its union members that follow correspond to the definitions of the attr_char, attr_int, attr_long, and attr_llong macros, respectively:

un_char
Attributes of type SEC_CHAR and SEC_LIST store a pointer to the returned attribute in this member when the requested attribute is successfully read. The caller is responsible for freeing this memory.

un_int
Attributes of type SEC_INT and SEC_BOOL store the value of the attribute in this member when the requested attribute is successfully read.

un_long
Attributes of type SEC_LONG store the value of the attribute in this member when the requested attribute is successfully read.

un_llong
Attributes of type SEC_LLONG store the value of the attribute in this member when the requested attribute is successfully read.

attr_domain
The authentication domain containing the attribute. The getuserattr subroutine is responsible for managing the memory referenced by this pointer.

If attr_domain is specified for an attribute, the get request is sent only to that domain.

If attr_domain is not specified (that is, set to NULL), getuserattr searches the domains known to the system and sets this field to the name of the domain from which the value is retrieved.
search space can be restricted with the `setauthdb` subroutine so that only the domain specified in the `setauthdb` call is searched.
If the request for a NULL domain was not satisfied, the request is tried from the local files using the default stanza.

Use the `setuserdb` and `enduserdb` subroutines to open and close the user database. Failure to explicitly open and close the user database can result in loss of memory and performance.

**Parameters**

*User*  
Specifies the name of the user for which the attributes are to be read.

*Attributes*  
A pointer to an array of zero or more elements of type `dbattr_t`. The list of user attributes is defined in the `usersec.h` header file.

*Count*  
The number of array elements in `Attributes`.

**Security**

Files accessed:

<table>
<thead>
<tr>
<th>Mode</th>
<th>File</th>
</tr>
</thead>
<tbody>
<tr>
<td>rw</td>
<td>/etc/passwd</td>
</tr>
<tr>
<td>rw</td>
<td>/etc/group</td>
</tr>
<tr>
<td>rw</td>
<td>/etc/security/user</td>
</tr>
<tr>
<td>rw</td>
<td>/etc/security/limits</td>
</tr>
<tr>
<td>rw</td>
<td>/etc/security/group</td>
</tr>
<tr>
<td>rw</td>
<td>/etc/security/environ</td>
</tr>
</tbody>
</table>

**Return Values**

If *User* exists, the `getuserattrs` subroutine returns 0. Otherwise, a value of -1 is returned and the `errno` global variable is set to indicate the error. Each element in the `Attributes` array must be examined on a successful call to `getuserattrs` to determine if the `Attributes` array entry was successfully retrieved.

**Error Codes**

The `getuserattrs` subroutine returns an error that indicates that the user does or does not exist. Additional errors can indicate an error querying the information databases for the requested attributes.

*EINVAL*  
The `Count` parameter is less than 0.

*EINVAL*  
The `Attributes` parameter is null and the `Count` parameter is greater than 0.

*ENOENT*  
The specified `User` parameter does not exist.

If the `getuserattrs` subroutine fails to query an attribute, one or more of the following errors is returned in the `attr_flag` field of the corresponding `Attributes` element:

*EACCESS*  
The user does not have access to the attribute specified in the `attr_name` field.

*EINVAL*  
The `attr_type` field in the `Attributes` entry contains an invalid type.

*EINVAL*  
The `attr_un` field in the `Attributes` entry does not point to a valid buffer or to valid data for this type of attribute. Limited testing is possible and all errors might not be detected.

*ENOATTR*  
The `attr_name` field in the `Attributes` entry specifies an attribute that is not defined for this user or group.

**Examples**

The following sample test program displays the output to a call to `getuserattrs`. In this example, the system has a user named `foo`.
#include <stdio.h>
#include <usersec.h>

#define NATTR 3
#define USERNAME "foo"

main() {

dbattr_t attributes[NATTR];
int i;
int rc;

memset(&attributes, 0, sizeof(attributes));

/*
 * Fill in the attributes array with "id", "expires" and
 * "SYSTEM" attributes.
 */
attributes[0].attr_name = S_ID;
attributes[0].attr_type = SEC_INT;;

attributes[1].attr_name = S_ADMIN;
attributes[1].attr_type = SEC_BOOL;

attributes[2].attr_name = S_AUTHSYSTEM;
attributes[2].attr_type = SEC_CHAR;

/*
 * Make a call to getuserattrs.
 */

setuserdb(S_READ);
rc = getuserattrs(USERNAME, attributes, NATTR);
enduserdb();

if (rc) {
    printf("getuserattrs failed ....\n");
    exit(-1);
}

for (i = 0; i < NATTR; i++) {
    printf("attribute name : %s \n", attributes[i].attr_name);
    printf("attribute flag : %d \n", attributes[i].attr_flag);

    if (attributes[i].attr_flag) {
        /*
         * No attribute value. Continue.
         */
        printf("\n");
        continue;
    }

    /*
     * We have a value.
     */
    printf("attribute domain : %s \n", attributes[i].attr_domain);
    printf("attribute value : ");

    switch (attributes[i].attr_type) {
    case SEC_CHAR:
        if (attributes[i].attr_char)
            printf("%s\n", attributes[i].attr_char);


free(attributes[i].attr_char);
}
    break;
case SEC_INT:
case SEC_BOOL:
    printf("%d\n", attributes[i].attr_int);
    break;
default:
    break;
}
printf("\n");
}
exit(0);
}

The following output for the call is expected:

attribute name : id
attribute flag : 0
attribute domain : files
attribute value : 206

attribute name : admin
attribute flag : 0
attribute domain : files
attribute value : 0

attribute name : SYSTEM
attribute flag : 0
attribute domain : files
attribute value : compat

Files

/etc/passwd Contains user IDs.

Related Information

The “getgroupattrs Subroutine” on page 373, “getuserpw, putuserpw, or putuserpwhist Subroutine” on page 463, “setpwdb Subroutine” “getuserdb Subroutine” in AIX 5L Version 5.3 General Programming Concepts: Writing and Debugging Programs.

GetUserAuths Subroutine

Purpose

Accesses the set of authorizations of a user.

Library

Security Library (libc.a)

Syntax

#include <usersec.h>
char *GetUserAuths(void);
Description
The GetUserAuths subroutine returns the list of authorizations associated with the real user ID and group set of the process. By default, the ALL authorization is returned for the root user.

Return Values
If successful, the GetUserAuths subroutine returns a list of authorizations associated with the user. The format of the list is a series of concatenated strings, each null-terminated. A null string terminates the list. Otherwise, a null pointer is returned and the errno global variable is set to indicate the error.

getUserpw, putuserpw, or putuserpwhist Subroutine

Purpose
Accesses the user authentication data.

Library
Security Library (libc.a)

Syntax
```c
#include <userpw.h>

struct userpw *getuserpw (User);
char *User;

int putuserpw (Password);
struct userpw *Password;

int putuserpwhist (Password, Message);
struct userpw *Password;
char **Message;
```

Description
These subroutines may be used to access user authentication information. Because of their greater granularity and extensibility, you should use them instead of the getpw routines.

The getuserpw subroutine reads the user's locally defined password information. If the setpwdb subroutine has not been called, the getuserpw subroutine will call it as setpwdb (S_READ). This can cause problems if the putuserpw subroutine is called later in the program.

The putuserpw subroutine updates or creates a locally defined password information stanza in the /etc/security/passwd file. The password entry created by the putuserpw subroutine is used only if there is an ! (exclamation point) in the /etc/passwd file's password field. The user application can use the putuserattr subroutine to add an ! to this field.

The putuserpw subroutine will open the authentication database read/write if no other access has taken place, but the program should call setpwdb (S_READ | S_WRITE) before calling the putuserpw subroutine.

The putuserpwhist subroutine updates or creates a locally defined password information stanza in the etc/security/passwd file. The subroutine also manages a database of previous passwords used for password reuse restriction checking. It is recommended to use the putuserpwhist subroutine, rather than the putuserpw subroutine, to ensure the password is added to the password history database.
**Parameters**

*Password* Specifies the password structure used to update the password information for this user. This structure is defined in the `userpw.h` file and contains the following members:

- **upw_name**
  Specifies the user’s name. (The first eight characters must be unique, since longer names are truncated.)

- **upw_passwd**
  Specifies the user’s password.

- **upw_lastupdate**
  Specifies the time, in seconds, since the epoch (that is, 00:00:00 GMT, January 1, 1970), when the password was last updated.

- **upw_flags**
  Specifies attributes of the password. This member is a bit mask of one or more of the following values, defined in the `userpw.h` file.

  - **PW_NOCHECK**
    Specifies that new passwords need not meet password restrictions in effect for the system.

  - **PW_ADMCHG**
    Specifies that the password was last set by an administrator and must be changed at the next successful use of the `login` or `su` command.

  - **PW_ADMIN**
    Specifies that password information for this user may only be changed by the root user.

*Message* Indicates a message that specifies an error occurred while updating the password history database. Upon return, the value is either a pointer to a valid string within the memory allocated storage or a null pointer.

*User* Specifies the name of the user for which password information is read. (The first eight characters must be unique, since longer names are truncated.)

**Security**

Files Accessed:

- **Mode**
  - File
    - `rw` `/etc/security/passwd`

**Return Values**

If successful, the `getuserpw` subroutine returns a valid pointer to a `userpw` structure. Otherwise, a null pointer is returned and the `errno` global variable is set to indicate the error. If the user exists but there is no user entry in the `/etc/security/passwd` file, the `getuserpw` subroutine returns success with the name field set to user name, the password field set to NULL, the lastupdate field set to 0 and the flags field set to 0. If the user exists and there is an entry in the `/etc/security/passwd` file but one or more fields are missing, the `getuserpw` subroutine returns the fields that exist.

If the user exists but there is no user entry in the `/etc/security/passwd` file, the `putuserpw` subroutine creates a user stanza in the `/etc/security/passwd` file. If the user exists and there is an entry in the `/etc/security/passwd` file but one or more fields are missing, the `putuserpw` subroutine updates the fields that exist and creates the fields that are missing.
If successful, the **putuserpwhist** subroutine returns a value of 0. If the subroutine failed to update or create a locally defined password information stanza in the `/etc/security/passwd` file, the **putuserpwhist** subroutine returns a nonzero value. If the subroutine was unable to update the password history database, a message is returned in the `Message` parameter and a return code of 0 is returned. If the user exists but there is no user entry in the `/etc/security/passwd` file, the **putuserpwhist** subroutine creates a user stanza in the `/etc/security/passwd` file and updates the password history. If the user exists and there is an entry in the `/etc/security/passwd` file but one or more fields are missing, the **putuserpwhist** subroutine updates the fields that exist, creates the fields that are missing and modifies the password history.

### Error Codes

The **getuserpw**, **putuserpw**, and **putuserpwhist** subroutines fail if the following values are true:

- **EACCESS**: The user is not able to open the files that contain the password attributes.
- **ENOENT**: The user does not exist in the `/etc/passwd` file.

Subroutines invoked by the **getuserpw**, **putuserpw**, or **putuserpwhist** subroutines can also set errors.

### Files

- `/etc/security/passwd` Contains user passwords.

### Related Information

The “**getgroupattr**, **IDtogroup**, **nextgroup**, or **putgroupattr** Subroutine” on page 370, “**getuserattr**, **IDtouser**, **nextuser**, or **putuserattr** Subroutine” on page 449, **setpwdb** or **endpwdb** subroutine, **setuserdb** subroutine.

[List of Security and Auditing Subroutines](#) and [Subroutines, Example Programs, and Libraries in](#) in AIX 5L Version 5.3 General Programming Concepts: Writing and Debugging Programs.

---

### getuserpwx Subroutine

#### Purpose

Accesses the user authentication data.

#### Library

Security Library (`libc.a`)

#### Syntax

```c
#include <userpw.h>

struct userpwx *getuserpwx (User);
char *User;
```

#### Description

The **getuserpwx** subroutine accesses user authentication information. Because of its greater granularity and extensibility, use it instead of the **getpwent** routines.

The **getuserpwx** subroutine reads the user's password information from the local administrative domain or from a loadable authentication module that supports the required user attributes.
The `getuserpw` subroutine opens the authentication database read-only if no other access has taken place, but the program should call `setpwb` (S_READ) followed by `endpwb` after access to the authentication database is no longer required.

The data returned by `getuserpw` is stored in allocated memory and must be freed by the caller when the data is no longer required. The entire structure can be freed by invoking the `free` subroutine with the pointer returned by `getuserpw`.

**Parameters**

*User* Specifies the name of the user for which password information is read.

**Security**

Files accessed:

<table>
<thead>
<tr>
<th>Mode</th>
<th>File</th>
</tr>
</thead>
<tbody>
<tr>
<td>r</td>
<td>/etc/passwd</td>
</tr>
<tr>
<td>r</td>
<td>/etc/security/passwd</td>
</tr>
</tbody>
</table>

**Return Values**

If successful, the `getuserpw` subroutine returns a valid pointer to a `userpw` structure. Otherwise, a null pointer is returned and the `errno` global variable is set to indicate the error. The fields in a `userpw` structure are defined in the `userpw.h` file, and they include the following members:

- `upw_name` Specifies the user’s name.
- `upw_passwd` Specifies the user’s encrypted password.
- `upw_lastupdate` Specifies the time, in seconds, since the epoch (that is, 00:00:00 GMT, 1 January 1970), when the password was last updated.
- `upw_flags` Specifies attributes of the password. This member is a bit mask of one or more of the following values, defined in the `userpw.h` file:
  - `PW_NOCHECK` Specifies that new passwords need not meet password restrictions in effect for the system.
  - `PW_ADMCHG` Specifies that the password was last set by an administrator and must be changed at the next successful use of the login or `su` command.
  - `PW_ADMIN` Specifies that password information for this user can only be changed by the root user.
- `upw_authdb` Specifies the administrative domain containing the authentication data.

**Error Codes**

The `getuserpw` subroutine fails if one of the following values is true:

- `EACCESS` The user is not able to open the files that contain the password attributes.
- `ENOENT` The user does not have an entry in the `/etc/security/passwd` file or other administrative domain.

Subroutines invoked by the `getuserpw` subroutine can also set errors.
Files

/etc/security/passwd    Contains user passwords.

Related Information
The "getgrouppatr Subroutine" on page 373, "getuserattr, IDtouser, nextuser, or putuserattr Subroutine" on page 449, "getuserattrs Subroutine" on page 455, setpwdb Subroutine, setuserdb Subroutine

getusraclattr, nextusracl or putusraclattr Subroutine

Purpose
Accesses the user screen information in the SMIT ACL database.

Library
Security Library (libc.a)

Syntax
#include <usersec.h>
int getusraclattr(User, Attribute, Value, Type)
char *User;
char *Attribute;
void *Value;
int Type;
char *nextusracl(void)
int putusraclattr(User, Attribute, Value, Type)
char *User;
char *Attribute;
void *Value;
int Type;

Description
The getusraclattr subroutine reads a specified user attribute from the SMIT ACL database. If the database is not already open, this subroutine does an implicit open for reading.

Similarly, the putusraclattr subroutine writes a specified attribute into the user SMIT ACL database. If the database is not already open, this subroutine does an implicit open for reading and writing. Data changed by the putusraclattr subroutine must be explicitly committed by calling the putusraclattr subroutine with a Type parameter specifying SEC_COMMIT. Until all the data is committed, only the getusraclattr subroutine within the process returns written data.

The nextusracl subroutine returns the next user in a linear search of the user SMIT ACL database. The consistency of consecutive searches depends upon the underlying storage-access mechanism and is not guaranteed by this subroutine.

The setacldb and endacldb subroutines should be used to open and close the database.
Parameters

Attribute Specifies which attribute is read. The following possible attributes are defined in the usersec.h file:

S_SCREENS String of SMIT screens. The attribute type is SEC_LIST.

S_ACLMODE String specifying the SMIT ACL database search scope. The attribute type is SEC_CHAR.

S_FUNCMODE String specifying the databases to be searched. The attribute type is SEC_CHAR.

Type Specifies the type of attribute expected. Valid types are defined in the usersec.h file and include:

SEC_CHAR The format of the attribute is a null-terminated character string.

For the getusraclattr subroutine, the user should supply a pointer to a defined character pointer variable. For the putusraclattr subroutine, the user should supply a character pointer.

SEC_LIST The format of the attribute is a series of concatenated strings, each null-terminated. The last string in the series must be an empty (zero character count) string.

For the getusraclattr subroutine, the user should supply a pointer to a defined character pointer variable. For the putusraclattr subroutine, the user should supply a character pointer.

SEC_COMMIT For the putusraclattr subroutine, this value specified by itself indicates that changes to the named user are to be committed to permanent storage. The Attribute and Value parameters are ignored. If no user is specified, the changes to all modified users are committed to permanent storage.

SEC_DELETE The corresponding attribute is deleted from the user SMIT ACL database.

SEC_NEW Updates the user SMIT ACL database file with the new user name when using the putusraclattr subroutine.

Value Specifies a buffer, a pointer to a buffer, or a pointer to a pointer depending on the Attribute and Type parameters. See the Type parameter for more details.

Return Values
If successful, the getusraclattr returns 0. Otherwise, a value of -1 is returned and the errno global variable is set to indicate the error.

Error Codes
Possible return codes are:

EACCES Access permission is denied for the data request.
ENOENT The specified User parameter does not exist or the attribute is not defined for this user.
ENOATTR The specified user attribute does not exist for this user.
EINVAL The Attribute parameter does not contain one of the defined attributes or null.
EINVAL The Value parameter does not point to a valid buffer or to valid data for this type of attribute.
EPERM Operation is not permitted.
Related Information
The getgrpaclattr, nextgrpacl, or putgrpaclattr subroutine, setacldb, or endacldb subroutine.

getutent, getutid, getutline, pututline, setutent, endutent, or utmpname Subroutine

Purpose
Accesses utmp file entries.

Library
Standard C Library (libc.a)

Syntax
#include <utmp.h>

struct utmp *getutent ( )

struct utmp *getutid ( ID )
struct utmp *ID;

struct utmp *getutline ( Line )
struct utmp *Line;

void pututline ( Utmp )
struct utmp *Utmp;

void setutent ( )

void endutent ( )

void utmpname ( File )
char *File;

Description
The getutent, getutid, and getutline subroutines return a pointer to a structure of the following type:

struct utmp
{
    char ut_user[256];        /* User name */
    char ut_id[14];            /* /etc/inittab ID */
    char ut_line[64];          /* Device name (console, lnx) */
    pid_t ut_pid;              /* Process ID */
    short ut_type;             /* Type of entry */
    int __time_t_space;        /* for 32vs64-bit time_t PPC */
    time_t ut_time;            /* time entry was made */
    struct exit_status
    {
        short e_termination;  /* Process termination status */
        short e_exit;          /* Process exit status */
    }
    ut_exit;                   /* The exit status of a process */

    /* marked as DEAD_PROCESS. */
    char ut_host[256];        /* host name */
}
The `getutent` subroutine reads the next entry from a utmp-like file. If the file is not open, this subroutine opens it. If the end of the file is reached, the `getutent` subroutine fails.

The `pututline` subroutine writes the supplied Utmp parameter structure into the utmp file. It is assumed that the user of the `pututline` subroutine has searched for the proper entry point using one of the `getut` subroutines. If not, the `pututline` subroutine calls `getutid` to search forward for the proper place. If so, `pututline` does not search. If the `pututline` subroutine does not find a matching slot for the entry, it adds a new entry to the end of the file.

The `setutent` subroutine resets the input stream to the beginning of the file. Issue a `setuid` call before each search for a new entry if you want to examine the entire file.

The `endutent` subroutine closes the file currently open.

The `utmpname` subroutine changes the name of a file to be examined from `/etc/utmp` to any other file. The name specified is usually `/var/adm/wtmp`. If the specified file does not exist, no indication is given. You are not aware of this fact until your first attempt to reference the file. The `utmpname` subroutine does not open the file. It closes the old file, if currently open, and saves the new file name.

The most current entry is saved in a static structure. To make multiple accesses, you must copy or use the structure between each access. The `getutid` and `getutline` subroutines examine the static structure first. If the contents of the static structure match what they are searching for, they do not read the `utmp` file. Therefore, you must fill the static structure with zeros after each use if you want to use these subroutines to search for multiple occurrences.

If the `pututline` subroutine finds that it is not already at the correct place in the file, the implicit read it performs does not overwrite the contents of the static structure returned by the `getutent` subroutine, the `getuid` subroutine, or the `getutline` subroutine. This allows you to get an entry with one of these subroutines, modify the structure, and pass the pointer back to the `pututline` subroutine for writing.

These subroutines use buffered standard I/O for input. However, the `pututline` subroutine uses an unbuffered nonstandard write to avoid race conditions between processes trying to modify the utmp and wtmp files.

### Parameters

**ID**

If you specify a type of `RUN_LVL`, `BOOT_TIME`, `OLD_TIME`, or `NEW_TIME` in the `ID` parameter, the `getutid` subroutine searches forward from the current point in the utmp file until an entry with a `ut_type` matching `ID->ut_type` is found.

If you specify a type of `INIT_PROCESS`, `LOGIN_PROCESS`, `USER_PROCESS`, or `DEAD_PROCESS` in the `ID` parameter, the `getutid` subroutine returns a pointer to the first entry whose type is one of these four and whose `ut_id` field matches `ID->ut_id`. If the end of the file is reached without a match, the `getutid` subroutine fails.

**Line**

The `getutline` subroutine searches forward from the current point in the utmp file until it finds an entry of type `LOGIN_PROCESS` or `USER_PROCESS` that also has a `ut_line` string matching the `Line->ut_line` parameter string. If the end of file is reached without a match, the `getutline` subroutine fails.

**Utmp**

Points to the utmp structure.

**File**

Specifies the name of the file to be examined.
Return Values
These subroutines fail and return a null pointer if a read or write fails due to a permission conflict or because the end of the file is reached.

Files
/etc/utmp
Path to the utmp file, which contains a record of users logged into the system.
/var/adm/wtmp
Path to the wtmp file, which contains accounting information about users logged in.

Related Information
The ttyslot subroutine.
The failedlogin, utmp, or wtmp file.

getvfsent, getvfsbytype, getvfsbyname, getvfsbyflag, setvfsent, or endvfsent Subroutine

Purpose
Gets a vfs file entry.

Library
Standard C Library(libc.a)

Syntax
#include <sys/vfs.h>
#include <sys/vmount.h>
struct vfs_ent *getvfsent( )

struct vfs_ent *getvfsbytype( vfsType
int vfsType;

struct vfs_ent *getvfsbyname( vfsName
char *vfsName;

struct vfs_ent *getvfsbyflag( vfsFlag
int vfsFlag;
void setvfsent( )
void endvfsent( )

Description
Attention: All information is contained in a static area and so must be copied to be saved.

The getvfsent subroutine, when first called, returns a pointer to the first vfs_ent structure in the file. On the next call, it returns a pointer to the next vfs_ent structure in the file. Successive calls are used to search the entire file.

The vfs_ent structure is defined in the vfs.h file and it contains the following fields:
The `getvfsbytype` subroutine searches from the beginning of the file until it finds a `vfs` type matching the `vfsType` parameter. The subroutine then returns a pointer to the structure in which it was found.

The `getvfsbyname` subroutine searches from the beginning of the file until it finds a `vfs` name matching the `vfsName` parameter. The search is made using flattened names; the search-string uses ASCII equivalent characters.

The `getvfsbytype` subroutine searches from the beginning of the file until it finds a type matching the `vfsType` parameter.

The `getvfsbyflag` subroutine searches from the beginning of the file until it finds the entry whose flag corresponds flags defined in the `vfs.h` file. Currently, these are `VFS_DFLT_LOCAL` and `VFS_DFLT_REMOTE`.

The `setvfsent` subroutine rewinds the `vfs` file to allow repeated searches.

The `endvfsent` subroutine closes the `vfs` file when processing is complete.

**Parameters**

- `vfsType` Specifies a `vfs` type.
- `vfsName` Specifies a `vfs` name.
- `vfsFlag` Specifies either `VFS_DFLT_LOCAL` or `VFS_DFLT_REMOTE`.

**Return Values**

The `getvfsent`, `getvfsbytype`, `getvfsbyname`, and `getvfsbyflag` subroutines return a pointer to a `vfs_ent` structure containing the broken-out fields of a line in the `/etc/vfs` file. If an end-of-file character or an error is encountered on reading, a null pointer is returned.

**Files**

- `/etc/vfs` Describes the virtual file system (VFS) installed on the system.

**Related Information**

The `getfsent`, `getfsspec`, `getfsfile`, `getfstype`, `setfsent`, or `endfsent` subroutine.

`getfsent, getfsspec, getfsfile, getfstype, setfsent, or endfsent Subroutine` on page 364


**getwc, fgetwc, or getwchar Subroutine**

**Purpose**

Gets a wide character from an input stream.
Library
Standard I/O Package (libc.a)

Syntax
#include <stdio.h>

wint_t getwc (Stream)
FILE *Stream;

wint_t fgetwc (Stream)
FILE *Stream;

wint_t getwchar (void)

Description
The fgetwc subroutine obtains the next wide character from the input stream specified by the Stream parameter, converts it to the corresponding wide character code, and advances the file position indicator the number of bytes corresponding to the obtained multibyte character. The getwc subroutine is equivalent to the fgetwc subroutine, except that when implemented as a macro, it may evaluate the Stream parameter more than once. The getwchar subroutine is equivalent to the getwc subroutine with stdin (the standard input stream).

The first successful run of the fgetc ("getc, getchar, fgetc, or getw Subroutine" on page 343), fgets ("gets or fgets Subroutine" on page 429), fgetwc, fgetws ("getws or fgetws Subroutine" on page 475), fread ("fread or fwrite Subroutine" on page 307), scanf, getc ("getc, getchar, fgetc, or getw Subroutine" on page 343), getchar ("getc, getchar, fgetc, or getw Subroutine" on page 343), gets ("gets or fgetws Subroutine" on page 429), or scanf subroutine using a stream that returns data not supplied by a prior call to the ungetc or ungetwc subroutine marks the st_atime field for update.

Parameters
Stream Specifies input data.

Return Values
Upon successful completion, the getwc and fgetwc subroutines return the next wide character from the input stream pointed to by the Stream parameter. The getwchar subroutine returns the next wide character from the input stream pointed to by stdin.

If the end of the file is reached, an indicator is set and WEOF is returned. If a read error occurs, an error indicator is set, WEOF is returned, and the errno global variable is set to indicate the error.

Error Codes
If the getwc, fgetwc, or getwchar subroutine is unsuccessful because the stream is not buffered or data needs to be read into the buffer, it returns one of the following error codes:

EAGAIN Indicates that the O_NONBLOCK flag is set for the file descriptor underlying the Stream parameter, delaying the process.

EBADF Indicates that the file descriptor underlying the Stream parameter is not valid and cannot be opened for reading.

EINTR Indicates that the process has received a signal that terminates the read operation.

EIO Indicates that a physical error has occurred, or the process is in a background process group attempting to read from the controlling terminal, and either the process is ignoring or blocking the SIGTTIN signal or the process group is orphaned.

EOVERFLOW Indicates that the file is a regular file and an attempt has been made to read at or beyond the offset maximum associated with the corresponding stream.
The `getwc`, `fgetwc`, or `getwchar` subroutine is also unsuccessful due to the following error conditions:

- **ENOMEM** Indicates that storage space is insufficient.
- **ENXIO** Indicates that the process sent a request to a nonexistent device, or the device cannot handle the request.
- **EILSEQ** Indicates that the `wc` wide-character code does not correspond to a valid character.

### Related Information

Other wide character I/O subroutines: `getws` or `fgetws` ("getws or fgetws Subroutine” on page 475) subroutine, `putwc`, `putwchar`, or `fputwc` ("putwc, putwchar, or fputwc Subroutine” on page 1317) subroutine, `putws` or `fputws` ("putws or fputws Subroutine” on page 1319) subroutine, `ungetwc` subroutine.

Related standard I/O subroutines: `fopen`, `freopen`, or `fdopen` ("fopen, fopen64, freopen, freopen64 or fdopen Subroutine” on page 284) subroutine, `gets` or `fgets` ("gets or fgets Subroutine” on page 429) subroutine, `fread` ("fread or fwrite Subroutine” on page 307) subroutine, `fwrite` ("fread or fwrite Subroutine” on page 307) subroutine, `printf`, `fprintf`, `sprintf`, `wprintf`, `vprintf`, `vfprintf`, `vsprintf`, or `vswprintf` ("printf, fprintf, sprintf, wprintf, vprintf, vfprintf, vsprintf, or vswprintf Subroutine” on page 1148) subroutine, `putc`, `putchar`, `fputc`, or `putw` ("putc, putchar, fputc, or putw Subroutine” on page 1299) subroutine, `puts` or `fputs` ("puts or fputs Subroutine” on page 1309) subroutine.

---

### getwd Subroutine

#### Purpose

Gets current directory path name.

#### Library

Standard C Library (`libc.a`)

#### Syntax

```c
#include <unistd.h>

char *getwd (PathName);
char *PathName;
```

#### Description

The `getwd` subroutine determines the absolute path name of the current directory, then copies that path name into the area pointed to by the `PathName` parameter.

The maximum path-name length, in characters, is set by the `PATH_MAX` value, as specified in the `limits.h` file.
Parameters

PathName
Points to the full path name.

Return Values
If the call to the getwd subroutine is successful, a pointer to the absolute path name of the current directory is returned. If an error occurs, the getwd subroutine returns a null value and places an error message in the PathName parameter.

Related Information
The getcwd subroutine.

Related Information
The getcwd subroutine.

Files, Directories, and File Systems for Programmers in AIX 5L Version 5.3 General Programming Concepts: Writing and Debugging Programs.

getws or fgetws Subroutine

Purpose
Gets a string from a stream.

Library
Standard I/O Library (libc.a)

Syntax

#include <stdio.h>

wchar_t *fgetws (WString, Number, Stream)
wchar_t *WString;
int Number;
FILE *Stream;
wchar_t *getws (WString)
wchar_t *WString;

Description
The fgetws subroutine reads characters from the input stream, converts them to the corresponding wide character codes, and places them in the array pointed to by the WString parameter. The subroutine continues until either the number of characters specified by the Number parameter minus 1 are read or the subroutine encounters a new-line or end-of-file character. The fgetws subroutine terminates the wide character string specified by the WString parameter with a null wide character.

The getws subroutine reads wide characters from the input stream pointed to by the standard input stream (stdin) into the array pointed to by the WString parameter. The subroutine continues until it encounters a new-line or the end-of-file character, then it discards any new-line character and places a null wide character after the last character read into the array.

Parameters

WString
Points to a string to receive characters.

Stream
Points to the FILE structure of an open file.

Number
Specifies the maximum number of characters to read.
Return Values

If the getws or fgetws subroutine reaches the end of the file without reading any characters, it transfers no characters to the String parameter and returns a null pointer. If a read error occurs, the getws or fgetws subroutine returns a null pointer and sets the errno global variable to indicate the error.

Error Codes

If the getws or fgetws subroutine is unsuccessful because the stream is not buffered or data needs to be read into the stream's buffer, it returns one or more of the following error codes:

- **EAGAIN**: Indicates that the O_NONBLOCK flag is set for the file descriptor underlying the Stream parameter, and the process is delayed in the fgetws subroutine.
- **EBADF**: Indicates that the file descriptor specifying the Stream parameter is not a read-access file.
- **EINTR**: Indicates that the read operation is terminated due to the receipt of a signal, and either no data was transferred or the implementation does not report partial transfer for this file.
- **EIO**: Indicates that insufficient storage space is available.
- **ENOMEM**: Indicates that insufficient storage space is available.
- **EILSEQ**: Indicates that the data read from the input stream does not form a valid character.

Related Information

Other wide character I/O subroutines: fgetwc subroutine, fputwc subroutine, putws or fputws subroutine, getwc subroutine, getwchar subroutine, getwchar Subroutine, putwc subroutine, putwchar subroutine, putws subroutine, putwchar subroutine.

Related standard I/O subroutines: fdopen subroutine, fgetc subroutine, getchar subroutine, fgetwc subroutine, getwchar subroutine, getwc subroutine, fscanf subroutine, printf subroutine, scanf subroutine, sprintf subroutine, vsprintf subroutine, ungetwc subroutine.

Understanding Wide Character Input/Output Subroutines and Subroutines, Example Programs, and Libraries in AIX 5L Version 5.3 General Programming Concepts: Writing and Debugging Programs.

glob Subroutine

Purpose
Generates path names.

Library
Standard C Library (libc.a)

Syntax
#include <glob.h>

int glob (Pattern, Flags, (Errfunc)(), Pglob)
const char *Pattern;
int Flags;
int *Errfunc (Epath, Eerrno)
const char *Epath;
int Eerrno;
glob_t *Pglob;

Description
The glob subroutine constructs a list of accessible files that match the Pattern parameter.

The glob subroutine matches all accessible path names against this pattern and develops a list of all matching path names. To have access to a path name, the glob subroutine requires search permission on every component of a path except the last, and read permission on each directory of any file name component of the Pattern parameter that contains any of the special characters * (asterisk), ? (question mark), or [ (left bracket). The glob subroutine stores the number of matched path names and a pointer to a list of pointers to path names in the Pglob parameter. The path names are in sort order, based on the setting of the LC_COLLATE category in the current locale. The first pointer after the last path name is a null character. If the pattern does not match any path names, the returned number of matched paths is zero.

Parameters
Pattern
Contains the file name pattern to compare against accessible path names.

Flags
Controls the customizable behavior of the glob subroutine.

The Flags parameter controls the behavior of the glob subroutine. The Flags value is the bitwise inclusive OR of any of the following constants, which are defined in the glob.h file:

GLOB_APPEND
Appends path names located with this call to any path names previously located. If the GLOB_APPEND constant is not set, new path names overwrite previous entries in the Pglob array. The GLOB_APPEND constant should not be set on the first call to the glob subroutine. It may, however, be set on subsequent calls.

The GLOB_APPEND flag can be used to append a new set of path names to those found in a previous call to the glob subroutine. If the GLOB_APPEND flag is specified in the Flags parameter, the following rules apply:

• If the application sets the GLOB_DOOFFS flag in the first call to the glob subroutine, it is also set in the second. The value of the Pglob parameter is not modified between the calls.
• If the application did not set the GLOB_DOOFFS flag in the first call to the glob subroutine, it is not set in the second.

• After the second call, the Pglob parameter points to a list containing the following:
  – Zero or more null characters, as specified by the GLOB_DOOFFS flag.
  – Pointers to the path names that were in the Pglob list before the call, in the same order as after the first call to the glob subroutine.
  – Pointers to the new path names generated by the second call, in the specified order.

• The count returned in the Pglob parameter is the total number of path names from the two calls.

• The application should not modify the Pglob parameter between the two calls.

It is the caller’s responsibility to create the structure pointed to by the Pglob parameter. The glob subroutine allocates other space as needed.

GLOB_DOOFFS
  Uses the gl_offs structure to specify the number of null pointers to add to the beginning of the gl_pathv component of the Pglob parameter.

GLOB_ERR
  Causes the glob subroutine to return when it encounters a directory that it cannot open or read. If the GLOB_ERR flag is not set, the glob subroutine continues to find matches if it encounters a directory that it cannot open or read.

GLOB_MARK
  Specifies that each path name that is a directory should have a / (slash) appended.

GLOB_NOCHECK
  If the Pattern parameter does not match any path name, the glob subroutine returns a list consisting only of the Pattern parameter, and the number of matched patterns is one.

GLOB_NOSORT
  Specifies that the list of path names need not be sorted. If the GLOB_NOSORT flag is not set, path names are collated according to the current locale.

GLOB_QUOTE
  If the GLOB_QUOTE flag is set, a \ (backslash) can be used to escape metacharacters.

Errfunc
  Specifies an optional subroutine that, if specified, is called when the glob subroutine detects an error condition.

Pglob
  Contains a pointer to a glob_t structure. The structure is allocated by the caller. The array of structures containing the file names matching the Pattern parameter are defined by the glob subroutine. The last entry is a null pointer.

Epath
  Specifies the path that failed because a directory could not be opened or read.

Eerrno
  Specifies the errno value of the failure indicated by the Epath parameter. This value is set by the opendir, readdir, or stat subroutines.

Return Values
On successful completion, the glob subroutine returns a value of 0. The Pglob parameter returns the number of matched path names and a pointer to a null-terminated list of matched and sorted path names. If the number of matched path names in the Pglob parameter is zero, the pointer in the Pglob parameter is undefined.
Error Codes

If the *glob* subroutine terminates due to an error, it returns one of the nonzero constants below. These are defined in the *glob.h* file. In this case, the *Pglobs* values are still set as defined in the Return Values section.

<table>
<thead>
<tr>
<th>Constant</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>GLOB_ABORTED</td>
<td>Indicates the scan was stopped because the GLOB_ERROR flag was set or the subroutine specified by the <em>errfunc</em> parameter returned a nonzero value.</td>
</tr>
<tr>
<td>GLOB_NOSPACE</td>
<td>Indicates a failed attempt to allocate memory.</td>
</tr>
</tbody>
</table>

If, during the search, a directory is encountered that cannot be opened or read and the *Errfunc* parameter is not a null value, the *glob* subroutine calls the subroutine specified by the *errfunc* parameter with two arguments:

- The *Epath* parameter specifies the path that failed.
- The *Eerrno* parameter specifies the value of the *errno* global variable from the failure, as set by the *opendir*, *readdir*, or *stat* subroutine.

If the subroutine specified by the *Errfunc* parameter is called and returns nonzero, or if the GLOB_ERR flag is set in the *Flags* parameter, the *glob* subroutine stops the scan and returns GLOB_ABORTED after setting the *Pglobs* parameter to reflect the paths already scanned. If GLOB_ERR is not set and either the *Errfunc* parameter is null or *errfunc* returns zero, the error is ignored.

The *Pglobs* parameter has meaning even if the *glob* subroutine fails. Therefore, the *glob* subroutine can report partial results in the event of an error. However, if the number of matched path names is 0, the pointer in the *Pglobs* parameter is unspecified even if the *glob* subroutine did not return an error.

Examples

The GLOB_NOCHECK flag can be used with an application to expand any path name using wildcard characters. However, the GLOB_NOCHECK flag treats the pattern as just a string by default. The *sh* command can use this facility for option parameters, for example.

The GLOB_DOOFFS flag can be used by applications that build an argument list for use with the *execv*, *execve*, or *execvp* subroutine. For example, an application needs to do the equivalent of *ls* -l *.c*, but for some reason cannot. The application could still obtain approximately the same result using the sequence:

```c
globbuf.gl_offs = 2;
glob ("*.c", GLOB_DOOFFS, NULL, &globbuf);
globbuf.gl_pathv[0] = "ls";
globbuf.gl_pathv[1] = "-l";
execvp ("ls", &globbuf.gl_pathv[0]);
```

Using the same example, *ls* -l *.c* *.h* could be approximated using the GLOB_APPEND flag as follows:

```c
globbuf.gl_offs = 2;
glob ("*.c", GLOB_DOOFFS, NULL, &globbuf);
glob ("*.h", GLOB_DOOFFS|GLOB_APPEND, NULL, &globbuf);
```

The new path names generated by a subsequent call with the GLOB_APPEND flag set are not sorted together with the previous path names. This is the same way the shell handles path name expansion when multiple expansions are done on a command line.

Related Information

The exec: *execl*, *execv*, *exect*, *execle*, *execvp*, *execvp*, or *execve* subroutine, fnmatch ("fnmatch Subroutine" on page 282) subroutine, opendir, readdir, telldir, seekdir, rewinddir, or closedir subroutine.
globfree Subroutine

Purpose
Frees all memory associated with the \texttt{pglob} parameter.

Library
Standard C Library (\texttt{libc.a})

Syntax
\begin{verbatim}
#include <glob.h>

void globfree ( \texttt{pglob})
glob_t *pglob;
\end{verbatim}

Description
The \texttt{globfree} subroutine frees any memory associated with the \texttt{pglob} parameter due to a previous call to the \texttt{glob} subroutine.

Parameters
\begin{itemize}
  \item \texttt{pglob} Structure containing the results of a previous call to the \texttt{glob} subroutine.
\end{itemize}

Related Information
The \texttt{glob} subroutine.

\begin{verbatim}
#include <stdlib.h>

int grantpt ( \texttt{FileDescriptor})
int FileDescriptor;
\end{verbatim}
Description
The `grantpt` subroutine changes the mode and the ownership of the slave pseudo-terminal associated with the master pseudo-terminal device defined by the `FileDescriptor` parameter. The user ID of the slave pseudo-terminal is set to the real UID of the calling process. The group ID of the slave pseudo-terminal is set to an unspecified group ID. The permission mode of the slave pseudo-terminal is set to readable and writeable by the owner, and writeable by the group.

Parameters

`FileDescriptor` Specifies the file descriptor of the master pseudo-terminal device.

Return Value
Upon successful completion, a value of 0 is returned. Otherwise, a value of -1 is returned and the `errno` global variable is set to indicate the error.

Error Codes
The `grantpt` function may fail if:

- **EBADF**: The `fildes` argument is not a valid open file descriptor.
- **EINVAL**: The `fildes` argument is not associated with a master pseudo-terminal device.
- **EACCES**: The corresponding slave pseudo-terminal device could not be accessed.

Related Information
The `unlockpt` subroutine.


HBA_CloseAdapter Subroutine

Purpose
Closes the adapter opened by the `HBA_OpenAdapter` subroutine.

Library
Common Host Bus Adapter Library (`libHBAAPI.a`)

Syntax
```
#include <sys/hbaapi.h>

void HBA_CloseAdapter (HBA_HANDLE handle);
```

Description
The `HBA_CloseAdapter` subroutine closes the file associated with the file handle that was the result of a call to the `HBA_OpenAdapter` subroutine. The `HBA_CloseAdapter` subroutine calls the `close` subroutine, and applies it to the file handle. After performing the operation, the handle is set to NULL.

Parameters

`handle` Specifies the open file descriptor obtained from a successful call to the `open` subroutine.
Related Information

The "HBA_OpenAdapter Subroutine" on page 498.

Special Files in AIX 5L Version 5.3 Files Reference describes specific qualities of the files that define devices.

HBA_FreeLibrary Subroutine

Purpose
Frees all the resources allocated to build the Common HBA API Library.

Library
Common Host Bus Adapter Library (libHBAAPI.a)

Syntax
#include <sys/hbaapi.h>
HBA_STATUS HBA_FreeLibrary ()

Description
The HBA_FreeLibrary subroutine frees all resources allocated to build the Common HBA API library. This subroutine must be called after calling any other routine from the Common HBA API library.

Error Codes
The Storage Area Network Host Bus Adapter API subroutines return the following error codes:

HBA_STATUS_OK
A value of 0 on successful completion.

HBA_STATUS_ERROR
A value of 1 if an error occurred.

Related Information
The "HBA_LoadLibrary Subroutine" on page 497.

Special Files in AIX 5L Version 5.3 Files Reference describes specific qualities of the files that define devices.

HBA_GetAdapterAttributes, HBA_GetPortAttributes, HBA_GetDiscoveredPortAttributes, HBA_GetPortAttributesByWWN Subroutine

Purpose
Gets the attributes of the end device’s adapter, port, or remote port.

Library
Common Host Bus Adapter Library (libHBAAPI.a)
Syntax

```c
#include <sys/hbaapi.h>

HBA_STATUS HBA_GetAdapterAttributes (handle, hbaattributes);
HBA_STATUS HBA_GetAdapterPortAttributes (handle, portindex, portattributes);
HBA_STATUS HBA_GetDiscoveredPortAttributes (handle, portindex, discoveredportindex, portattributes);
HBA_STATUS HBA_GetPortAttributesByWWN (handle, PortWWN, portattributes);

HBA_HANDLE handle;
HBA_ADAPTERATTRIBUTES *hbaattributes;
HBA_UINT32 portindex;
HBA_PORTATTRIBUTES *portattributes;
HBA_UINT32 discoveredportindex;
HBA_WWN PortWWN;
```

Description

The **HBA_GetAdapterAttributes** subroutine queries the ODM and makes system calls to gather information pertaining to the adapter. This information is returned to the **HBA_ADAPTERATTRIBUTES** structure. This structure is defined in the `/usr/include/sys/hbaapi.h` file.

The **HBA_GetAdapterAttributes**, **HBA_GetAdapterPortAttributes**, **HBA_GetDiscoveredPortAttributes**, and **HBA_GetPortAttributesByWWN** subroutines return the attributes of the adapter, port or remote port.

These attributes include:
- Manufacturer
- SerialNumber
- Model
- ModelDescription
- NodeWWN
- NodeSymbolicName
- HardwareVersion
- DriverVersion
- OptionROMVersion
- FirmwareVersion
- VendorSpecificID
- NumberOfPorts
- Drivername

The **HBA_GetAdapterPortAttributes**, **HBA_GetDiscoveredPortAttributes**, and **HBA_GetPortAttributesByWWN** subroutines also query the ODM and make system calls to gather information. The gathered information pertains to the port attached to the adapter or discovered on the network. The attributes are stored in the **HBA_PORTATTRIBUTES** structure. This structure is defined in the `/usr/include/sys/hbaapi.h` file.

These attributes include:
- NodeWWN
- PortWWN
- PortFcId
- PortType
- PortState
- PortSupportedClassofService
- PortSupportedFc4Types
The **HBA_GetAdapterPortAttributes** subroutine returns the attributes of the attached port.

The **HBA_GetDiscoveredPortAttributes**, and **HBA_GetPortAttributesByWWN** subroutines return the same information. However, these subroutines differ in the way they are called, and in the way they acquire the information.

### Parameters

- **handle**: Specifies the open file descriptor obtained from a successful call to the `open` subroutine.
- **hbaattributes**: Points to an `HBA_AdapterAttributes` structure, which is used to store information pertaining to the Host Bus Adapter.
- **portindex**: Specifies the index number of the port where the information was obtained.
- **portattributes**: Points to an `HBA_PortAttributes` structure used to store information pertaining to the port attached to the Host Bus Adapter.
- **discoveredportindex**: Specifies the index of the attached port discovered over the network.
- **PortWWN**: Specifies the world wide name or port name of the target device.

### Return Values

Upon successful completion, the attributes and a value of **HBA_STATUS_OK**, or 0 are returned.

If no information for a particular attribute is available, a null value is returned for that attribute. **HBA_STATUS_ERROR** or 1 is returned if certain ODM queries or system calls fail while trying to retrieve the attributes.

### Error Codes

The Storage Area Network Host Bus Adapter API subroutines return the following error codes:

- **HBA_STATUS_OK**: A value of 0 on successful completion.
- **HBA_STATUS_ERROR**: A value of 1 if an error occurred.
- **HBA_STATUS_ERROR_INVALID_HANDLE**: A value of 3 if there was an invalid file handle.
- **HBA_STATUS_ERROR_ARG**: A value of 4 if there was a bad argument.
- **HBA_STATUS_ERROR_ILLEGAL_WWN**: A value of 5 if the world wide name was not recognized.

### Related Information

- "**HBA_GetAdapterName Subroutine**" on page 485, and "**HBA_GetNumberOfAdapters Subroutine**" on page 492.

**Special Files** in **AIX 5L Version 5.3 Files Reference** describes specific qualities of the files that define devices.
HBA_GetAdapterName Subroutine

Purpose
Gets the name of a Common Host Bus Adapter.

Library
Common Host Bus Adapter Library (libHBAAP1.a)

Syntax
#include <sys/hbaapi.h>

HBA_STATUS HBA_GetAdapterName (adapterindex, adaptername);
HBA_UINT32 adapterindex;
char *adaptername;

Description
The HBA_GetAdapterName subroutine gets the name of a Common Host Bus Adapter. The adapterindex parameter is an index into an internal table containing all FCP adapters on the machine. The adapterindex parameter is used to search the table and obtain the adapter name. The name of the adapter is returned in the form of mgfdomain-model-adapterindex. The name of the adapter is used as an argument for the HBA_OpenAdapter subroutine. From the HBA_OpenAdapter subroutine, the file descriptor will be obtained where additional Common HBA API routines can then be called using the file descriptor as the argument.

Parameters

adapterindex Specifies the index of the adapter held in the adapter table for which the name of the adapter is to be returned.
adaptername Points to a character string that will be used to hold the name of the adapter.

Return Values
Upon successful completion, the HBA_GetAdapterName subroutine returns the name of the adapter and a 0, or a status code of HBA_STATUS_OK. If unsuccessful, a null value will be returned for adaptername and an value of 1, or a status code of HBA_STATUS_ERROR.

Error Codes
The Storage Area Network Host Bus Adapter API subroutines return the following error codes:

HBA_STATUS_OK
A value of 0 on successful completion.

HBA_STATUS_ERROR
A value of 1 if an error occurred.

HBA_STATUS_ERROR_NOT_SUPPORTED
A value of 2 if the function is not supported.

HBA_STATUS_ERROR_INVALID_HANDLE
A value of 3 if there was an invalid file handle.

HBA_STATUS_ERROR_ARG
A value of 4 if there was a bad argument.

HBA_STATUS_ERROR_ILLEGAL_WWN
A value of 5 if the world wide name was not recognized.

HBA_STATUS_ERROR_ILLEGAL_INDEX
A value of 6 if an index was not recognized.

HBA_STATUS_ERROR_MORE_DATA
A value of 7 if a larger buffer is required.

HBA_STATUS_ERROR_STALE_DATA
A value of 8 if information has changed since the last call to the HBA_RefreshInformation subroutine.

Base Operating System (BOS) Runtime Services (A-P) 485
HBA_STATUS_SCSI_CHECK_CONDITION
A value of 9 if a SCSI Check Condition was reported.

HBA_STATUS_ERROR_BUSY
A value of 10 if the adapter was busy or reserved. Try again later.

HBA_STATUS_ERROR_TRY_AGAIN
A value of 11 if the request timed out. Try again later.

HBA_STATUS_ERROR_UNAVAILABLE
A value of 12 if the referenced HBA has been removed or deactivated.

Related Information
The "HBA_GetNumberOfAdapters Subroutine" on page 492.

Special Files in AIX 5L Version 5.3 Files Reference describes specific qualities of the files that define devices.

HBA_GetEventBuffer Subroutine
Purpose
Removes and returns the next events from the HBA’s event queue.

Syntax
HBA_STATUS HBA_GetEventBuffer(
    HBA_HANDLE handle,
    HBA_EVENTINFO *pEventBuffer,
    HBA_UINT32 *pEventCount
);

Description
The HBA_GetEventBuffer function removes and returns the next events from the HBA’s event queue. The number of events returned is the lesser of the value of the EventCount parameter at the time of the call and the number of entries available in the event queue.

Parameters
handle
A handle to an open HBA.
pEventBuffer
Pointer to a buffer to receive events.
pEventCount
Pointer to the number of event records that fit in the space allocated for the buffer to receive events. It is set to the size (in event records) of the buffer for receiving events on call, and is returned as the number of events actually delivered.

Return Values
The value of the HBA_GetEventBuffer function is a valid status return value that indicates the reason for completion of the requested function. HBA_STATUS_OK is returned to indicate that no errors were encountered and pEventCount indicates the number of event records returned. A valid status return value that most closely describes the result of the function should be returned to indicate a reason with no required value.

The return values for the following parameters are as follows:

pEventBuffer
Remains unchanged. The buffer to which it points contains event records representing previously undelivered events.
Remains unchanged. The value of the integer to which it points contains the number of event records that actually were delivered.

**Error Codes**

**HBA_STATUS_ERROR**  
Returned to indicate any problem with no required value.

**Related Information**


---

**HBA_GetFC4Statistics Subroutine**

**Purpose**  
Returns traffic statistics for a specific FC-4 protocol through a specific local HBA and local end port.

**Syntax**

```c
HBA_STATUS HBA_GetFC4Statistics(
    HBA_HANDLE handle,
    HBA_WWN hbaPortWWN,
    HBA_UINT8 FC4type,
    HBA_FC4STATISTICS *statistics
);
```

**Description**

The **HBA_GetFC4Statistics** function returns traffic statistics for a specific FC-4 protocol through a specific local HBA and local end port.

**Note:** Basic Link Service, Extended Link Service, and CT each have specific Data Structure TYPE values, so their traffic can be requested.

**Parameters**

- **handle**  
  A handle to an open HBA containing the end port for which FC-4 statistics can return.

- **hbaPortWWN**  
  The Port Name of the local HBA end port for which FC-4 statistics can return.

- **FC4type**  
  The Data Structure TYPE assigned by FC-FS to the FC-4 protocol for which FC-4 statistics are requested.

- **statistics**  
  A pointer to an FC-4 Statistics structure in which the statistics for the specified FC-4 protocol can be returned.

**Return Values**

The value of the **HBA_GetFC4Statistics** function is a valid status return value that indicates the reason for completion of the requested function. **HBA_STATUS_OK** is returned to indicate that the statistics for the specified FC-4 and end port have been returned. A valid status return value that most closely describes the result of the function should be returned to indicate a reason with no required value.
The return value for the following parameter is as follows:

statistics Remains unchanged. The structure to which it points contains the statistics for the specified FC-4 protocol.

## Error Codes

**HBA_STATUS_ERROR_ILLEGAL_WWN**
- Indicates that the HBA referenced by `handle` does not contain an end port with Port Name `hbaPortWWN`.

**HBA_STATUS_ERROR_UNSUPPORTED_FC4**
- Indicates that the specified HBA end port does not support the specified FC-4 protocol.

**HBA_STATUS_ERROR**
- Returned to indicate any problem with no required value.

## Related Information

- "HBA_GetEventBuffer Subroutine" on page 486
- "HBA_GetFCPStatistics Subroutine" on page 489
- "HBA_GetFcpTargetMappingV2 Subroutine" on page 490
- "HBA_GetPersistentBindingV2 Subroutine" on page 493
- "HBA_OpenAdapterByWWN Subroutine" on page 498
- "HBA_ScsiInquiryV2 Subroutine" on page 499
- "HBA_ScsiReadCapacityV2 Subroutine" on page 501
- "HBA_ScsiReportLunsV2 Subroutine" on page 502
- "HBA_SendCTPassThruV2 Subroutine" on page 506
- "HBA_SendRNL Subroutine" on page 510
- "HBA_SendRNIDV2 Subroutine" on page 512
- "HBA_SendRPS Subroutine" on page 514
- "HBA_ScsiGetPersistentBinding Subroutine" on page 515

## HBA_GetFcpPersistentBinding Subroutine

### Purpose
Gets persistent binding information of SCSI LUNs.

### Library
Common Host Bus Adapter Library (libHBAAPI.a)

### Syntax
```c
#include <sys/hbaapi.h>

HBA_STATUS HBA_GetFcpPersistentBinding (HBA_HANDLE handle, PHBA_FCPBINDING binding);
```

### Description
For the specified HBA_HANDLE, the `HBA_GetFcpPersistentBinding` subroutine returns the full binding information of local SCSI LUNs to FCP LUNs for each child of the specified HBA_HANDLE. Applications must allocate memory for the `HBA_FCPBINDING` structure, and also pass to the subroutine the number of entries allocated. If the subroutine determines that the structure is not large enough to represent the full binding information, it will set the `NumberOfEntries` variable to the correct value and return an error.

### Parameters
- `handle` An HBA_HANDLE to an open adapter.
A pointer to a structure containing the binding information of the handle’s children. The `HBA_FCPBINDING` structure has the following form:

```
struct HBA_FCPBinding {
    HBA_UINT32 NumberOfEntries;
    HBA_FCPBINDINGENTRY entry[1]; /* Variable length array */
};
```

The size of the structure is determined by the calling application, and is passed in by the `NumberOfEntries` variable.

**Return Values**

Upon successful completion, HBA_STATUS_OK is returned, and the `binding` parameter points to the full binding structure. If the application has not allocated enough space for the full binding, HBA_STATUS_ERROR_MORE_DATA is returned and the `NumberOfEntries` field in the binding structure is set to the correct value.

**Error Codes**

If there is insufficient space allocated for the full binding. HBA_STATUS_ERROR_MORE_DATA is returned.

**Related Information**

The "HBA_GetFcpTargetMapping Subroutine" on page 491.

---

**HBA_GetFCPStatistics Subroutine**

**Purpose**

Returns traffic statistics for a specific OS SCSI logical unit provided by the FCP protocol on a specific local HBA.

**Syntax**

```
HBA_STATUS HBA_GetFCPStatistics(
    HBA_HANDLE handle,
    const HBA_SCSIID *lunit,
    HBA_FC4STATISTICS *statistics
);
```

**Description**

The `HBA_GetFCPStatistics` function returns traffic statistics for a specific OS SCSI logical unit provided by the FCP protocol on a specific local HBA.

**Parameters**

- `handle` A handle to an open HBA containing the end port for which FCP-2 statistics can be returned.
- `lunit` Pointer to a structure specifying the OS SCSI logical unit for which FCP-2 statistics are requested.
- `statistics` Pointer to a FC-4 Statistics structure in which the FCP-2 statistics for the specified logical unit can be returned.

**Return Values**

The value of the `HBA_GetFCPStatistics` function is a valid status return value that indicates the reason for completion of the requested function. HBA_STATUS_OK is returned to indicate that FCP-2 statistics...
have been returned for the specified HBA. A valid status return value that most closely describes the result of the function should be returned to indicate a reason with no required value.

The return value for the following parameter is as follows:

- **statistics**
  - Remains unchanged. The structure to which it points contains the FCP-2 statistics for the specified HBA and logical unit.

**Error Codes**

- **HBA_STATUS_ERROR_INVALID_LUN**
  - The HBA referenced by `handle` does not support the logical unit referenced by `lunit`.

- **HBA_STATUS_ERROR_UNSUPPORTED_FC4**
  - The specified HBA end port does not support FCP-2.

- **HBA_STATUS_ERROR**
  - Returned to indicate any problem with no required value.

**Related Information**

- "HBA_GetEventBuffer Subroutine" on page 486
- "HBA_GetFC4Statistics Subroutine" on page 487
- "HBA_GetFcpTargetMappingV2 Subroutine" on page 493
- "HBA_GetPersistentBindingV2 Subroutine" on page 493
- "HBA_OpenAdapterByWWN Subroutine" on page 498
- "HBA_ScsiInquiryV2 Subroutine" on page 500
- "HBA_ScsiReadCapacityV2 Subroutine" on page 502
- "HBA_ScsiReadCapacityV2 Subroutine" on page 502
- "HBA_ScsiReportLunsV2 Subroutine" on page 504
- "HBA_SendCTPassThruV2 Subroutine" on page 506
- "HBA_SendCTPassThruV2 Subroutine" on page 506
- "HBA_SendRLS Subroutine" on page 510
- "HBA_SendRNIDV2 Subroutine" on page 512
- "HBA_SendRPL Subroutine" on page 514
- "HBA_SendRPS Subroutine" on page 515

**HBA_GetFcpTargetMappingV2 Subroutine**

**Purpose**

Returns the mapping between OS targets or logical units and FCP targets or logical units offered by the specified HBA end port at the time the function call is processed.

**Syntax**

```c
HBA_STATUS HBA_GetFcpTargetMappingV2(
    HBA_HANDLE handle,
    HBA_WWN hbaPortWWN,
    HBA_FCPTARGETMAPPINGV2 *pMapping
);
```

**Description**

The **HBA_GetFcpTargetMappingV2** function returns the mapping between OS identification of SCSI targets or logical units and FCP identification of targets or logical units offered by the specified HBA end port at the time the function call is processed. Space in the `pMapping` structure permitting, one mapping entry is returned for each FCP logical unit represented in the OS and one mapping entry is returned for each FCP target that is represented in the OS but for which no logical units are represented in the OS. No target mapping entries are returned to represent FCP objects that are not represented in the OS (that is, objects that are unmapped).

The mappings returned include a Logical Unit Unique Device Identifier (LUID) for each logical unit that provides one. For logical units that provide more than one LUID, the LUID returned is the type 3 (FC Name_Identifier) LUID with the smallest identifier value if any LUID of type 3 is provided; otherwise, the type 2 (IEEE EUI-64) LUID with the smallest identifier value if any LUID of type 2 is provided; otherwise, the type 1 (T10 vendor identification) LUID with the smallest identifier value if any LUID of type 1 is provided; otherwise, the type 0 (vendor specific) LUID with the smallest identifier value. If the logical unit provides no LUID, the value of the first four bytes of the LUID field are 0.
Parameters

handle
A handle to an open HBA containing the end port for which target mappings are requested.

hbaPortWWN
Port Name of the local HBA end port for which target mappings are requested.

pMapping
Pointer to an HBA_FCPTARGETMAPPINGV2 structure. The size of this structure shall be limited by the NumberOfEntries value within the structure.

Return Values

The value of the HBA_GetFcpTargetMappingV2 function is a valid status return value that indicates the reason for completion of the requested function. HBA_STATUS_OK is returned to indicate that all mapping entries have been returned for the specified end port. A valid status return value that most closely describes the result of the function should be returned to indicate a reason with no required value.

The return value for the following parameter is as follows:

pMapping
Remains unchanged. The structure to which it points contains mapping information from OS identifications of SCSI logical units to FCP identifications of logical units for the specified local HBA end port. The number of entries in the structure is the minimum of the number of entries specified at function call or the full mapping. The value of the NumberOfEntries field of the returned structure is the total number of mappings the end port has established. This is true even when the function returns an error stating that the buffer is too small to return all of the established mappings. An upper-level application can either allocate a sufficiently large buffer and check this value after a read, or do a read of the NumberOfEntries value separately and allocate a new buffer given the size to accommodate the entire mapping structure.

Error Codes

HBA_STATUS_ERROR_MORE_DATA
More space in the buffer is required to contain mapping information.

HBA_STATUS_ERROR_ILLEGAL_WWN
The HBA referenced by handle does not contain an end port with Port Name hbaPortWWN.

HBA_STATUS_ERROR_NOT_SUPPORTED
The HBA referenced by handle does not support target mapping.

HBA_STATUS_ERROR
Returned to indicate any problem with no required value.

Related Information

"HBA_GetEventBuffer Subroutine" on page 486, "HBA_GetFC4Statistics Subroutine" on page 487, "HBA_GetFCPStatistics Subroutine" on page 489, "HBA_GetPersistentBindingV2 Subroutine" on page 493, "HBA_OpenAdapterByWWN Subroutine" on page 498, "HBA_ScsiInquiryV2 Subroutine" on page 500, "HBA_ScsiReadCapacityV2 Subroutine" on page 502, "HBA_ScsiReportLunsV2 Subroutine" on page 504, "HBA_SendCTPassThruV2 Subroutine" on page 506, "HBA_SendRLS Subroutine" on page 510, "HBA_SendRNIDV2 Subroutine" on page 512, "HBA_SendRPL Subroutine" on page 514, "HBA_SendRPS Subroutine" on page 515

HBA_GetFcpTargetMapping Subroutine

Purpose

Gets mapping of OS identification to FCP indentification for each child of the specified HBA_HANDLE.

Library

Common Host Bus Adapter Library (libHBAAPI.a)
Syntax

```
#include <sys/hbaapi.h>

HBA_STATUS HBA_GetFcpTargetMapping (HBA_HANDLE handle, PHBA_FCPTARGETMAPPING mapping);
```

### Description

For the specified HBA_HANDLE, the `HBA_GetFcpTargetMapping` subroutine maps OS identification of all its SCSI logical units to their FCP indentification. Applications must allocate memory for the `HBA_FCPTARGETMAPPING` structure, and also pass to the subroutine the number of entries allocated. If the subroutine determines that the structure is not large enough to represent the entire mapping, it will set the `NumberOfEntries` variable to the correct value and return an error.

### Parameters

- **handle**
  - An HBA_HANDLE to an open adapter.

- **mapping**
  - A pointer to a structure containing a mapping of the handle's children. The `HBA_FCPTARGETMAPPING` structure has the following form:

  ```
  struct HBA_FCPTargetMapping {
    HBA_UINT32 NumberOfEntries;
    HBA_FCPSCSIENTRY entry[1] /* Variable length array containing mappings */
  };
  ```

  The size of the structure is determined by the calling application, and is passed in by the `NumberOfEntries` variable.

### Return Values

If successful, `HBA_STATUS_OK` is returned and the mapping parameter points to the full mapping structure. If the application has not allocated enough space for the full mapping, `HBA_STATUS_ERROR_MORE_DATA` is returned, and the `NumberOfEntries` field in the mapping structure is set to the correct value.

### Error Codes

If there is insufficient space allocated for the full mapping, `HBA_STATUS_ERROR_MORE_DATA` is returned.

### Related Information

- Special Files in AIX 5L Version 5.3 Files Reference describes specific qualities of the files that define devices.

---

**HBA_GetNumberOfAdapters Subroutine**

### Purpose

Returns the number of adapters discovered on the system.

### Library

Common Host Bus Adapter Library (`libHBAAPI.a`)
Syntax
#include <sys/hbaapi.h>

HBA_UINT32 HBA_GetNumberOfAdapters ()

Description
The HBA_GetNumberOfAdapters subroutine returns the number of HBAs supported by the library. The value returned is the current number of HBAs and reflects dynamic change of the HBA inventory without requiring a restart of the system, driver, or library.

Return Values
The HBA_GetNumberOfAdapters subroutine returns an integer representing the number of adapters on the machine.

Related Information
The "HBA_GetAdapterName Subroutine" on page 485.

Special Files in AIX 5L Version 5.3 Files Reference describes specific qualities of the files that define devices.

HBA_GetPersistentBindingV2 Subroutine

Purpose
Returns persistent bindings between an FCP target and a SCSI ID for a specified HBA end port.

Syntax
HBA_STATUS HBA_GetPersistentBindingV2(
    HBA_HANDLE handle,
    HBA_WWN hbaPortWWN,
    HBA_FCPTARGETMAPPINGV2 *binding
);

Description
The HBA_GetFcpPersistentBindingV2 function returns persistent bindings between an FCP target and a SCSI ID for a specified HBA end port. The binding information can include bindings to Logical Unit Unique Device Identifiers (LUIDs).

Parameters
handle A handle to an open HBA containing the end port for which persistent binding can be returned.
hbaPortWWN The Port Name of the local HBA end port for which persistent binding can be returned.
binding Pointer to an HBA_FCPBINDING2 structure. The NumberOfEntries field in the structure limits the number of entries that are returned.

Return Values
The value of the HBA_GetPersistentBindingV2 function is a valid status return value that indicates the reason for completion of the requested function. HBA_STATUS_OK is returned to indicate that all binding entries have been returned for the specified end port. A valid status return value that most closely describes the result of the function should be returned to indicate a reason with no required value.
The return value for the following parameter is as follows:

binding  
Remains unchanged. The structure to which it points contains binding information from OS identifications of SCSI logical units to FCP and LUID identifications of logical units for the specified HBA end port. The number of entries in the structure is the minimum of the number of entries specified at function call or the full set of bindings. The NumberOfEntries field contains the total number of bindings established by the end port. An application can either call HBA_GetPersistentBindingV2 with NumberOfEntries set to 0 to retrieve the number of entries available, or allocate a sufficiently large buffer to retrieve entries at first call. The Status field of each HBA_FCPBINDINGENTRY2 substructure is 0.

Error Codes

- **HBA_STATUS_ERROR_MORE_DATA**  
  More space in the buffer is required to contain binding information.

- **HBA_STATUS_ERROR_ILLEGAL_WWN**  
  The HBA referenced by handle does not contain an end port with Port Name hbaPortWWN.

- **HBA_STATUS_ERROR_NOT_SUPPORTED**  
  The HBA referenced by handle does not support persistent binding.

- **HBA_STATUS_ERROR**  
  Returned to indicate any problem with no required value.

Related Information

- "HBA_GetEventBuffer Subroutine" on page 486
- "HBA_GetFC4Statistics Subroutine" on page 487
- "HBA_GetFCPStatistics Subroutine" on page 489
- "HBA_GetFcpTargetMappingV2 Subroutine" on page 490
- "HBA_OpenAdapterByWWN Subroutine" on page 498
- "HBA_ScsiInquiryV2 Subroutine" on page 500
- "HBA_ScsiReadCapacityV2 Subroutine" on page 502
- "HBA_ScsiReportLunsV2 Subroutine" on page 504
- "HBA_SendCTPassThruV2 Subroutine" on page 506
- "HBA_SendRNIDV2 Subroutine" on page 512
- "HBA_SendRPL Subroutine" on page 514
- "HBA_SendRPS Subroutine" on page 515

HBA_GetPortStatistics Subroutine

**Purpose**

Gets the statistics for a Host Bus Adapter (HBA).

**Library**

Common Host Bus Adapter Library (libHBAAPI.a)

**Syntax**

```c
#include <sys/hbaapi.h>

HBA_STATUS HBA_GetPortStatistics (HBA_HANDLE handle, HBA_UINT32 portindex, HBA_PORTSTATISTICS *portstatistics);
```

**Description**

The **HBA_GetPortStatistics** subroutine retrieves the statistics for the specified adapter. Only single-port adapters are supported, and the portindex parameter is disregarded. The exact meaning of events being counted for each statistic is vendor specific. The **HBA_PORTSTATISTICS** structure includes the following fields:

- **SecondsSinceLastReset**
• TxFrames
• TxWords
• RxFrames
• RxWords
• LIPCount
• NOSCount
• ErrorFrames
• DumpedFrames
• LinkFailureCount
• LossOfSyncCount
• LossOfSignalCount
• PrimitiveSeqProtocolErrCount
• InvalidTxWordCount
• InvalidCRCCount

**Parameters**

`handle` HBA_HANDLE to an open adapter.
`portindex` Not used.
`portstatistics` Pointer to an HBA_PORTSTATISTICS structure.

**Return Values**

Upon successful completion, HBA_STATUS_OK is returned. If the subroutine is unable to retrieve the statistics for an HBA, it returns HBA_STATUS_ERROR.

---

**HBA_GetRNIDMgmtInfo Subroutine**

**Purpose**

Sends a SCSI GET RNID command to a remote port of the end device.

**Library**

Common Host Bus Adapter Library (libHBAAPI.a)

**Syntax**

```c
#include <sys/hbaapi.h>

HBA_STATUS HBA_GetRNIDMgmtInfo (handle, pInfo)
HBA_HANDLE handle;
HBA_MGMTINFO *pInfo;
```

**Description**

The HBA_GetRNIDMgmtInfo subroutine sends a SCSI GET RNID (Request Node Identification Data) command through a call to ioctl with the SCIOLCHBA operation as its argument. The arg parameter for the SCIOLCHBA operation is the address of a scsi_chba structure. This structure is defined in the /usr/include/sys/scsi_buf.h file. The scsi_chba parameter block allows the caller to select the GET RNID command to be sent to the adapter. The pInfo structure stores the RNID data returned from SCIOLCHBA. The pInfo structure is defined in the /usr/include/sys/hbaapi.h file. The structure includes:

- wwn
- unittype
If successful, the GET RNID data in pInfo is returned from the adapter.

Parameters

`handle` Specifies the open file descriptor obtained from a successful call to the `open` subroutine.

`pInfo` Specifies the structure containing the information to get or set from the RNID command.

Return Values

Upon successful completion, the HBA_GetRNIDMgmtInfo subroutine returns a pointer to a structure containing the data from the GET RNID command and a value of HBA_STATUS_OK, or a value of 0. If unsuccessful, a null value is returned along with a value of HBA_STATUS_ERROR, or a value of 1.

Upon successful completion, the HBA_SetRNIDMgmtInfo subroutine returns a value of HBA_STATUS_OK, or a value of 0. If unsuccessful, an HBA_STATUS_ERROR value, or a value of 1 is returned.

Error Codes

The Storage Area Network Host Bus Adapter API subroutines return the following error codes:

- **HBA_STATUS_OK**: A value of 0 on successful completion.
- **HBA_STATUS_ERROR**: A value of 1 if an error occurred.
- **HBA_STATUS_ERROR_INVALID_HANDLE**: A value of 3 if there was an invalid file handle.

Related Information

- “HBA_SendScsiInquiry Subroutine” on page 516
- “HBA_SendReadCapacity Subroutine” on page 507
- “HBA_SendCTPassThru Subroutine” on page 505
- “HBA_SendReportLUNs Subroutine” on page 508
- “HBA_SendRNID Subroutine” on page 511
- “HBA_SetRNIDMgmtInfo Subroutine” on page 518

Special Files in AIX 5L Version 5.3 Files Reference.


**HBA_GetVersion Subroutine**

**Purpose**

Returns the version number of the Common HBA API.
Library
Common Host Bus Adapter Library (libHBAAPLI.a)

Syntax
#include <sys/hbaapi.h>
HBA_UINT32 HBA_GetVersion ()

Description
The HBA_GetVersion subroutine returns the version number representing the release of the Common HBA API.

Return Values
Upon successful completion, the HBA_GetVersion subroutine returns an integer value designating the version number of the Common HBA API.

Related Information
“HBA_LoadLibrary Subroutine” and “HBA_FreeLibrary Subroutine” on page 482

Special Files in AIX 5L Version 5.3 Files Reference describes specific qualities of the files that define devices.

HBA_LoadLibrary Subroutine

Purpose
Loads a vendor specific library from the Common HBA API.

Library
Common Host Bus Adapter Library (libHBAAPLI.a)

Syntax
#include <sys/hbaapi.h>
HBA_STATUS HBA_LoadLibrary ()

Description
The HBA_LoadLibrary subroutine loads a vendor specific library from the Common HBA API. This library must be called first before calling any other routine from the Common HBA API.

Return Values
The HBA_LoadLibrary subroutine returns a value of 0, or HBA_STATUS_OK.

Related Information
The “HBA_FreeLibrary Subroutine” on page 482

Special Files in AIX 5L Version 5.3 Files Reference describes specific qualities of the files that define devices.
HBA_OpenAdapter Subroutine

Purpose
Opens the specified adapter for reading.

Library
Common Host Bus Adapter Library (libHBAAPI.a)

Syntax
#include <sys/hbaapi.h>

HBA_HANDLE HBA_OpenAdapter (char *adaptername);

Description
The HBA_OpenAdapter subroutine opens the adapter for reading for the purpose of getting it ready for additional calls from other subroutines in the Common HBA API.

The HBA_OpenAdapter subroutine allows an application to open a specified HBA device, giving the application access to the device through the HBA_HANDLE return value. The library ensures that all access to this HBA_HANDLE between HBA_OpenAdapter and HBA_CloseAdapter calls is to the same device.

Parameters
adaptername Specifies a string that contains the description of the adapter as returned by the HBA_GetAdapterName subroutine.

Return Values
If successful, the HBA_OpenAdapter subroutine returns an HBA_HANDLE with a value greater than 0. If unsuccessful, the subroutine returns a 0.

Related Information
"HBA_CloseAdapter Subroutine" on page 481, and "HBA_GetAdapterName Subroutine" on page 485.

HBA_OpenAdapterByWWN Subroutine

Purpose
Attempts to open a handle to the HBA that contains a Node_Name or N_Port_Name matching the wwn argument.

Syntax
HBA_STATUS HBA_OpenAdapterByWWN (HBA_HANDLE *pHandle,
HBA_WWN wwn);

498 Technical Reference, Volume 1: Base Operating System and Extensions
Description
The HBA_OpenAdapterByWWN function attempts to open a handle to the HBA that contains a Node_Name or N_Port_Name matching the wwn argument. The specified Name_Identifier matches the Node_Name or N_Port_Name of the HBA. Discovered end ports (remote end ports) are not checked for a match.

Parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>pHandle</td>
<td>Pointer to a handle. The value at entry is irrelevant.</td>
</tr>
<tr>
<td>wwn</td>
<td>Name_Identifier to match the Node_Name or N_Port_Name of the HBA to open.</td>
</tr>
</tbody>
</table>

Return Values
The value of the HBA_OpenAdapterByWWN function is a valid status return value that indicates the reason for completion of the requested function. HBA_STATUS_OK is returned to indicate that the handle contains a valid HBA handle.

The return values for the following parameter is as follows:

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>pHandle</td>
<td>Remains unchanged. If the open succeeds, the value to which it points is a handle to the requested HBA. On failure, the value is undefined.</td>
</tr>
</tbody>
</table>

Error Codes

<table>
<thead>
<tr>
<th>Error Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>HBA_STATUS_ERROR_ILLEGAL_WWN</td>
<td>There is no HBA with a Node_Name or N_Port_Name that matches wwn.</td>
</tr>
<tr>
<td>HBA_STATUS_ERROR_AMBIGUOUS_WWN</td>
<td>Multiple HBAs have a matching Name_Identifier. This can occur if the Node_Names of multiple HBAs are identical.</td>
</tr>
<tr>
<td>HBA_STATUS_ERROR</td>
<td>Returned to indicate any other problem with opening the HBA.</td>
</tr>
</tbody>
</table>

Related Information


HBA_RefreshInformation Subroutine

Purpose
Refreshes stale information from the Host Bus Adapter.

Library
Common Host Bus Adapter Library (libHBAAPI.a)
Syntax
#include <sys/hbaapi.h>

void HBA_RefreshInformation ( handle
HBA_HANDLE handle; )

Description
The HBA_RefreshInformation subroutine refreshes stale information from the Host Bus Adapter. This
would reflect changes to information obtained from calls to the HBA_GetAdapterPortAttributes, or
HBA_GetDiscoveredPortAttributes subroutine. Once the application calls the HBA_RefreshInformation
subroutine, it can proceed to the attributes’s call to get the new data.

Parameters
handle Specifies the open file descriptor obtained from a successful call to the open subroutine for which the
refresh operation is to be performed.

Related Information
Special Files in AIX 5L Version 5.3 Files Reference describes specific qualities of the files that define
devices.

HBA_ScsiInquiryV2 Subroutine

Purpose
Sends a SCSI INQUIRY command to a remote end port.

Syntax
HBA_STATUS HBA_ScsiInquiryV2 ( HBA_HANDLE handle,
HBA_WWN hbaPortWWN,
HBA_WWN discoveredPortWWN,
HBA_UINT64 fcLUN,
HBA_UINT8 CDB_Byte1,
HBA_UINT8 CDB_Byte2,
void *pRspBuffer,
HBA_UINT32 *pRspBufferSize,
HBA_UINT8 *pScsiStatus,
void *pSenseBuffer,
HBA_UINT32 *pSenseBufferSize
);

Description
The HBA_ScsiInquiryV2 function sends a SCSI INQUIRY command to a remote end port.

A SCSI command is never sent to an end port that does not have SCSI target functionality. If sending a
SCSI command causes a SCSI overlapped command condition with a correctly operating target, the
command does not get sent. Proper use of tagged commands is an acceptable means of avoiding a SCSI
overlapped command condition with targets that support tagged commands.

Parameters
handle Open HBA through which the SCSI INQUIRY command can be issued.
hbaPortWWN The Port Name for a local HBA end port through which the SCSI INQUIRY command can
be issued.
discoveredPortWWN  The Port Name for an end port to which the **SCSI INQUIRY** command can be sent.

fcLUN  The SCSI LUN to which the **SCSI INQUIRY** command can be sent.

**CDB_Byte1**  The second byte of the CDB for the **SCSI INQUIRY** command. This contains control flag bits. At the time this standard was written, the effects of the value of CDB_Byte1 on a **SCSI INQUIRY** command were as follows:

- 0
  - Requests the standard SCSI INQUIRY data.
- 1
  - Requests the vital product data (EVPD) specified by CDB_Byte2.
- 2
  - Requests command support data (CmdDt) for the command specified in CDB_Byte2.
- Other values
  - Can cause SCSI Check Condition.

**CDB_Byte2**  The third byte of the CDB for the **SCSI INQUIRY** command. If CDB_Byte1 is 1, CDB_Byte2 is the Vital Product Data page code to request. If CDB_Byte1 is 2, CDB_Byte2 is the Operation Code of the command support data requested. For other values of CDB_Byte1, the value of CDB_Byte2 is unspecified, and values other than 0 can cause a SCSI Check Condition.

**pRspBuffer**  A pointer to a buffer to receive the **SCSI INQUIRY** command response.

**pRspBufferSize**  A pointer to the size in bytes of the buffer to receive the **SCSI INQUIRY** command response.

**pScsiStatus**  A pointer to a buffer to receive SCSI status.

**pSenseBuffer**  A pointer to a buffer to receive SCSI sense data.

**pSenseBufferSize**  A pointer to the size in bytes of the buffer to receive SCSI sense data.

### Return Values

The value of the **HBA_ScsiInquiryV2** function is a valid status return value that indicates the reason for completion of the requested function. **HBA_STATUS_OK** is returned to indicate that the complete payload of a reply to the **SCSI INQUIRY** command has been returned. A valid status return value that most closely describes the result of the function should be returned to indicate a reason with no required value.

The return values for the following parameters are as follows:

- **pRspBuffer**  Remains unchanged. If the function value is **HBA_STATUS_OK**, the buffer to which it points contains the response to the **SCSI INQUIRY** command.

- **pRspBufferSize**  Remains unchanged. The value of the integer to which it points is the size in bytes of the response returned by the command. This cannot exceed the size passed as an argument at this pointer.

- **pScsiStatus**  Remains unchanged. The value of the byte to which it points is the SCSI status. If the function value is **HBA_STATUS_OK** or **HBA_STATUS_SCSI_CHECK_CONDITION**, the value of the SCSI status can be interpreted based on the SCSI spec. A SCSI status of **HBA_STATUS_OK** indicates that a SCSI response is in the response buffer. A SCSI status of **HBA_STATUS_SCSI_CHECK_CONDITION** indicates that no value is stored in the response, and the sense buffer contains failure information if available.

- **pSenseBuffer**  Remains unchanged. If the function value is **HBA_STATUS_SCSI_CHECK_CONDITION**, the buffer to which it points contains the sense data for the command.

- **pSenseBufferSize**  Remains unchanged. The value of the integer to which it points is the size in bytes of the sense information returned by the command. This cannot exceed the size passed as an argument at this pointer.
Error Codes

**HBA_STATUS_ERROR_ILLEGAL_WWN**
The HBA referenced by handle does not contain an end port with Port Name `hbaPortWWN`.

**HBA_STATUS_ERROR_NOT_A_TARGET**
The identified remote end port does not have SCSI Target functionality.

**HBA_STATUS_ERROR_TARGET_BUSY**
Unable to send the requested command without causing a SCSI overlapped command condition.

**HBA_STATUS_ERROR**
Returned to indicate any problem with no required value.

Related Information

- “HBA_GetEventBuffer Subroutine” on page 486
- “HBA_GetFC4Statistics Subroutine” on page 487
- “HBA_GetFCPStatistics Subroutine” on page 489
- “HBA_GetFCpTargetMappingV2 Subroutine” on page 490
- “HBA_GetPERSISTENTBINDINGV2 Subroutine” on page 493
- “HBA_OpenAdapterByWWN Subroutine” on page 498
- “HBA_ScsiReadCapacityV2 Subroutine” on page 502
- “HBA_ScsiReportLunsV2 Subroutine” on page 504
- “HBA_SendCTPassThruV2 Subroutine” on page 506
- “HBA_SendRLS Subroutine” on page 510
- “HBA_SendRNIDV2 Subroutine” on page 512
- “HBA_SendRPL Subroutine” on page 514
- “HBA_SendRPS Subroutine” on page 515

### HBA_ScsiReadCapacityV2 Subroutine

**Purpose**
Sends a SCSI READ CAPACITY command to a remote end port.

**Syntax**

```c
HBA_STATUS HBA_ScsiReadCapacityV2(
    HBA_HANDLE handle,
    HBA_WWN hbaPortWWN,
    HBA_WWN discoveredPortWWN,
    HBA_UINT64 fcLUN,
    void *pRspBuffer,
    HBA_UINT32 *pRspBufferSize,
    HBA_UINT8 *pScsiStatus,
    void *pSenseBuffer,
    HBA_UINT32 *pSenseBufferSize
);
```

**Description**
The `HBA_ScsiReadCapacityV2` function sends a SCSI READ CAPACITY command to a remote end port.

A SCSI command is never sent to an end port that does not have SCSI target functionality. If sending a SCSI command causes a SCSI overlapped command condition with a correctly operating target, the command will not be sent. Proper use of tagged commands is an acceptable means of avoiding a SCSI overlapped command condition with targets that support tagged commands.

**Parameters**

- `handle`:
  A handle to an open HBA through which the SCSI READ CAPACITY command is issued.
- `hbaPortWWN`:
  The Port Name for a local HBA end port through which the SCSI READ CAPACITY command is issued.
- `discoveredPortWWN`:
  The Port Name for an end port to which the SCSI READ CAPACITY command is sent.
- `fcLUN`:
  The SCSI LUN to which the SCSI READ CAPACITY command is sent.
- `pRspBuffer`:
  Pointer to a buffer to receive the SCSI READ CAPACITY command response.
**pRspBufferSize**  
Pointer to the size in bytes of the buffer to receive the SCSI READ CAPACITY command response.

**pScsiStatus**  
Pointer to a buffer to receive SCSI status.

**pSenseBuffer**  
Pointer to a buffer to receive SCSI sense data.

**pSenseBufferSize**  
Pointer to the size in bytes of the buffer to receive SCSI sense data.

### Return Values

The value of the **HBA_ScsiReadCapacityV2** function is a valid status return value that indicates the reason for completion of the requested function. **HBA_STATUS_OK** is returned to indicate that the complete payload of a reply to the SCSI READ CAPACITY command has been returned. A valid status return value that most closely describes the result of the function should be returned to indicate a reason with no required value.

The return values for the following parameters are as follows:

- **pRspBuffer**
  Remains unchanged. If the function value is **HBA_STATUS_OK**, the buffer to which it points contains the response to the SCSI READ CAPACITY command.

- **pRspBufferSize**
  Remains unchanged. The value of the integer to which it points is the size in bytes of the response returned by the command. This cannot exceed the size passed as an argument at this pointer.

- **pScsiStatus**
  Remains unchanged. The value of the byte to which it points is the SCSI status. If the function value is **HBA_STATUS_OK** or **HBA_STATUS_SCSI_CHECK_CONDITION**, the value of the SCSI status can be interpreted based on the SCSI spec. A SCSI status of **HBA_STATUS_OK** indicates that a SCSI response is in the response buffer. A SCSI status of **HBA_STATUS_SCSI_CHECK_CONDITION** indicates that no value is stored in the response, and the sense buffer contains failure information if available.

- **pSenseBuffer**
  Remains unchanged. If the function value is **HBA_STATUS_SCSI_CHECK_CONDITION**, the buffer to which it points contains the sense data for the command.

- **pSenseBufferSize**
  Remains unchanged. The value of the integer to which it points is the size in bytes of the sense information returned by the command. This cannot exceed the size passed as an argument at this pointer.

### Error Codes

- **HBA_STATUS_ERROR_ILLEGAL_WWN**  
The HBA referenced by **handle** does not contain an end port with Port Name **hbaPortWWN**.

- **HBA_STATUS_ERROR_NOT_A_TARGET**  
The identified remote end port does not have SCSI Target functionality.

- **HBA_STATUS_ERROR_TARGET_BUSY**  
Unable to send the requested command without causing a SCSI overlapped command condition.

- **HBA_STATUS_ERROR**  
Returned to indicate any problem with no required value.

### Related Information

- "HBA_GetEventBuffer Subroutine" on page 486, "HBA_GetFC4Statistics Subroutine" on page 487.
- "HBA_GetFCPStatistics Subroutine" on page 489, "HBA_GetFcpTargetMappingV2 Subroutine" on page 490.
- "HBA_GetPersistentBindingV2 Subroutine" on page 493, "HBA_OpenAdapterByWWN Subroutine" on page 498.
- "HBA_ScsiInquiryV2 Subroutine" on page 500, "HBA_ScsiReportLunsV2 Subroutine" on page 504.
- "HBA_SendCTPassThruV2 Subroutine" on page 506, "HBA_SendRLS Subroutine" on page 510.
- "HBA_SendRNIDV2 Subroutine" on page 512, "HBA_SendRPL Subroutine" on page 514, "HBA_SendRPS Subroutine" on page 515.
HBA_ScsiReportLnsV2 Subroutine

Purpose
Sends a SCSI REPORT LUNS command to Logical Unit Number 0 of a remote end port.

Syntax

```c
HBA_STATUS HBA_ScsiReportLUNsV2(
    HBA_HANDLE handle,
    HBA_WWN hbaPortWWN,
    HBA_WWN discoveredPortWWN,
    void *pRspBuffer,
    HBA_UINT32 *pRspBufferSize,
    HBA_UINT8 *pScsiStatus,
    void *pSenseBuffer,
    HBA_UINT32 *pSenseBufferSize
);
```

Description
The HBA_ScsiReportLnsV2 function shall send a SCSI REPORT LUNS command to Logical Unit Number 0 of a remote end port.

A SCSI command is never sent to an end port that does not have SCSI target functionality. If sending a SCSI command causes a SCSI overlapped command condition with a correctly operating target, the command will not be sent. Proper use of tagged commands is an acceptable means of avoiding a SCSI overlapped command condition with targets that support tagged commands.

Parameters

- **handle**: A handle to an open HBA through which the SCSI REPORT LUNS command is issued.
- **hbaPortWWN**: The Port Name for a local HBA end port through which the SCSI REPORT LUNS command is issued.
- **discoveredPortWWN**: The Port Name for an end port to which the SCSI REPORT LUNS command is sent.
- **pRspBuffer**: Pointer to a buffer to receive the SCSI REPORT LUNS command response.
- **pRspBufferSize**: Pointer to the size in bytes of the buffer to receive the SCSI REPORT LUNS command response.
- **pScsiStatus**: Pointer to a buffer to receive SCSI status.
- **pSenseBuffer**: Pointer to a buffer to receive SCSI sense data.
- **pSenseBufferSize**: Pointer to the size in bytes of the buffer to receive SCSI sense data.

Return Values
The value of the HBA_ScsiReportLnsV2 function is a valid status return value that indicates the reason for completion of the requested function. HBA_STATUS_OK is returned to indicate that the complete payload of a reply to the SCSI REPORT LUNS command has been returned. A valid status return value that most closely describes the result of the function should be returned to indicate a reason with no required value.

The return values for the following parameters are as follows:

- **pRspBuffer**: Remains unchanged. If the function value is HBA_STATUS_OK, the buffer to which it points contains the response to the SCSI REPORT LUNS command.
- **pRspBufferSize**: Remains unchanged. The value of the integer to which it points is the size in bytes of the response returned by the command. This cannot exceed the size passed as an argument at this pointer.
Remains unchanged. The value of the byte to which it points is the SCSI status. If the function value is \texttt{HBA\_STATUS\_OK} or \texttt{HBA\_STATUS\_SCSI\_CHECK\_CONDITION}, the value of the SCSI status can be interpreted based on the SCSI spec. A SCSI status of \texttt{HBA\_STATUS\_OK} indicates that a SCSI response is in the response buffer. A SCSI status of \texttt{HBA\_STATUS\_SCSI\_CHECK\_CONDITION} indicates that no value is stored in the response, and the sense buffer contains failure information if available.

Remains unchanged. If the function value is \texttt{HBA\_STATUS\_SCSI\_CHECK\_CONDITION}, the buffer to which it points contains the sense data for the command.

Remains unchanged. The value of the integer to which it points is the size in bytes of the sense information returned by the command. This cannot exceed the size passed as an argument at this pointer.

Error Codes

\begin{itemize}
  \item \texttt{HBA\_STATUS\_ERROR\_ILLEGAL\_WWN} \hfill The HBA referenced by \texttt{handle} does not contain an end port with Port Name \texttt{hbaPortWWN}.
  \item \texttt{HBA\_STATUS\_ERROR\_NOT\_A\_TARGET} \hfill The identified remote end port does not have SCSI Target functionality.
  \item \texttt{HBA\_STATUS\_ERROR\_TARGET\_BUSY} \hfill Unable to send the requested command without causing a SCSI overlapped command condition.
  \item \texttt{HBA\_STATUS\_ERROR} \hfill Returned to indicate any problem with no required value.
\end{itemize}

Related Information


HBA\_SendCTPassThru Subroutine

Purpose

Sends a CT pass through frame.

Library

Common Host Bus Adapter Library (libHBAAPI.a)

Syntax

\begin{verbatim}
#include <sys/hbaapi.h>

HBA\_STATUS\ HBA\_SendCTPassThru (handle, pReqBuffer, ReqBufferSize, pRspBuffer, RspBufferSize);

\end{verbatim}

Description

The \texttt{HBA\_SendCTPassThru} subroutine sends a CT pass through frame to a fabric connected to the specified handle. The CT frame is routed in the fabric according to the GS\_TYPE field in the CT frame.
Parameters

handle HBA_HANDLE to an open adapter.
pReqBuffer Pointer to a buffer that contains the CT request.
ReqBufferSize Size of the request buffer.
pRspBuffer Pointer to a buffer that receives the response of the command.
RspBufferSize Size of the response buffer.

Return Values
If successful, HBA_STATUS_OK is returned, and the pRspBuffer parameter points to the CT response.

Error Codes
If the adapter specified by the handle parameter is connected to an arbitrated loop, the HBA_SendCTPassThru subroutine returns HBA STATUS_ERROR_NOT_SUPPORTED. This subroutine is only valid when connected to a fabric.

Related Information
Special Files in AIX 5L Version 5.3 Files Reference describes specific qualities of the files that define devices.

HBA_SendCTPassThruV2 Subroutine

Purpose
Sends a CT request payload.

Syntax
HBA_STATUS HBA_SendCTPassThruV2(  
    HBA_HANDLE handle,  
    HBA_WWN hbaPortWWN,  
    void *pReqBuffer,  
    HBA_UINT32 *ReqBufferSize,  
    void *pRspBuffer,  
    HBA_UINT32 *pRspBufferSize);  

Description
The HBA_SendCTPassThruV2 function sends a CT request payload. An HBA should decode this CT_IU request by, routing the CT frame in a fabric according to the GS_TYPE field within the CT frame.

Parameters

handle A handle to an open HBA through which the CT request is issued.
hbaPortWWN The Port Name for a local HBA Nx_Port through which the CT request is issued.
pReqBuffer Pointer to a buffer containing the full CT payload, including the CT header, to be sent with byte ordering.
ReqBufferSize The size of the full CT payload, including the CT header, in bytes.
pRSPBuffer Pointer to a buffer for the CT response.
pRSPBufferSize Pointer to the size in bytes of the buffer for the CT response payload.
**Return Values**

The value of the `SendCTPassThruV2` function is a valid status return value that indicates the reason for completion of the requested function. `HBA_STATUS_OK` is returned to indicate that the complete reply to the CT `Passthru` command has been returned. A valid status return value that most closely describes the result of the function should be returned to indicate a reason with no required value.

The return values for the following parameters are as follows:

- **pRspBuffer**: Remains unchanged. The buffer to which it points contains the CT response payload, including the CT header received in response to the frame sent, with byte ordering. If the size of the actual response exceeds the size of the response buffer, trailing data is truncated from the response so that the returned data equals the size of the buffer.

- **pRspBufferSize**: Remains unchanged. The value of the integer to which it points is set to the size (in bytes) of the actual response data.

**Error Codes**

- **HBA_STATUS_ERROR_ILLEGAL_WWN**: The HBA referenced by `handle` does not contain an `Nx_Port` with Port Name `hbaPortWWN`.

- **HBA_STATUS_ERROR**: Returned to indicate any problem with no required value.

**Related Information**

[HBA_GetEventBuffer Subroutine](#) on page 486, [HBA_GetFC4Statistics Subroutine](#) on page 487, [HBA_GetFCPStatistics Subroutine](#) on page 489, [HBA_GetFcpTargetMappingV2 Subroutine](#) on page 490, [HBA_GetPersistentBindingV2 Subroutine](#) on page 493, [HBA_OpenAdapterByWWN Subroutine](#) on page 498, [HBA_ScsiInquiryV2 Subroutine](#) on page 500, [HBA_ScsiReadCapacityV2 Subroutine](#) on page 502, [HBA_ScsiReportLunsV2 Subroutine](#) on page 504, [HBA_SendRLS Subroutine](#) on page 510, [HBA_SendRNIDV2 Subroutine](#) on page 512, [HBA_SendRPL Subroutine](#) on page 514, [HBA_SendRPS Subroutine](#) on page 515

**HBA_SendReadCapacity Subroutine**

**Purpose**

Sends a **SCSI READ CAPACITY** command to a Fibre Channel port.

**Library**

Common Host Bus Adapter Library (libHBAAPL.a)

**Syntax**

```c
#include <sys/hbaapi.h>

HBA_STATUS HBA_SendReadCapacity (handle, portWWN, fcLUN, pRspBuffer, RspBufferSize, pSenseBuffer, SenseBufferSize)

HBA_HANDLE handle;
HBA_WWN portWWN;
HBA_UINT64 fcLUN;
void *pRspBuffer;
HBA_UINT32 RspBufferSize;
void *pSenseBuffer;
HBA_UINT32 SenseBufferSize;
```
Description

The HBA_SendReadCapacity subroutine sends a SCSI READ CAPACITY command to the Fibre Channel port connected to the handle parameter and specified by the portWWN and fcLUN parameters.

Parameters

- **handle**: HBA_HANDLE to an open adapter.
- **portWWN**: Port world-wide name of an adapter.
- **fcLUN**: Fibre Channel LUN to send the SCSI READ CAPACITY command to.
- **pRspBuffer**: Pointer to a buffer that receives the response of the command.
- **RspBufferSize**: Size of the response buffer.
- **pSenseBuffer**: Pointer to a buffer that receives sense information.
- **SenseBufferSize**: Size of the sense buffer.

Return Values

If successful, HBA_STATUS_OK is returned and the pRspBuffer parameter points to the response to the READ CAPACITY command. If an error occurs, HBA_STATUS_ERROR is returned.

Error Codes

If the portWWN value is not a valid world-wide name connected to the specified handle, HBA_STATUS_ERROR_ILLEGAL_WWN is returned. On any other types of failures, HBA_STATUS_ERROR is returned.

Related Information

The "HBA_SendScsiInquiry Subroutine" on page 516.

Special Files in AIX 5L Version 5.3 Files Reference describes specific qualities of the files that define devices.

HBA_SendReportLUNs Subroutine

Purpose

Sends a SCSI REPORT LUNs command to a remote port of the end device.

Library

Common Host Bus Adapter Library (libHBAAPI.a)

Syntax

```c
#include <sys/hbaapi.h>

HBA_STATUS HBA_SendReportLUNs ( handle, PortWWN, pRspBuffer, RspBufferSize, pSenseBuffer, SenseBufferSize);
```

Description

The HBA_SendReportLUNs subroutine sends a SCSI REPORT LUNs command through a call to ioctl with the SCIOLCMD operation as its argument. The arg parameter for the SCIOLCMD operation is the address of a scsi_iocmd structure. This structure is defined in the /usr/include/sys/scsi_buf.h file. The
The *scsi_ioctl* parameter block allows the caller to select the SCSI and LUN IDs to be queried. The caller also specifies the SCSI command descriptor block area, command length (SCSI command block length), the time-out value for the command, and a *flags* field.

If successful, the report LUNs data is returned in *pRspBuffer*. The returned report LUNs data must be examined to see if the requested LUN exists.

**Parameters**

- **handle** Specifies the open file descriptor obtained from a successful call to the *open* subroutine.
- **PortWWN** Specifies the world wide name or port name of the target device.
- **pRspBuffer** Points to a buffer containing the requested instruction for a send/read capacity request to an open adapter.
- **RspBufferSize** Specifies the size of the buffer to the *pRspBuffer* parameter.
- **pSenseBuffer** Points to a buffer containing the data returned from a send/read capacity request to an open adapter.
- **SenseBufferSize** Specifies the size of the buffer to the *pSenseBuffer* parameter.

**Return Values**

Upon successful completion, the *HBA_SendReportLUNs* subroutine returns a buffer in bytes containing the SCSI report of LUNs, a buffer containing the SCSI sense data, and a value of HBA_STATUS_OK, or a value of 0.

If unsuccessful, an empty buffer for the SCSI report of LUNs, a response buffer containing the failure, and a value of HBA_STATUS_ERROR, or a value of 1 is returned.

**Error Codes**

The Storage Area Network Host Bus Adapter API subroutines return the following error codes:

- **HBA_STATUS_OK** A value of 0 on successful completion.
- **HBA_STATUS_ERROR** A value of 1 if an error occurred.
- **HBA_STATUS_ERROR_INVALID_HANDLE** A value of 3 if there was an invalid file handle.
- **HBA_STATUS_ERROR_ILLEGAL_WWN** A value of 5 if the world wide name was not recognized.
- **HBA_STATUS_SCSI_CHECK_CONDITION** A value of 9 if a SCSI Check Condition was reported.

**Related Information**

[HBA_SendScsiInquiry Subroutine](#) on page 516, [HBA_SendReadCapacity Subroutine](#) on page 507, [HBA_SendCTPassThru Subroutine](#) on page 505, [HBA_SendRNID Subroutine](#) on page 511, [HBA_SetRNIDMgmtInfo Subroutine](#) on page 518, and [HBA_GetRNIDMgmtInfo Subroutine](#) on page 495.

SCSI Adapter Device Driver in *AIX 5L Version 5.3 Technical Reference: Kernel and Subsystems Volume 2*.

Special Files in *AIX 5L Version 5.3 Files Reference*.

SCSI Subsystem Overview, A Typical Initiator-Mode SCSI Driver Transaction Sequence, Required SCSI Adapter Device Driver ioctl Commands, Understanding the Execution of Initiator I/O Requests, SCSI Error Recovery, and Understanding the *sc_buf* Structure in *AIX 5L Version 5.3 Kernel Extensions and Device Support Programming Concepts.*
HBA_SendRLS Subroutine

Purpose
Issues a Read Link Error Status Block (RLS) Extended Link Service through the specified HBA end port.

Syntax
```c
HBA_STATUS HBA_SendRLS (  
    HBA_HANDLE handle,  
    HBA_WWN hbaPortWWN,  
    HBA_WWN destWWN,  
    void *pRspBuffer,  
    HBA_UINT32 *pRspBufferSize
);```

Description
The HBA_SendRLS function issues a Read Link Error Status Block (RLS) Extended Link Service through the specified HBA end port to request a specified remote FC_Port to return the Link Error Status Block associated with the destination Port Name.

Parameters
- `handle`: A handle to an open HBA through which the ELS is sent.
- `hbaPortWWN`: Port Name of the local HBA end port through which the ELS is sent.
- `destWWN`: Port Name of the remote FC_Port to which the ELS is sent.
- `pRspBuffer`: Pointer to a buffer to receive the ELS response.
- `pRspBufferSize`: Pointer to the size in bytes of `pRspBuffer`. A size of 28 is sufficient for the largest response.

Return Values
The value of the HBA_SendRLS function is a valid status return value that indicates the reason for completion of the requested function. HBA_STATUS_OK is returned to indicate that the complete LS_ACC to the RLS ELS has been returned. A valid status return value that most closely describes the result of the function should be returned to indicate a reason with no required value.

The return values for the following parameters are as follows:
- `pRspBuffer`: Remains unchanged. The buffer to which it points contains the payload data from the RLS Reply. Note that if the ELS was rejected, this is the LS_RJT payload. If the size of the reply payload exceeds the size specified in the `pRspBufferSize` parameter at entry to the function, the returned data is truncated to the size specified in the argument.
- `pRspBufferSize`: Remains unchanged. The value of the integer to which it points contains the size in bytes of the complete ELS reply payload. This can exceed the size specified as an argument. This indicates that the data in `pRspBuffer` has been truncated.

Error Codes
- HBA_STATUS_ERROR_ELS_REJECT: The RNID ELS was rejected by the destination FC_Port.
- HBA_STATUS_ERROR_ILLEGAL_WWN: The HBA referenced by `handle` does not contain an end port with Port Name `hbaPortWWN`.
- HBA_STATUS_ERROR: Returned to indicate any problem with no required value.
HBA_SendRNID Subroutine

Purpose
Sends an RNID command through a call to SCIOLPAYLD to a remote port of the end device.

Library
Common Host Bus Adapter Library (libHBAAPI.a)

Syntax
```
#include <sys/hbaapi.h>

HBA_STATUS HBA_SendRNID (handle, wwn, wwnotype, pRspBuffer, RspBufferSize)
HBA_HANDLE handle;
HBA_WWN wwn;
HBA_WWNTYPE wwnotype;
void *pRspBuffer;
HBA_UINT32 RspBufferSize;
```

Description
The HBA_SendRNID subroutine sends a SCSI RNID command with the Node Identification Data Format set to indicate the default Topology Discovery format. This is done through a call to ioctl with the SCIOLPAYLD operation as its argument. The arg parameter for the SCIOLPAYLD operation is the address of an scsi_trans_payld structure. This structure is defined in the /usr/include/sys/scsi_buf.h file. The scsi_trans_payld parameter block allows the caller to select the SCSI and LUN IDs to be queried. In addition, the caller must specify the fcph_rnid_payld_t structure to hold the command and the topology format for SCIOLPAYLD. The structure for the fcph_rnid_payld_t structure is defined in the /usr/include/sys/fcph.h file.

If successful, the RNID data is returned in pRspBuffer. The returned RNID data must be examined to see if the requested information exists.

Parameters

- **handle**: Specifies the open file descriptor obtained from a successful call to the open subroutine.
- **wwn**: Specifies the world wide name or port name of the target device.
- **wwntype**: Specifies the type of the world wide name or port name of the target device.
- **pRspBuffer**: Points to a buffer containing the requested instruction for a send/read capacity request to an open adapter.
- **RspBufferSize**: Specifies the size of the buffer to the pRspBuffer parameter.

Return Values
Upon successful completion, the HBA_SendRNID subroutine returns a buffer in bytes containing the SCSI RNID data and a value of HBA_STATUS_OK, or a value of 0. If unsuccessful, an empty buffer for the SCSI RNID and a value of HBA_STATUS_ERROR, or a value of 1 is returned.
Error Codes
The Storage Area Network Host Bus Adapter API subroutines return the following error codes:

- **HBA_STATUS_OK**  
  A value of 0 on successful completion.
- **HBA_STATUS_ERROR**  
  A value of 1 if an error occurred.
- **HBA_STATUS_ERROR_NOT_SUPPORTED**  
  A value of 2 if the function is not supported.
- **HBA_STATUS_ERROR_INVALID_HANDLE**  
  A value of 3 if there was an invalid file handle.
- **HBA_STATUS_ERROR_ILLEGAL_WWN**  
  A value of 5 if the world wide name was not recognized.

Related Information
- “HBA_SendScsiInquiry Subroutine” on page 516
- “HBA_SendReadCapacity Subroutine” on page 507
- “HBA_SendCTPassThru Subroutine” on page 505
- “HBA_SendReportLUNs Subroutine” on page 508
- “HBA_SetRNIDMgmtInfo Subroutine” on page 518
- “HBA_GetRNIDMgmtInfo Subroutine” on page 495


Special Files in AIX 5L Version 5.3 Files Reference.


### HBA_SendRNIDV2 Subroutine

**Purpose**
Issues an RNID ELS to another FC_Port requesting a specified Node Identification Data Format.

**Syntax**
```c
HBA_STATUS HBA_SendRNIDV2(
    HBA_HANDLE handle,
    HBA_WWN hbaPortWWN,
    HBA_WWN destWWN,
    HBA_UINT32 destFCID,
    HBA_UINT32 NodeIdDataFormat,
    void * pRspBuffer,
    HBA_UINT32 * pRspBufferSize
);
```

**Description**
The `HBA_SendRNIDV2` function issues an RNID ELS to another FC_Port requesting a specified Node Identification Data Format.

The `destFCID` parameter can be set to allow the RNID ELS to be sent to an FC_Port that might not be registered with the name server. If `destFCID` is set to x’00 00 00’, the parameter is ignored. If `destFCID` is not 0, the HBA API library verifies that the `destWWN`/`destFCID` pair match in order to limit visibility that can violate scoping mechanisms (such as soft zoning):

- If the `destWWN`/`destFCID` pair matches an entry in the discovered ports table, the RNID is sent.
- If there is no entry in the discovered ports table for the `destWWN` or `destFCID`, the RNID is sent.
- If there is an entry in the discovered ports table for the `destWWN`, but the `destFCID` does not match, then the request is rejected.
On completion of the **HBA_SendRNIDV2**, if the Common Identification Data Length is nonzero in the RNID response, the API library compares the **N_Port_Name** in the Common Identification Data of the RNID response with `destWWN` and fails the operation without returning the response data if they do not match. If the Common Identification Data Length is 0 in the RNID response, this test is omitted.

### Parameters

- **handle**: A handle to an open HBA through which the ELS is sent.
- **hbaPortWWN**: Port Name of the local HBA end port through which the ELS is sent.
- **destWWN**: Port Name of the remote FC_Port to which the ELS is sent.
- **destFCID**: Address identifier of the destination to which the ELS is sent if `destFCID` is nonzero. `destFCID` is ignored if `destFCID` is 0.
- **NodeIdDataFormat**: Valid value for Node Identification Data Format.
- **pRSPBuffer**: Pointer to a buffer to receive the ELS response.
- **pRSPBufferSize**: Pointer to the size in bytes of `pRspBuffer`.

### Return Values

The value of the **HBA_SendRNIDV2** function is a valid status return value that indicates the reason for completion of the requested function. **HBA_STATUS_OK** is returned to indicate that the complete LS_ACC to the RNID ELS has been returned. A valid status return value that most closely describes the result of the function should be returned to indicate a reason with no required value.

The return values for the following parameters are as follows:

- **pRspBuffer**: Remains unchanged. The buffer to which it points contains the payload data from the RNID Reply. Note that if the ELS was rejected, this is the LS_RJT payload. If the size of the reply payload exceeds the size specified in the `pRspBufferSize` parameter at entry to the function, the returned data is truncated to the size specified in the argument.

- **pRspBufferSize**: Remains unchanged. The value of the integer to which it points contains the size in bytes of the complete ELS reply payload. This can exceed the size specified as an argument. This indicates that the data in `pRspBuffer` has been truncated.

### Error Codes

- **HBA_STATUS_ERROR_ELS_REJECT**: The RNID ELS was rejected by the destination end port.
- **HBA_STATUS_ERROR_ILLEGAL_WWN**: The HBA referenced by `handle` does not contain an end port with Port Name `hbaPortWWN`.
- **HBA_STATUS_ERROR_ILLEGAL_FCID**: The `destWWN`/`destFCID` pair conflicts with a discovered Port Name/address identifier pair known by the HBA referenced by `handle`.
- **HBA_STATUS_ERROR_ILLEGAL_FCID**: The **N_Port_Name** in the RNID response does not match the `destWWN`.
- **HBA_STATUS_ERROR**: Returned to indicate any problem with no required value.

### Related Information

- "**HBA_GetEventBuffer Subroutine**" on page 486
- "**HBA_GetFC4Statistics Subroutine**" on page 487
- "**HBA_GetFCPStatistics Subroutine**" on page 489
- "**HBA_GetFcpTargetMappingV2 Subroutine**" on page 490
- "**HBA_GetPersistentBindingV2 Subroutine**" on page 493
- "**HBA_OpenAdapterByWWN Subroutine**" on page 498
- "**HBA_ScsiInquiryV2 Subroutine**" on page 500
- "**HBA_ScsiReadCapacityV2 Subroutine**" on page 502
- "**HBA_ScsiReportLunsV2 Subroutine**" on page 504
- "**HBA_SendCTPassThruV2 Subroutine**" on page 506
- "**HBA_SendRHL Subroutine**" on page 510
- "**HBA_SendRPL Subroutine**" on page 514
- "**HBA_SendRPS Subroutine**" on page 515
HBA_SendRPL Subroutine

Purpose
Issues a Read Port List (RPL) Extended Link Service through the specified HBA to a specified end port or domain controller.

Syntax
```c
HBA_STATUS HBA_SendRPL(
    HBA_HANDLE handle,
    HBA_WWN hbaPortWWN,
    HBA_WWN agent_wwn,
    HBA_UINT32 agent_domain,
    HBA_UINT32 portIndex,
    void *pRspBuffer,
    HBA_UINT32 *pRspBufferSize
);
```

Description
The HBA_SendRPL function issues a Read Port List (RPL) Extended Link Service through the specified HBA to a specified end port or domain controller.

Parameters
- `handle`: A handle to an open HBA through which the ELS is sent.
- `hbaPortWWN`: Port Name of the local HBA end port through which the ELS is sent.
- `agent_wwn`: Port Name of an FC_Port that is requested to provide its list of FC_Ports if `agent_wwn` is nonzero. If `agent_wwn` is 0, it is ignored.
- `agent_domain`: Domain number and the domain controller for that domain shall be the entity that shall be requested to provide its list of FC_Ports if `agent_wwn` is 0. If `agent_wwn` is nonzero, `agent_domain` is ignored.
- `portIndex`: Index of the first FC_Port requested in the response list.
  **Note:** If the recipient has proper compliance, the index of the first FC_Port in the complete list maintained by the recipient of the request is 0.
- `pRspBuffer`: Pointer to a buffer to receive the ELS response.
- `pRspBufferSize`: Pointer to the size in bytes of `pRspBuffer`.  
  **Note:** If the responding entity has proper compliance, it truncates the list in the response to the number of FC_Ports that fit.

Return Values
The value of the HBA_SendRPL function is a valid status return value that indicates the reason for completion of the requested function. HBA_STATUS_OK is returned to indicate that the complete LS_ACC to the RPL ELS has been returned. A valid status return value that most closely describes the result of the function should be returned to indicate a reason with no required value.

The return values for the following parameters are as follows:
- `pRspBuffer`: Remains unchanged. The buffer to which it points contains the payload data from the RPL Reply. If the ELS was rejected, this is the LS_RJT payload. If the size of the reply payload exceeds the size specified in the `pRspBufferSize` parameter at entry to the function, the returned data is truncated to the size specified in the argument.
- `pRspBufferSize`: Remains unchanged. The value of the integer to which it points contains the size in bytes of the complete ELS reply payload. This can exceed the size specified as an argument. This indicates that the data in `pRspBuffer` has been truncated.  
  **Note:** Truncation is not necessary if the responding entity is of proper compliance.
Error Codes

HBA_STATUS_ERROR_ELS_REJECT
The RPL ELS was rejected by the destination end port.

HBA_STATUS_ERROR_ILLEGAL_WWN
The HBA referenced by handle does not contain an end port with Port Name hbaPortWWN.

HBA_STATUS_ERROR
Returned to indicate any problem with no required value.

Related Information

"HBA_GetEventBuffer Subroutine" on page 486, "HBA_GetFC4Statistics Subroutine" on page 487
"HBA_GetFCPStatistics Subroutine" on page 489, "HBA_GetPersistentBindingV2 Subroutine" on page 490, "HBA_OpenAdapterByWWN Subroutine" on page 498, "HBA_ScsiInquiryV2 Subroutine" on page 500, "HBA_ScsiReadCapacityV2 Subroutine" on page 502, "HBA_ScsiReportLunsV2 Subroutine" on page 504, "HBA_SendCTPassThruV2 Subroutine" on page 506, "HBA_SendRPIV2 Subroutine" on page 510, "HBA_SendRNIDV2 Subroutine" on page 512, "HBA_SendRPS Subroutine"

HBA_SendRPS Subroutine

Purpose
Issues a Read Port Status Block (RPS) Extended Link Service through the specified HBA to a specified FC_Port or domain controller.

Syntax

HBA_STATUS HBA_SendRPS (  
  HBA_HANDLE handle,  
  HBA_WWN hbaPortWWN,  
  HBA_WWN agent_wwn,  
  HBA_UINT32 agent_domain,  
  HBA_WWN object_wwn,  
  HBA_UINT32 object_port_number,  
  void *pRspBuffer,  
  HBA_UINT32 *pRspBufferSize
);

Description
The HBA_SendRPS function issues a Read Port Status Block (RPS) Extended Link Service through the specified HBA to a specified FC_Port or domain controller.

Parameters

handle A handle to an open HBA through which the ELS is sent.
hbaPortWWN Port Name of the local HBA end port through which the ELS is sent.
agent_wwn Port Name of an FC_Port that is requested to provide Port Status if agent_wwn is nonzero. agent_wwn is ignored if its value is 0.
agent_domain Domain number for the domain controller that is requested to provide Port status if agent_wwn is 0. agent_domain is ignored if agent_wwn is nonzero.
object_wwn Port Name of an FC_Port for which Port Status is returned if object_wwn is nonzero. object_wwn is ignored if its value is 0.
object_port_number Relative port number of the FC_Port for which Port Status is returned if object_wwn is 0. The relative port number is defined in a vendor-specific manner within the entity to which the request is sent. object_port_number is ignored if object_wwn is nonzero.
pRspBuffer Pointer to a buffer to receive the ELS response.
pRSPBufferSize Pointer to the size in bytes of pRspBuffer. A size of 56 is sufficient for the largest response.
Return Values
The value of the HBA_SendRPS function is a valid status return value that indicates the reason for completion of the requested function. HBA_STATUS_OK is returned to indicate that the complete LS_ACC to the RPS ELS has been returned. A valid status return value that most closely describes the result of the function should be returned to indicate a reason with no required value.

The return values for the following parameters are as follows:

\begin{itemize}
\item \textit{pRspBuffer} Remains unchanged. The buffer to which it points contains the payload data from the RPS Reply. If the ELS was rejected, this is the LS_RJT payload. If the size of the reply payload exceeds the size specified in the \textit{pRspBufferSize} parameter at entry to the function, the returned data is truncated to the size specified in the argument.
\item \textit{pRspBufferSize} Remains unchanged. The value of the integer to which it points contains the size in bytes of the complete ELS reply payload. This can exceed the size specified as an argument. This indicates that the data in \textit{pRspBuffer} has been truncated.
\end{itemize}

Error Codes

\begin{itemize}
\item HBA_STATUS_ERROR_ELS_REJECT The RPS ELS was rejected by the destination end port.
\item HBA_STATUS_ERROR_ILLEGAL_WWN The HBA referenced by \textit{handle} does not contain an end port with Port Name \textit{hbaPortWWN}.
\item HBA_STATUS_ERROR Returned to indicate any problem with no required value.
\end{itemize}

Related Information


HBA_SendScsiInquiry Subroutine

Purpose
Sends a SCSI device inquiry command to a remote port of the end device.

Library
Common Host Bus Adapter Library (libHBAAPI.a)

Syntax

\begin{verbatim}
#include <sys/hbaapi.h>

HBA_STATUS HBA_SendScsiInquiry (HBA_HANDLE handle, HBA_WWN PortWWN, HBA_UINT64 fcLUN, HBA_UINT8 EVPD, HBA_UINT32 PageCode, void *pRspBuffer, HBA_UINT32 RspBufferSize, HBA_UINT32 SenseBufferSize, void *pSenseBuffer);

HBA_HANDLE handle;
HBA_WWN PortWWN;
HBA_UINT64 fcLUN;
HBA_UINT8 EVPD;
HBA_UINT32 PageCode;
void *pRspBuffer;
HBA_UINT32 RspBufferSize;
void *pSenseBuffer;
HBA_UINT32 SenseBufferSize;
\end{verbatim}
Description

The **HBA_SendScsiInquiry** subroutine sends a **SCSI INQUIRY** command through a call to **ioctl** with the **SCIOLINQU** operation as its argument. The **arg** parameter for the **SCIOLINQU** operation is the address of an **scsi_inquiry** structure. This structure is defined in the **/usr/include/sys/scsi_buf.h** file. The **scsi_inquiry** parameter block allows the caller to select the SCSI and LUN IDs to be queried. If successful, the inquiry data is returned in the **pRspBuffer** parameter. Successful completion occurs if a device responds at the requested SCSI ID, but the returned inquiry data must be examined to see if the requested LUN exists.

Parameters

- **handle**
  Specifies the open file descriptor obtained from a successful call to the **open** subroutine.
- **PortWWN**
  Specifies the world wide name or port name of the target device.
- **fcLUN**
  Specifies the fcLUN.
- **EVPD**
  Specifies the value for the EVPD bit. If the value is 1, the Vital Product Data page code will be specified by the **PageCode** parameter.
- **PageCode**
  Specifies the Vital Product Data that is to be requested if the EVPD parameter is set to 1.
- **pRspBuffer**
  Points to a buffer containing the requested instruction for a send/read capacity request to an open adapter. The size of this buffer must not be greater than 255 bytes.
- **RspBufferSize**
  Specifies the size of the buffer to the **pRspBuffer** parameter.
- **pSenseBuffer**
  Points to a buffer containing the data returned from a send/read capacity request to an open adapter.
- **SenseBufferSize**
  Specifies the size of the buffer to the **pSenseBuffer** parameter.

Return Values

Upon successful completion, the **HBA_SendScsiInquiry** subroutine returns a buffer in bytes containing the SCSI inquiry, a buffer containing the SCSI sense data, and a value of **HBA_STATUS_OK**, or a value of 0.

If unsuccessful, an empty buffer for the SCSI inquiry, a response buffer containing the failure, and a value of **HBA_STATUS_ERROR**, or a value of 1 is returned.

Error Codes

The Storage Area Network Host Bus Adapter API subroutines return the following error codes:

- **HBA_STATUS_OK**
  A value of 0 on successful completion.
- **HBA_STATUS_ERROR**
  A value of 1 if an error occurred.
- **HBA_STATUS_ERROR_INVALID_HANDLE**
  A value of 3 if there was an invalid file handle.
- **HBA_STATUS_ERROR_ARG**
  A value of 4 if there was a bad argument.
- **HBA_STATUS_ERROR_ILLEGAL_WWN**
  A value of 5 if the world wide name was not recognized.
- **HBA_STATUS_SCSI_CHECK_CONDITION**
  A value of 9 if a SCSI Check Condition was reported.

Related Information

- “**HBA_SendReportLUNs Subroutine**” on page 508
- “**HBA_SendCapacity Subroutine**” on page 507
- “**HBA_SendCTPassThru Subroutine**” on page 505
- “**HBA_SendRNID Subroutine**” on page 511
- “**HBA_SetRNIDMgmtInfo Subroutine**” on page 518
- “**HBA_GetRNIDMgmtInfo Subroutine**” on page 495

**SCSI Adapter Device Driver** in **AIX 5L Version 5.3 Technical Reference: Kernel and Subsystems Volume 2**.

**Special Files** in **AIX 5L Version 5.3 Files Reference**.
HBA_SetRNIDMgmtInfo Subroutine

Purpose
Sends a SCSI SET RNID command to a remote port of the end device.

Library
Common Host Bus Adapter Library (libHBAAPI.a)

Syntax
#include <sys/hbaapi.h>

HBA_STATUS HBA_SetRNIDMgmtInfo (HBA_HANDLE handle, HBA_MGMTINFO info);

Description
The HBA_SetRNIDMgmtInfo subroutine sends a SCSI SET RNID (Request Node Identification Data) command with the SCIOLCHBA operation as its argument. This is done through a call to ioctl. The arg parameter for the SCIOLCHBA operation is the address of a scsi_chba structure. This structure is defined in the /usr/include/sys/scsi_buf.h file. The scsi_chba parameter block allows the caller to select the SET RNID command to be sent to the adapter. The info structure stores the RNID data to be set. The info structure is defined in the /usr/include/sys/hbaapi.h file. The structure includes:

- wwn
- unittype
- PortId
- NumberOfAttachedNodes
- IPVersion
- UDPPort
- IPAddress
- reserved
- TopologyDiscoveryFlags

If successful, the SET RNID data in info is sent to the adapter.

Parameters
handle Specifies the open file descriptor obtained from a successful call to the open subroutine.
info Specifies the structure containing the information to be set or received from the RNID command

Return Values
Upon successful completion, the HBA_SetRNIDMgmtInfo subroutine returns a value of HBA_STATUS_OK, or a value of 0. If unsuccessful, a value of HBA_STATUS_ERROR, or a 1 is returned.

Error Codes
The Storage Area Network Host Bus Adapter API subroutines return the following error codes:
HBA_STATUS_OK
A value of 0 on successful completion.
HBA_STATUS_ERROR
A value of 1 if an error occurred.
HBA_STATUS_ERROR_INVALID_HANDLE
A value of 3 if there was an invalid file handle.

Related Information
“HBA_SendScsiInquiry Subroutine” on page 516, “HBA_SendReadCapacity Subroutine” on page 507, “HBA_SendCTPassThru Subroutine” on page 505, “HBA_SendReportLUNs Subroutine” on page 508, “HBA_SendRNID Subroutine” on page 511, and “HBA_GetRNIDMgmtInfo Subroutine” on page 495.


Special Files in AIX 5L Version 5.3 Files Reference.


hpmInit, f_hpminit, hpmStart, f_hpmstart, hpmStop, f_hpmstop, hpmTstart, f_hpmtstart, hpmTstop, f_hpmtstop, hpmGetTimeAndCounters, f_hpmgettimeandcounters, hpmGetCounters, f_hpmgetcounters, hpmTerminate, and f_hpmtterminate Subroutine

Purpose
Provides application instrumentation for performance monitoring.

Library
HPM Library (libhpm.a)

HPM Library (libhpm.a) includes four additional subroutines for threaded applications.

Syntax
#include <libhpm.h>

void hpmInit(int taskID, char *progName);
void f_hpminit(int taskID, char *progName);
void hpmStart(int instID, char *label);
void f_hpmstart(int instID, char *label);
void hpmStop(int instID);
void f_hpmstop(int instID);

(libhpmer only)
void hpmTstart(int instID, char *label);
void f_hpmtstart(int instID, char *label);
(libhpmer only)
void hpmTstop(int instID);
void f_hpmtstop(int instID);

void hpmGetTimeAndCounters(int numCounters, double *time, long long *values);
void f_hpmgettimeandcounters(int numCounters, double *time, long long *values);
void hpmGetCounters(long long *values);
void f_hpmgetcounters(long long *values);

void hpmTerminate(int taskID);
void f_hpmterminate(int taskID);

Description
The hpmInit and f_hpminit subroutines initialize tasks specified by the taskID and progName parameters.

The hpmStart and f_hpmstart subroutines debut an instrumented code segment. If more than 100 instrumented sections are required, the HPM_NUM_INST_PTS environment variable can be set to indicate the higher value and instID should be less than this value.

The hpmStop and f_hpmstop subroutines indicate the end of the instrumented code segment instID. For each call to hpmStart and f_hpmstart, there should be a corresponding call to hpmStop and f_hpmstop with the matching instID.

The hpmTstart and f_hpmtstart subroutines perform the same function as hpmStart and f_hpmstart, but are used in threaded applications.

The hpmTstop and f_hpmtstop subroutines perform the same function as hpmStop and f_hpmstop, but are used in threaded applications.

The hpmGetTimeAndCounters and f_hpmgettimeandcounters subroutines are used to return the time in seconds and the accumulated counts since the call to hpmInit or f_hpminit.

The hpmGetCounters and f_hpmgetcounters subroutines return all the accumulated counts since the call to hpmInit or f_hpminit. To minimize intrusion and overhead, the hpmGetCounters and f_hpmgetcounters subroutines do not perform any check on the size of the values array. The number of counters can be obtained from the pm_info2_t.maxpmcs structure element supplied by pm_initialize or by using the pmlist -s command. Alternatively, the application can use the current maximum value of 8.

The hpmTerminate and f_hpmterminate subroutines end the taskID and generate the output. Applications that do not call hpmTerminate or f_hpmterminate, do not generate performance information.

A summary report for each task is written by default in the progName_pid_taskID.hpm file, where progName is the second parameter to the hpmInit subroutine. If progName contains a space or tab character, or is otherwise invalid, a diagnostic message is written to stderr and the library exits with an error to avoid further problems.

The output file name can be defined with the HPM_OUTPUT_NAME environment flag. The libhpm still adds the file name suffix _taskID.hpm for the performance files. By using this environment variable, you can specify the date and time for the output file name. For example:
MYDATE=$(date +%Y%m%d:%H%M%S)
export HPM_OUTPUT_NAME=myprogram_$MYDATE

where the output file for task 27 will have the following name:
myprogram_yyyymmdd:HHMMSS_0027.hpm

The GUI and .viz output is deactivated by default. The aligned set of performance files named progName_pid_taskID.viz or HPM_OUTPUT_NAME_taskID.viz will not be generated (the generation of the .viz file was previously activated by default and avoided with the HPM_VIZ_OUTPUT = FALSE environment variable).
Parameters

- **instID**: Specifies the instrumented section ID as an integer value greater than 0 and less than 100.
- **label**: Specifies a label with a character string.
- **numCounters**: Specifies an integer value that indicates the number of counters to be accessed.
- **progName**: Specifies a program name using a character string label.
- **taskID**: Specifies a node ID with an integer value.
- **time**: Specifies a double precision float.
- **values**: Specifies an array of type `long long` of size `numCounters`.

Execution Environment

Functionality provided by the `libhpm` library is dependent upon corresponding functions in the `libpmapi` and `libm` libraries. Therefore, the `-lpmapi -lm` link flags must be specified when compiling applications.

Return Values

No return values are defined.

Error Codes

Upon failure, these `libhpm` subroutines either write error messages explicitly to `stderr` or use the PMAPI `pm_error` function. The `pm_error` function is called following an error return from any of the following subroutines:

- `pm_init_private`
- `pm_set_program_mygroup`
- `pm_stop_mygroup`
- `pm_get_data_mygroup`
- `pm_start_mygroup`
- `pm_stop_mythread`
- `pm_get_data_mythread`
- `pm_start_mythread`
- `pm_get_data_mythread`

Diagnostic messages are explicitly written to `stderr` or `stdout` in the following situations:

- `pm_cycles` or `gettimeofday` returns an error
- The value of the `instID` parameter is invalid
- An event set is out of range
- The `libHPMevents` file or `HPM_flags.env` file has an incorrect format
- There are internal errors.

Error messages that are not fatal are written to `stdout` or `stderr` with the text `WARNING`.

Related Information

The "getrusage, getrusage64, times, or vtimes Subroutine" on page 423, "pm_initialize Subroutine" on page 1071.

hsearch, hcreate, or hdestroy Subroutine

Purpose
Manages hash tables.

Library
Standard C Library (libc.a)

Syntax
#include <search.h>

ENTRY *hsearch ( Item, Action)
ENTRY Item;
Action Action;

int hcreate (NumberOfElements)
size_t NumberOfElements;
void hdestroy ( )

Description
Attention: Do not use the hsearch, hcreate, or hdestroy subroutine in a multithreaded environment.

The hsearch subroutine searches a hash table. It returns a pointer into a hash table that indicates the location of the given item. The hsearch subroutine uses open addressing with a multiplicative hash function.

The hcreate subroutine allocates sufficient space for the table. You must call the hcreate subroutine before calling the hsearch subroutine. The NumberOfElements parameter is an estimate of the maximum number of entries that the table will contain. This number may be adjusted upward by the algorithm in order to obtain certain mathematically favorable circumstances.

The hdestroy subroutine deletes the hash table. This action allows you to start a new hash table since only one table can be active at a time. After the call to the hdestroy subroutine, the data can no longer be considered accessible.

Parameters

Item
Identifies a structure of the type ENTRY as defined in the search.h file. It contains two pointers:

Item.key
Points to the comparison key. The key field is of the char type.

Item.data
Points to any other data associated with that key. The data field is of the void type.

Pointers to data types other than the char type should be declared to pointer-to-character.
**Action**

Specifies the value of the *Action* enumeration parameter that indicates what is to be done with an entry if it cannot be found in the table. Values are:

- **ENTER** Enters the value of the *Item* parameter into the table at the appropriate point. If the table is full, the **hsearch** subroutine returns a null pointer.

- **FIND** Does not enter the value of the *Item* parameter into the table. If the value of the *Item* parameter cannot be found, the **hsearch** subroutine returns a null pointer. If the value of the *Item* parameter is found, the subroutine returns the address of the item in the hash table.

**NumberOfElements**

Provides an estimate of the maximum number of entries that the table contains. Under some circumstances, the **hcreate** subroutine may actually make the table larger than specified.

**Return Values**

The **hcreate** subroutine returns a value of 0 if it cannot allocate sufficient space for the table.

**Related Information**

The **bsearch** (**bsearch Subroutine** on page 123) subroutine, **lsearch** (**lsearch or lfind Subroutine** on page 754) subroutine, **malloc** (**malloc, free, realloc, calloc, mallopt, mallinfo, mallinfo_heap, alloca, valloc, or posix_memalign Subroutine** on page 769) subroutine, **strcmp** subroutine, **tsearch** subroutine.

[Searching and Sorting Example Program and Subroutines Overview](index.html) in **AIX 5L Version 5.3 General Programming Concepts: Writing and Debugging Programs**.

---

**hypot, hypotf, or hypotl Subroutine**

**Purpose**

Computes the Euclidean distance function and complex absolute value.

**Libraries**

IEEE Math Library (**libm.a**)

System V Math Library (**libmsaa.a**)

**Syntax**

```c
#include <math.h>

double hypot (double x, double y);
float hypotf (float x, float y);
long double hypotl (long double x, long double y);
```

**Description**

The **hypot**, **hypotf** and **hypotl** subroutines compute the value of the square root of \( x^2 + y^2 \) without undue overflow or underflow.
An application wishing to check for error situations should set the \texttt{errno} global variable to zero and call \texttt{feclearexcept(FE_ALL_EXCEPT)} before calling these subroutines. Upon return, if \texttt{errno} is nonzero or \texttt{fetestexcept(FE_INVALID | FE_DIVBYZERO | FE_OVERFLOW | FE_UNDERFLOW)} is nonzero, an error has occurred.

**Parameters**

- \(x\) specifies some double-precision floating-point value.
- \(y\) specifies some double-precision floating-point value.

**Return Values**

Upon successful completion, the \texttt{hypot}, \texttt{hypotf} and \texttt{hypotl} subroutines return the length of the hypotenuse of a right-angled triangle with sides of length \(x\) and \(y\).

If the correct value would cause overflow, a range error occurs and the \texttt{hypot} and \texttt{hypotl} subroutines return the value of the macro \texttt{HUGE_VALF} and \texttt{HUGE_VALL}, respectively.

If \(x\) or \(y\) is ±Inf, +Inf is returned (even if one of \(x\) or \(y\) is NaN).

If \(x\) or \(y\) is NaN, and the other is not ±Inf, a NaN is returned.

If both arguments are subnormal and the correct result is subnormal, a range error may occur and the correct result is returned.

**Error Codes**

When using the \texttt{libm.a} (-lm) library, if the correct value overflows, the \texttt{hypot} subroutine returns a \texttt{HUGE_VAL} value.

**Note:** \texttt{(hypot(INF, value)} and \texttt{hypot(value, INF)} are both equal to \texttt{+INF} for all values, even if \texttt{value} = NaN.

When using \texttt{libmsaa.a} (-lmsaa), if the correct value overflows, the \texttt{hypot} subroutine returns \texttt{HUGE_VAL} and sets the global variable \texttt{errno} to \texttt{ERANGE}.

These error-handling procedures may be changed with the \texttt{matherr} subroutine when using the \texttt{libmsaa.a} (-lmsaa) library.

**Related Information**

- \texttt{fetestexcept Subroutine} on page 262, \texttt{fetestexcept Subroutine} on page 270, and \texttt{class, _class, finite, isnan, or unordered Subroutines} on page 167.

The \texttt{matherr} \texttt{matherr Subroutine} on page 780 subroutine, \texttt{sqrt} subroutine.

- \texttt{Subroutines Overview} in AIX 5L Version 5.3 General Programming Concepts: Writing and Debugging Programs.

- \texttt{math.h} in AIX 5L Version 5.3 Files Reference.

**iconv Subroutine**

**Purpose**

Converts a string of characters in one character code set to another character code set.
Library
The iconv Library (libiconv.a)

Syntax
#include <iconv.h>
size_t iconv (CD, InBuf, InBytesLeft, OutBuf, OutBytesLeft)
iconv_t CD;
char **OutBuf, **InBuf;
size_t *OutBytesLeft, *InBytesLeft;

Description
The iconv subroutine converts the string specified by the InBuf parameter into a different code set and returns the results in the OutBuf parameter. The required conversion method is identified by the CD parameter, which must be valid conversion descriptor returned by a previous, successful call to the iconv_open subroutine.

On calling, the InBytesLeft parameter indicates the number of bytes in the InBuf buffer to be converted, and the OutBytesLeft parameter indicates the number of bytes remaining in the OutBuf buffer that do not contain converted data. These values are updated upon return so they indicate the new state of their associated buffers.

For state-dependent encodings, calling the iconv subroutine with the InBuf buffer set to null will reset the conversion descriptor in the CD parameter to its initial state. Subsequent calls with the InBuf buffer, specifying other than a null pointer, may cause the internal state of the subroutine to be altered a necessary.

Parameters
CD Specifies the conversion descriptor that points to the correct code set converter.
InBuf Points to a buffer that contains the number of bytes in the InBytesLeft parameter to be converted.
InBytesLeft Points to an integer that contains the number of bytes in the InBuf parameter.
OutBuf Points to a buffer that contains the number of bytes in the OutBytesLeft parameter that has been converted.
OutBytesLeft Points to an integer that contains the number of bytes in the OutBuf parameter.

Return Values
Upon successful conversion of all the characters in the InBuf buffer and after placing the converted characters in the OutBuf buffer, the iconv subroutine returns 0, updates the InBytesLeft and OutBytesLeft parameters, and increments the InBuf and OutBuf pointers. Otherwise, it updates the variables pointed to by the parameters to indicate the extent to the conversion, returns the number of bytes still left to be converted in the input buffer, and sets the errno global variable to indicate the error.

Error Codes
If the iconv subroutine is unsuccessful, it updates the variables to reflect the extent of the conversion before it stopped and sets the errno global variable to one of the following values:

EILSEQ Indicates an unusable character. If an input character does not belong to the input code set, no conversion is attempted on the unusable on the character. In InBytesLeft parameters indicates the bytes left to be converted, including the first byte of the unusable character. InBuf parameter points to the first byte of the unusable character sequence.

The values of OutBuf and OutBytesLeft are updated according to the number of bytes available in the output buffer that do not contain converted data.
E2BIG  Indicates an output buffer overflow. If the OutBuf buffer is too small to contain all the converted characters, the character that causes the overflow is not converted. The InBytesLeft parameter indicates the bytes left to be converted (including the character that caused the overflow). The InBuf parameter points to the first byte of the characters left to convert.

EINVAL Indicates the input buffer was truncated. If the original value of InBytesLeft is exhausted in the middle of a character conversion or shift/lock block, the InBytesLeft parameter indicates the number of bytes undefined in the character being converted.

If an input character of shift sequence is truncated by the InBuf buffer, no conversion is attempted on the truncated data, and the InBytesLeft parameter indicates the bytes left to be converted. The InBuf parameter points to the first bytes if the truncated sequence. The OutBuf and OutBytesLeft values are updated according to the number of characters that were previously converted. Because some encoding may have ambiguous data, the EINVAL return value has a special meaning at the end of stream conversion. As such, if a user detects an EOF character on a stream that is being converted and the last return code from the iconv subroutine was EINVAL, the iconv subroutine should be called again, with the same InBytesLeft parameter and the same character string pointed to by the InBuf parameter as when the EINVAL return occurred. As a result, the converter will either convert the string as is or declare it an unusable sequence (EILSEQ).

Files

/usr/lib/nls/loc/iconv/*  Contains code set converter methods.

Related Information

The iconv command, genxlt command.

The iconv_close subroutine, iconv_open subroutine.

iconv_close Subroutine

Purpose

Closes a specified code set converter.

Library

iconv Library (libiconv.a)

Syntax

#include <iconv.h>

int iconv_close (CD)
iconv_t CD;

Description

The iconv_close subroutine closes a specified code set converter and deallocates any resources used by the converter.

Parameters

CD  Specifies the conversion descriptor to be closed.
Return Values
When successful, the `iconv_close` subroutine returns a value of 0. Otherwise, it returns a value of -1 and sets the `errno` global variable to indicate the error.

Error Codes
The following error code is defined for the `iconv_close` subroutine:

EBADF  The conversion descriptor is not valid.

Related Information
The `iconv` subroutine, `iconv_open` subroutine.

The `genxlt` command, `iconv` command.

National Language Support Overview and Converters Overview for Programming in AIX 5L Version 5.3

iconv_open Subroutine

Purpose
Opens a character code set converter.

Library
iconv Library (libiconv.a)

Syntax

```c
#include <iconv.h>

iconv_t iconv_open (const char *ToCode, *FromCode);
```

Description
The `iconv_open` subroutine initializes a code set converter. The code set converter is used by the `iconv` subroutine to convert characters from one code set to another. The `iconv_open` subroutine finds the converter that performs the character code set conversion specified by the `FromCode` and `ToCode` parameters, initializes that converter, and returns a conversion descriptor of type `iconv_t` to identify the code set converter.

The `iconv_open` subroutine first searches the `LOCPATH` environment variable for a converter, using the two user-provided code set names, based on the file name convention that follows:

- `FromCode`: "IBM-850"
- `ToCode`: "ISO8859-1"
- conversion file: "IBM-850_ISO8859-1"

The conversion file name is formed by concatenating the `ToCode` code set name onto the `FromCode` code set name, with an _ (underscore) between them.

The `LOCPATH` environment variable contains a list of colon-separated directory names. The system default for the `LOCPATH` environment variable is:

```
LOCPATH=/usr/lib/nls/loc
```
See Locales in AIX 5L Version 5.3 National Language Support Guide and Reference for more information on the `LOCPATH` environment variable.

The `iconv_open` subroutine first attempts to find the specified converter in an `iconv` subdirectory under any of the directories specified by the `LOCPATH` environment variable, for example, `/usr/lib/nls/loc/iconv`. If the `iconv_open` subroutine cannot find a converter in any of these directories, it looks for a conversion table in an `iconvTable` subdirectory under any of the directories specified by the `LOCPATH` environment variable, for example, `/usr/lib/nls/loc/iconvTable`.

If the `iconv_open` subroutine cannot find the specified converter in either of these locations, it returns `iconv_t` -1 to the calling process and sets the `errno` global variable.

The `iconvTable` directories are expected to contain conversion tables that are the output of the `genxlt` command. The conversion tables are limited to single-byte stateless code sets. See the "List of PC, ISO, and EBCDIC Code Set Converters" in AIX 5L Version 5.3 National Language Support Guide and Reference for more information.

If the named converter is found, the `iconv_open` subroutine will perform the `load` subroutine operation and initialize the converter. A converter descriptor (`iconv_t`) is returned.

**Note:** When a process calls the `exec` subroutine or a `fork` subroutine, all of the opened converters are discarded.

The `iconv_open` subroutine links the converter function using the `load` subroutine, which is similar to the `exec` subroutine and effectively performs a run-time linking of the converter program. Since the `iconv_open` subroutine is called as a library function, it must ensure that security is preserved for certain programs. Thus, when the `iconv_open` subroutine is called from a set root ID program (a program with permission `-s-s-x`), it will ignore the `LOCPATH` environment variable and search for converters only in the `/usr/lib/nls/loc/iconv` directory.

### Parameters

- **ToCode**: Specifies the destination code set.
- **FromCode**: Specifies the originating code set.

### Return Values

A conversion descriptor (`iconv_t`) is returned if successful. Otherwise, the subroutine returns -1, and the `errno` global variable is set to indicate the error.

### Error Codes

- **EINVAL**: The conversion specified by the `FromCode` and `ToCode` parameters is not supported by the implementation.
- **EMFILE**: The number of file descriptors specified by the `OPEN_MAX` configuration variable is currently open in the calling process.
- **ENOMEM**: Too many files are currently open in the system.

### Files

- `/usr/lib/nls/loc/iconv`: Contains loadable method converters.
- `/usr/lib/nls/loc/iconvTable`: Contains conversion tables for single-byte stateless code sets.
Related Information
The “iconv Subroutine” on page 524, “iconv_close Subroutine” on page 526.

The `genxlt` command, `iconv` command.


ilogbf, ilogbl, or ilogb Subroutine

Purpose
Returns an unbiased exponent.

Syntax
```
#include <math.h>

int ilogbf (x)
float x;

int ilogbl (x)
long double x;

int ilogb (x)
double x;
```

Description
The `ilogbf`, `ilogbl`, and `ilogb` subroutines return the exponent part of the x parameter. The return value is the integral part of log_r |x| as a signed integral value, for nonzero x, where r is the radix of the machine’s floating-point arithmetic (r=2).

An application wishing to check for error situations should set the `errno` global variable to zero and call `feclearexcept(FE_ALL_EXCEPT)` before calling these subroutines. Upon return, if `errno` is nonzero or `fetestexcept(FE_INVALID | FE_DIVBYZERO | FE_OVERFLOW | FE_UNDERFLOW)` is nonzero, an error has occurred.

Parameters
x
Specifies the value to be computed.

Return Values
Upon successful completion, the `ilogbf`, `ilogbl`, and `ilogb` subroutines return the exponent part of x as a signed integer value. They are equivalent to calling the corresponding `logb` function and casting the returned value to type `int`.

If x is 0, a domain error occurs, and the value `FP_ILOGB0` is returned.

If x is ±Inf, a domain error occurs, and the value `{INT_MAX}` is returned.

If x is a NaN, a domain error occurs, and the value `FP_ILOGBNAN` is returned.

If the correct value is greater than `{INT_MAX}`, `{INT_MAX}` is returned and a domain error occurs.
If the correct value is less than {INT_MIN}, {INT_MIN} is returned and a domain error occurs.

Related Information
“fclearexcept Subroutine” on page 262 and “fetestexcept Subroutine” on page 270.

The math.h file in AIX 5L Version 5.3 Files Reference.

imaxabs Subroutine

Purpose
Returns absolute value.

Syntax

```c
#include <inttypes.h>

intmax_t imaxabs (j);
intmax_t j;
```

Description
The imaxabs subroutine computes the absolute value of an integer j. If the result cannot be represented, the behavior is undefined.

Parameters

j
Specifies the value to be computed.

Return Values
The imaxabs subroutine returns the absolute value.

Related Information
The “imaxdiv Subroutine.”

The inttypes.h file in AIX 5L Version 5.3 Files Reference.

imaxdiv Subroutine

Purpose
Returns quotient and remainder.

Syntax

```c
#include <inttypes.h>

imaxdiv_t imaxdiv (numer, denom);
intmax_t numer;
intmax_t denom;
```

Description
The imaxdiv subroutine computes numer / denom and numer % denom in a single operation.
Parameters

numerator
Specifies the numerator value to be computed.
denominator
Specifies the denominator value to be computed.

Return Values
The \texttt{imaxdiv} subroutine returns a structure of type \texttt{imaxdiv_t}, comprising both the quotient and the remainder. The structure contains (in either order) the members \texttt{quot} (the quotient) and \texttt{rem} (the remainder), each of which has type \texttt{intmax_t}.

If either part of the result cannot be represented, the behavior is undefined.

Related Information
The \texttt{imaxabs Subroutine} on page 530.
\texttt{inttypes.h File} in AIX 5L Version 5.3 Files Reference.

IMAIIXMapping Subroutine

Purpose
Translates a pair of \texttt{Key} and \texttt{State} parameters to a string and returns a pointer to this string.

Library
Input Method Library (libIM.a)

Syntax
\begin{verbatim}
caddr_t IMAIXMapping(IMMap, Key, State, NBytes)
IMMap IMMap;
KeySym Key;
uint State;
int * NBytes;
\end{verbatim}

Description
The \texttt{IMAIIXMapping} subroutine translates a pair of \texttt{Key} and \texttt{State} parameters to a string and returns a pointer to this string.

This function handles the diacritic character sequence and Alt-NumPad key sequence.

Parameters
\begin{itemize}
\item \texttt{IMMap} \hspace{1cm} Identifies the keymap.
\item \texttt{Key} \hspace{1cm} Specifies the key symbol to which the string is mapped.
\item \texttt{State} \hspace{1cm} Specifies the state to which the string is mapped.
\item \texttt{NBytes} \hspace{1cm} Returns the length of the returning string.
\end{itemize}

Return Values
If the length set by the \texttt{NBytes} parameter has a positive value, the \texttt{IMAIIXMapping} subroutine returns a pointer to the returning string.

\textbf{Note:} The returning string is not null-terminated.
IMAuxCreate Callback Subroutine

Purpose
Tells the application program to create an auxiliary area.

Syntax

```c
int IMAuxCreate( IM, AuxiliaryID, UData)
IMObject IM;
caddr_t *AuxiliaryID;
caddr_t UData;
```

Description
The `IMAuxCreate` subroutine is invoked by the input method of the operating system to create an auxiliary area. The auxiliary area can contain several different forms of data and is not restricted by the interface.

Most input methods display one auxiliary area at a time, but callbacks must be capable of handling multiple auxiliary areas.

This subroutine is provided by applications that use input methods.

Parameters

- `IM`: Indicates the input method instance.
- `AuxiliaryID`: Identifies the newly created auxiliary area.
- `UData`: Identifies an argument passed by the `IMCreate` subroutine.

Return Values
On successful return of the `IMAuxCreate` subroutine, a newly created auxiliary area is set to the `AuxiliaryID` value and the `IMError` global variable is returned. Otherwise, the `IMNoError` value is returned.

Related Information
The `IMCreate` subroutine.

IMAuxDestroy Callback Subroutine

Purpose
Tells the application to destroy the auxiliary area.

Syntax

```c
int IMAuxDestroy( IM, AuxiliaryID, UData)
IMObject IM;
caddr_t AuxiliaryID;
caddr_t UData;
```

Description
The `IMAuxDestroy` subroutine is called by the input method of the operating system to tell the application to destroy an auxiliary area.
This subroutine is provided by applications that use input methods.

**Parameters**

- **IM**
  Indicates the input method instance.

- **AuxiliaryID**
  Identifies the auxiliary area to be destroyed.

- **UData**
  An argument passed by the **IMCreate** subroutine.

**Return Values**

If an error occurs, the **IMAuxDestroy** subroutine returns the **IMError** global variable. Otherwise, the **IMNoError** value is returned.

**Related Information**

The **IMCreate** subroutine.

**IMAuxDraw Callback Subroutine**

**Purpose**

Tells the application program to draw the auxiliary area.

**Syntax**

```c
int IMAuxDraw(IM, AuxiliaryID, AuxiliaryInformation, UData)
```

**Description**

The **IMAuxDraw** subroutine is invoked by the input method to draw an auxiliary area. The auxiliary area should have been previously created.

This subroutine is provided by applications that use input methods.

**Parameters**

- **IM**
  Indicates the input method instance.

- **AuxiliaryID**
  Identifies the auxiliary area.

- **AuxiliaryInformation**
  Points to the **IMAuxInfo** structure.

- **UData**
  An argument passed by the **IMCreate** subroutine.

**Return Values**

If an error occurs, the **IMAuxDraw** subroutine returns the **IMError** global variable. Otherwise, the **IMNoError** value is returned.

**Related Information**

The **IMAuxCreate** subroutine, **IMCreate** subroutine.
IMAuxHide Callback Subroutine

**Purpose**
Tells the application program to hide an auxiliary area.

**Syntax**
```c
int IMAuxHide( IM, AuxiliaryID, UData)
IMObject IM;
caddr_t AuxiliaryID;
caddr_t UData;
```

**Description**
The **IMAuxHide** subroutine is called by the input method to hide an auxiliary area.

This subroutine is provided by applications that use input methods.

**Parameters**
- **IM**
  Indicates the input method instance.
- **AuxiliaryID**
  Identifies the auxiliary area to be hidden.
- **UData**
  An argument passed by the **IMCreate** subroutine.

**Return Values**
If an error occurs, the **IMAuxHide** subroutine returns the **IMError** global variable. Otherwise, the **IMNoError** value is returned.

**Related Information**
The **IMAuxCreate** subroutine, **IMCreate** subroutine.

IMBeep Callback Subroutine

**Purpose**
Tells the application program to emit a beep sound.

**Syntax**
```c
int IMBeep( IM, Percent, UData)
IMObject IM;
int Percent;
caddr_t UData;
```
Description
The **IMBeep** subroutine tells the application program to emit a beep sound.

This subroutine is provided by applications that use input methods.

Parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>IM</td>
<td>Indicates the input method instance.</td>
</tr>
<tr>
<td>Percent</td>
<td>Specifies the beep level. The value range is from -100 to 100, inclusively. A -100 value means no beep.</td>
</tr>
<tr>
<td>UData</td>
<td>An argument passed by the <strong>IMCreate</strong> subroutine.</td>
</tr>
</tbody>
</table>

Return Values
If an error occurs, the **IMBeep** subroutine returns the **IMError** global variable. Otherwise, the **IMNoError** value is returned.

Related Information
The **IMCreate** subroutine.

---

**IMClose Subroutine**

Purpose
Closes the input method.

Library
Input Method Library (libIM.a)

Syntax
```c
void IMClose(IMfep)
IMFep IMfep;
```

Description
The **IMClose** subroutine closes the input method. Before the **IMClose** subroutine is called, all previously created input method instances must be destroyed with the **IMDestroy** subroutine, or memory will not be cleared.

Parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>IMfep</td>
<td>Specifies the input method.</td>
</tr>
</tbody>
</table>

Related Information
The **IMDestroy** subroutine.

---

IMCreate Subroutine

Purpose
Creates one instance of an IMObject object for a particular input method.

Library
Input Method Library (libIM.a)

Syntax
IMObject IMCreate( IMFep, IMCallback, UData)
IMFep IMFep;
IMCallback *IMCallback;
caddr_t UData;

Description
The IMCreate subroutine creates one instance of a particular input method. Several input method instances can be created under one input method.

Parameters
IMFep Specifies the input method.
IMCallback Specifies a pointer to the caller-supplied IMCallback structure.
UData Optionally specifies an application's own information to the callback functions. With this information, the application can avoid external references from the callback functions. The input method does not change this parameter, but merely passes it to the callback functions. The UData parameter is usually a pointer to the application data structure, which contains the information about location, font ID, and so forth.

Return Values
The IMCreate subroutine returns a pointer to the created input method instance of type IMObject. If the subroutine is unsuccessful, a null value is returned and the imerrno global variable is set to indicate the error.

Related Information
The IMDestroy subroutine, IMFilter subroutine, IMLookupString subroutine, IMProcess subroutine.

IMDestroy Subroutine

Purpose
Destroys an input method instance.

Library
Input Method Library (libIM.a)
Syntax

```c
void IMDestroy(IM)
IMObject IM;
```

Description

The `IMDestroy` subroutine destroys an input method instance.

Parameters

- `IM`: Specifies the input method instance to be destroyed.

Related Information


---

**IMFilter Subroutine**

**Purpose**

Determines if a keyboard event is used by the input method for internal processing.

**Library**

Input Method Library (`libIM.a`)

**Syntax**

```c
int IMFilter(Im, Key, State, String, Length)
IMObject Im;
Keysym Key;
uint State, *Length;
caddr_t *String;
```

**Description**

The `IMFilter` subroutine is used to process a keyboard event and determine if the input method for this operating system uses this event. The return value indicates:

- The event is filtered (used by the input method) if the return value is `IMInputUsed`. Otherwise, the input method did not accept the event.
- Independent of the return value, a string may be generated by the keyboard event if pre-editing is complete.

**Note:** The buffer returned from the `IMFilter` subroutine is owned by the input method editor and can not continue between calls.

**Parameters**

- `Im`: Specifies the input method instance.
- `Key`: Specifies the keysym for the event.
- `State`: Defines the state of the keysym. A value of 0 means that the keysym is not redefined.
- `String`: Holds the returned string if one exists. A null value means that no composed string is ready.
Length

Defines the length of the input string. If the string is not null, returns the length.

Return Values

| IMInputUsed                        | The input method for this operating system filtered the event. |
| IMInputNotUsed                    | The input method for this operating system did not use the event. |

Related Information


**IMFreeKeymap Subroutine**

**Purpose**

Frees resources allocated by the `IMInitializeKeymap` subroutine.

**Library**

Input Method Library (`libIM.a`)

**Syntax**

```c
void IMFreeKeymap(IMMap IMMap);
```

**Description**

The `IMFreeKeymap` subroutine frees resources allocated by the `IMInitializeKeymap` subroutine.

**Parameters**

- `IMMap` identifies the keymap.

**Related Information**

The `IMInitializeKeymap` subroutine.


**IMIndicatorDraw Callback Subroutine**

**Purpose**

Tells the application program to draw the indicator.

**Syntax**

```c
int IMIndicatorDraw(IM IM, IMIndicatorInfo *IndicatorInformation, caddr_t UData);
```
Description
The **IMIndicatorDraw** callback subroutine is called by the input method when the value of the indicator is changed. The application program then draws the indicator.

This subroutine is provided by applications that use input methods.

Parameters

<table>
<thead>
<tr>
<th><strong>IM</strong></th>
<th>Indicates the input method instance.</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>IndicatorInformation</strong></td>
<td>Points to the <strong>IMIndicatorInfo</strong> structure that holds the current value of the indicator. The interpretation of this value varies among phonic languages. However, the input method provides a function to interpret this value.</td>
</tr>
<tr>
<td><strong>UData</strong></td>
<td>An argument passed by the <strong>IMCreate</strong> subroutine.</td>
</tr>
</tbody>
</table>

Return Values
If an error happens, the **IMIndicatorDraw** subroutine returns the **IMError** global variable. Otherwise, the **IMNoError** value is returned.

Related Information
The **IMCreate** ("IMCreate Subroutine" on page 536) subroutine, **IMIndicatorHide** ("IMIndicatorHide Callback Subroutine") subroutine.


**IMIndicatorHide** Callback Subroutine

Purpose
Tells the application program to hide the indicator.

Syntax

```c
int IMIndicatorHide( IM, UData )
IMObject IM;
caddr_t UData;
```

Description
The **IMIndicatorHide** subroutine is called by the input method to tell the application program to hide the indicator.

This subroutine is provided by applications that use input methods.

Parameters

<table>
<thead>
<tr>
<th><strong>IM</strong></th>
<th>Indicates the input method instance.</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>UData</strong></td>
<td>Specifies an argument passed by the <strong>IMCreate</strong> subroutine.</td>
</tr>
</tbody>
</table>

Return Values
If an error occurs, the **IMIndicatorHide** subroutine returns the **IMError** global variable. Otherwise, the **IMNoError** value is returned.
Related Information

The **IMCreate** [“IMCreate Subroutine” on page 536](#) subroutine, **IMIndicatorDraw** [“IMIndicatorDraw Callback Subroutine” on page 538](#) subroutine.


**IMInitialize Subroutine**

**Purpose**

Initializes the input method for a particular language.

**Library**

Input Method Library (libIM.a)

**Syntax**

```
IMFep IMInitialize(Name)
char *Name;
```

**Description**

The **IMInitialize** subroutine initializes an input method. The **IMCreate**, **IMFilter**, and **IMLookupString** subroutines use the input method to perform input processing of keyboard events in the form of keysym state modifiers. The **IMInitialize** subroutine finds the input method that performs the input processing specified by the **Name** parameter and returns an Input Method Front End Processor (**IMFep**) descriptor.

Before calling any of the key event-handling functions, the application must create an instance of an **IMObject** object using the **IMFep** descriptor. Each input method can produce one or more instances of **IMObject** object with the **IMCreate** subroutine.

When the **IMInitialize** subroutine is called, strings returned from the input method are encoded in the code set of the locale. Each **IMFep** description inherits the code set of the locale when the input method is initialized. The locale setting does not change the code set of the **IMFep** description after it is created.

The **IMInitialize** subroutine calls the **load** subroutine to load a file whose name is in the form **Name.im**. The **Name** parameter is passed to the **IMInitialize** subroutine. The loadable input method file is accessed in the directories specified by the **LOCPATH** environment variable. The default location for loadable input-method files is the `/usr/lib/nls/loc` directory. If none of the **LOCPATH** directories contain the input method specified by the **Name** parameter, the default location is searched.

**Note:** All **setuid** and **setgid** programs will ignore the **LOCPATH** environment variable.

The name of the input method file usually corresponds to the locale name, which is in the form **Language_territory.codesest@modifier**. In the environment, the modifier is in the form **@im=modifier**. The **IMInitialize** subroutine converts the **@im=** substring to **@** when searching for loadable input-method files.

**Parameters**

**Name** Specifies the language to be used. Each input method is dynamically linked to the application program.
Return Values
If `IMInitialize` succeeds, it returns an `IMFep` handle. Otherwise, null is returned and the `imerrno` global variable is set to indicate the error.

Files

```
/usr/lib/nls/loc
```
Contains loadable input-method files.

Related Information

The `IMCreate` subroutine.


IMInitializeKeymap Subroutine

Purpose
Initializes the keymap associated with a specified language.

Library
Input Method Library (libIM.a)

Syntax

```
IMMap IMInitializeKeymap(char *Name);
```

Description

The `IMInitializeKeymap` subroutine initializes an input method keymap (imkeymap). The `IMAIXMapping` and `IMSimpleMapping` subroutines use the imkeymap to perform mapping of keysym state modifiers to strings. The `IMInitializeKeymap` subroutine finds the imkeymap that performs the keysym mapping and returns an imkeymap descriptor, `IMMap`. The strings returned by the imkeymap mapping functions are treated as unsigned bytes.

The applications that use input methods usually do not need to manage imkeymaps separately. The imkeymaps are managed internally by input methods.

The `IMInitializeKeymap` subroutine searches for an imkeymap file whose name is in the form `Name.im`. The `Name` parameter is passed to the `IMInitializeKeymap` subroutine. The imkeymap file is accessed in the directories specified by the `LOCPATH` environment variable. The default location for input method files is the `/usr/lib/nls/loc` directory. If none of the `LOCPATH` directories contain the keymap method specified by the `Name` parameter, the default location is searched.

Note: All setuid and setgid programs will ignore the `LOCPATH` environment variable.

The name of the imkeymap file usually corresponds to the locale name, which is in the form `Language_territory.codesest@modifier`. In the AIXwindows environment, the modifier is in the form `@im=modifier`. The `IMInitializeKeymap` subroutine converts the `@im= substring to `@` (at sign) when searching for loadable input method files.
Parameters

Name Specifies the name of the imkeymap.

Return Values
The IMInitializeKeymap subroutine returns a descriptor of type IMMap. Returning a null value indicates the occurrence of an error. The IMMap descriptor is defined in the im.h file as the caddr_t structure. This descriptor is used for keymap manipulation functions.

Files
/usr/lib/nls/loc Contains loadable input-method files.

Related Information
The IMFreeKeymap (“IMFreeKeymap Subroutine” on page 538), IMQueryLanguage (“IMQueryLanguage Subroutine” on page 547) subroutine.


IMIOctl Subroutine

Purpose
Performs a variety of control or query operations on the input method.

Library
Input Method Library (libIM.a)

Syntax

```c
int IMIOctl(IMObject IM, int Operation, char *Argument);
```  

Description
The IMIOctl subroutine performs a variety of control or query operations on the input method specified by the IM parameter. In addition, this subroutine can be used to control the unique function of each language input method because it provides input method-specific extensions. Each input method defines its own function.

Parameters

IM Specifies the input method instance.

Operation Specifies the operation.

Argument The use of this parameter depends on which of the following operations is performed.
IM_Refresh
Refreshes the text area, auxiliary areas, and indicator by calling the needed callback functions if these areas are not empty. The Argument parameter is not used.

IM_GetString
Gets the current pre-editing string. The Argument parameter specifies the address of the IMSTR structure supplied by the caller. The callback function is invoked to clear the pre-editing if it exists.

IM_Clear
Clears the text and auxiliary areas if they exist. If the Argument parameter is not a null value, this operation invokes the callback functions to clear the screen. The keyboard state remains the same.

IM_Reset
Clears the auxiliary area if it currently exists. If the Argument parameter is a null value, this operation clears only the internal buffer of the input method. Otherwise, the IMAuxHide subroutine is called, and the input method returns to its initial state.

IM_ChangeLength
Changes the maximum length of the pre-editing string.

IM_ChangeMode
Sets the Processing Mode of the input method to the mode specified by the Argument parameter. The valid value for Argument is:

IMNormalMode
Specifies the normal mode of pre-editing.

IMSuppressedMode
Suppresses pre-editing.

IM_QueryState
Returns the status of the text area, the auxiliary area, and the indicator. It also returns the beep status and the processing mode. The results are stored into the caller-supplied IMQueryState structure pointed to by the Argument parameter.

IM_QueryText
Returns detailed information about the text area. The results are stored in the caller-supplied IMQueryText structure pointed to by the Argument parameter.

IM_QueryAuxiliary
Returns detailed information about the auxiliary area. The results are stored in the caller-supplied IMQueryAuxiliary structure pointed to by the Argument parameter.

IM_QueryIndicator
Returns detailed information about the indicator. The results are stored in the caller-supplied IMQueryIndicator structure pointed to by the Argument parameter.

IM_QueryIndicatorString
Returns an indicator string corresponding to the current indicator. Results are stored in the caller-supplied IMQueryIndicatorString structure pointed to by the Argument parameter. The caller can request either a short or long form with the format member of the IMQueryIndicatorString structure.

IM_SupportSelection
Informs the input method whether or not an application supports an auxiliary area selection list. The application must support selections inside the auxiliary area and determine how selections are displayed. If this operation is not performed, the input method assumes the application does not support an auxiliary area selection list.
Return Values
The IMioctl subroutine returns a value to the IMError global variable that indicates the type of error encountered. Some error types are provided in the /usr/include/imerrno.h file.

Related Information

IMLookupString Subroutine

Purpose
Maps a Key/State (key symbol/state) pair to a string.

Library
Input Method Library (libIM.a)

Syntax
int IMLookupString(Im, Key, State, String, Length)
IMObject Im;
KeySym Key;
uint State, *Length;
caddr_t *String;

Description
The IMLookupString subroutine is used to map a Key/State pair to a localized string. It uses an internal input method keymap (imkeymap) file to map a keysym/modifier to a string. The string returned is encoded in the same code set as the locale of IMObject and IM Front End Processor.

Note: The buffer returned from the IMLookupString subroutine is owned by the input method editor and can not continue between calls.

Parameters
Im Specifies the input method instance.
Key Specifies the key symbol for the event.
State Defines the state for the event. A value of 0 means that the key is not redefined.
String Holds the returned string, if one exists. A null value means that no composed string is ready.
Length Defines the length string on input. If the string is not null, identifies the length returned.

Return Values
IMError Error encountered.
IMReturnNothing No string or keysym was returned.
IMReturnString String returned.
IMProcess Subroutine

Purpose
Processes keyboard events and language-specific input.

Library
Input Method Library (libIM.a)

Note: This subroutine will be removed in future releases. Use the IMFilter ("IMFilter Subroutine" on page 537) and IMLookupString ("IMLookupString Subroutine" on page 544) subroutines to process keyboard events.

Syntax
```c
int IMProcess (IM, KeySymbol, State, String, Length)
IMObject IM;
KeySym KeySymbol;
uint State;
caddr_t * String;
uint * Length;
```

Description
This subroutine is a main entry point to the input method of the operating system. The IMProcess subroutine processes one keyboard event at a time. Processing proceeds as follows:
- Validates the IM parameter.
- Performs keyboard translation for all supported modifier states.
- Invokes internal function to do language-dependent processing.
- Performs any necessary callback functions depending on the internal state.
- Returns to application, setting the String and Length parameters appropriately.

Parameters
- **IM** Specifies the input method instance.
- **KeySymbol** Defines the set of keyboard symbols that will be handled.
- **State** Specifies the state of the keyboard.
- **String** Holds the returned string. Returning a null value means that the input is used or discarded by the input method.
  - **Note:** The String parameter is not a null-terminated string.
- **Length** Stores the length, in bytes, of the String parameter.

Return Values
This subroutine returns the IMError global variable if an error occurs. The IMErrno global variable is set to indicate the error. Some of the variable values include:
- **IMError** Error occurred during this subroutine.
- **IMTextAndAuxiliaryOff** No text string in the Text area, and the Auxiliary area is not shown.
- **IMTextOn** Text string in the Text area, but no Auxiliary area.
Related Information
The IMClose ("IMClose Subroutine" on page 535) subroutine, IMCreate ("IMCreate Subroutine" on page 536) subroutine, IMFilter ("IMFilter Subroutine" on page 537) subroutine, IMLookupString ("IMLookupString Subroutine" on page 544) subroutine.


IMProcessAuxiliary Subroutine

Purpose
Notifies the input method of input for an auxiliary area.

Library
Input Method Library (libIM.a)

Syntax
```c
int IMProcessAuxiliary(IM, AuxiliaryID, Button, PanelRow, PanelColumn, ItemRow, ItemColumn, String, Length)
```

Description
The IMProcessAuxiliary subroutine notifies the input method instance of input for an auxiliary area.

Parameters
- **IM** Specifies the input method instance.
- **AuxiliaryID** Identifies the auxiliary area.
**Button**

Specifies one of the following types of input:

- **IM_ABORT**
  Abort button is pushed.

- **IM_CANCEL**
  Cancel button is pushed.

- **IM_ENTER**
  Enter button is pushed.

- **IM_HELP**
  Help button is pushed.

- **IM_IGNORE**
  Ignore button is pushed.

- **IM_NO**
  No button is pushed.

- **IM_OK**
  OK button is pushed.

- **IM_RETRY**
  Retry button is pushed.

- **IM_SELECTED**
  Selection has been made. Only in this case do the PanelRow, PanelColumn, ItemRow, and ItemColumn parameters have meaningful values.

- **IM_YES**
  Yes button is pushed.

**PanelRow**
Indicates the panel on which the selection event occurred.

**PanelColumn**
Indicates the panel on which the selection event occurred.

**ItemRow**
Indicates the selected item.

**ItemColumn**
Indicates the selected item.

**String**
Holds the returned string. If a null value is returned, the input is used or discarded by the input method. Note that the String parameter is not a null-terminated string.

**Length**
Stores the length, in bytes, of the String parameter.

---

**Related Information**

The [IMAuxCreate](#) subroutine.


---

**IMQueryLanguage Subroutine**

**Purpose**
Checks to see if the specified input method is supported.

**Library**
Input Method Library (libIM.a)

**Syntax**

```c
uint IMQueryLanguage( Name )
IMLanguage Name;
```
The **IMQueryLanguage** subroutine checks to see if the input method specified by the Name parameter is supported.

### Parameters

**Name**
Specifies the input method.

### Return Values

The **IMQueryLanguage** subroutine returns a true value if the specified input method is supported, a false value if not.

### Related Information

The **IMClose** subroutine, **IMInitialize** subroutine, **Input Methods**, **National Language Support Overview**, **Understanding Keyboard Mapping** contains a list of supported languages in *AIX 5L Version 5.3 National Language Support Guide and Reference*.

---

**IMSimpleMapping Subroutine**

### Purpose

Translates a pair of **KeySymbol** and **State** parameters to a string and returns a pointer to this string.

### Library

Input Method Library (*libIM.a*)

### Syntax

```c
void IMSimpleMapping (IMMap IMMap, KeySymbol KeySym, State, NBytes)
```

### Description

Like the **IMAIXMapping** subroutine, the **IMSimpleMapping** subroutine translates a pair of **KeySymbol** and **State** parameters to a string and returns a pointer to this string. The parameters have the same meaning as those in the **IMAIXMapping** subroutine.

The **IMSimpleMapping** subroutine differs from the **IMAIXMapping** subroutine in that it does not support the diacritic character sequence or the Alt-NumPad key sequence.

### Parameters

**IMMap**
Identifies the keymap.

**KeySymbol**
Key symbol to which the string is mapped.

**State**
Specifies the state to which the string is mapped.

**NBytes**
Returns the length of the returning string.
Related Information
The IMAIXMapping subroutine, IMFreeKeymap subroutine, IMInitializeKeymap subroutine, IMTextCursor Callback Subroutine

IMTextCursor Callback Subroutine

Purpose
Asks the application to move the text cursor.

Syntax
int IMTextCursor(IM, Direction, Cursor, UData)
IMObject IM;
uint Direction;
int *Cursor;
caddr_t UData;

Description
The IMTextCursor subroutine is called by the Input Method when the Cursor Up or Cursor Down key is input to the IMFilter and IMLookupString subroutines.

This subroutine sets the new display cursor position in the text area to the integer pointed to by the Cursor parameter. The cursor position is relative to the top of the text area. A value of -1 indicates the cursor should not be moved.

Because the input method does not know the actual length of the screen it always treats a text string as one-dimensional (a single line). However, in the terminal emulator, the text string sometimes wraps to the next line. The IMTextCursor subroutine performs this conversion from single-line to multiline text strings. When you move the cursor up or down, the subroutine interprets the cursor position on the text string relative to the input method.

This subroutine is provided by applications that use input methods.

Parameters

IM Indicates the Input Method instance.
Direction Specifies up or down.
Cursor Specifies the new cursor position or -1.
UData Specifies an argument passed by the IMCreate subroutine.

Return Values
If an error occurs, the IMTextCursor subroutine returns the IMError global variable. Otherwise, the IMNoError value is returned.

Related Information
The IMCreate subroutine, IMFilter subroutine, IMLookupString subroutine, IMTextDraw subroutine.
IMTextDraw Callback Subroutine

Purpose
Tells the application program to draw the text string.

Syntax
```c
int IMTextDraw(IM IM, IMTextInfo *TextInfo, UData UData);
```

Description
The IMTextDraw subroutine is invoked by the Input Method whenever it needs to update the screen with its internal string. This subroutine tells the application program to draw the text string.

This subroutine is provided by applications that use input methods.

Parameters
- **IM**: Indicates the input method instance.
- **TextInfo**: Points to the IMTextInfo structure.
- **UData**: An argument passed by the IMCreate subroutine.

Return Values
If an error occurs, the IMTextDraw subroutine returns the IMError global variable. Otherwise, the IMNoError value is returned.

Related Information
The IMCreate subroutine.

IMTextHide Callback Subroutine

Purpose
Tells the application program to hide the text area.

Syntax
```c
int IMTextHide(IM IM, UData UData);
```

Description
The IMTextHide subroutine is called by the input method when the text area should be cleared. This subroutine tells the application program to hide the text area.
This subroutine is provided by applications that use input methods.

**Parameters**

*IM*  
Indicates the input method instance.

*UData*  
Specifies an argument passed by the **IMCreate** subroutine.

**Return Values**

If an error occurs, the **IMTextHide** subroutine returns an **IMError** value. Otherwise, an **IMNoError** value is returned.

**Related Information**

The **IMTextDraw** subroutine.

---

**IMTextStart Callback Subroutine**

**Purpose**

Notifies the application program of the length of the pre-editing space.

**Syntax**

```c
int IMTextStart(IM, Space, UData)
IMObject IM;
int *Space;
caddr_t UData;
```

**Description**

The **IMTextStart** subroutine is called by the input method when the pre-editing is started, but prior to calling the **IMTextDraw** callback subroutine. This subroutine notifies the input method of the length, in terms of bytes, of pre-editing space. It sets the length of the available space (>=0) on the display to the integer pointed to by the *Space parameter. A value of -1 indicates that the pre-editing space is dynamic and has no limit.

This subroutine is provided by applications that use input methods.

**Parameters**

*IM*  
Indicates the input method instance.

*Space*  
Maximum length of pre-editing string.

*UData*  
An argument passed by the **IMCreate** subroutine.

**Related Information**

The **IMCreate** subroutine, **IMTextDraw** subroutine.  

---

**Base Operating System (BOS) Runtime Services (A-P) 551**

**inet_aton Subroutine**

**Purpose**
Converts an ASCII string into an Internet address.

**Library**
Standard C Library (**libc.a**)

**Syntax**
```
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>

int inet_aton (CharString, InternetAddr)
char * CharString;
struct in_addr * InternetAddr;
```

**Description**
The **inet_aton** subroutine takes an ASCII string representing the Internet address in dot notation and converts it into an Internet address.

All applications containing the **inet_aton** subroutine must be compiled with **_BSD** set to a specific value. Acceptable values are 43 and 44. In addition, all socket applications must include the BSD **libbsd.a** library.

**Parameters**
- **CharString** Contains the ASCII string to be converted to an Internet address.
- **InternetAddr** Contains the Internet address that was converted from the ASCII string.

**Return Values**
Upon successful completion, the **inet_aton** subroutine returns 1 if **CharString** is a valid ASCII representation of an Internet address.

The **inet_aton** subroutine returns 0 if **CharString** is not a valid ASCII representation of an Internet address.

**Files**
- **/etc/hosts** Contains host names.
- **/etc/networks** Contains network names.

**Related Information**
The **endhostent** subroutine, **endnetent** subroutine, **gethostbyaddr** subroutine, **gethostbyname** subroutine, **getnetbyaddr** subroutine, **getnetbyname** subroutine, **getnetent** subroutine, **inet_addr** subroutine, **inet_inaddr** subroutine, **inet_makeaddr** subroutine, **inet_network** subroutine, **inet_ntoa** subroutine, **sethostent** subroutine, **setnetent** subroutine.
initgroups Subroutine

Purpose
Initializes supplementary group ID.

Library
Standard C Library (libc.a)

Syntax
int initgroups (User, BaseGID);
const char *User;
int BaseGID;

Description
Attention: The initgroups subroutine uses the getgrent and getpwent family of subroutines. If the program that invokes the initgroups subroutine uses any of these subroutines, calling the initgroups subroutine overwrites the static storage areas used by these subroutines.

The initgroups subroutine reads the defined group membership of the specified User parameter and sets the supplementary group ID of the current process to that value. The BaseGID parameter is always included in the supplementary group ID. The supplementary group is normally the principal user’s group. If the user is in more than NGROUPS_MAX groups, set in the limits.h file, only NGROUPS_MAX groups are set, including the BaseGID group.

Parameters
User Identifies a user.
BaseGID Specifies an additional group to include in the group set.

Return Values
0 Indicates that the subroutine was success.
-1 Indicates that the subroutine failed. The errno global variable is set to indicate the error.

Related Information
The getgid subroutine, getgrent, getgrgid, getgrnam, putgrent, setgrent, or endgrent subroutine, getgroups subroutine, setgroups subroutine.

Related Information
The groups command, setgroups command.

initialize Subroutine

Purpose
Performs printer initialization.
Library
None (provided by the formatter).

Syntax
```c
#include <piostruct.h>
int initialize ()
```

Description
The `initialize` subroutine is invoked by the formatter driver after the `setup` subroutine returns.

If the `-j` flag passed from the `qprt` command has a nonzero value (true), the `initialize` subroutine uses the `piocmdout` subroutine to send a command string to the printer. This action initializes the printer to the proper state for printing the file. Any variables referenced by the command string should be the attribute values from the database, overridden by values from the command line.

If the `-j` flag passed from the `qprt` command has a nonzero value (true), any necessary fonts should be downloaded.

Return Values

0 Indicates a successful operation.

If the `initialize` subroutine detects an error, it uses the `piomsgout` subroutine to invoke an error message. It then invokes the `pioexit` subroutine with a value of `PIOEXITBAD`.

Note: If either the `piocmdout` or `piogetstr` subroutine detects an error, it issues its own error messages and terminates the print job.

Related Information
The `piocmdout` subroutine, `pioexit` subroutine, `piogetstr` subroutine, `piomsgout` subroutine, `setup` subroutine.

Adding a New Printer Type to Your System | Printer Addition Management Subsystem: Programming Overview | Understanding Embedded References in Printer Attribute Strings in AIX 5L Version 5.3 Kernel Extensions and Device Support Programming Concepts.

Print formatter example in AIX 5L Version 5.3 General Programming Concepts: Writing and Debugging Programs.

---

**insque or remque Subroutine**

**Purpose**
Inserts or removes an element in a queue.

**Library**
Standard C Library (`libc.a`)

**Syntax**
```c
#include <search.h>
```
insque (Element, Pred)
void *Element, *Pred;
remque (Element)
void *Element;

Description
The insque and remque subroutines manipulate queues built from double-linked lists. Each element in the queue must be in the form of a qelem structure. The next and prev elements of that structure must point to the elements in the queue immediately before and after the element to be inserted or deleted.

The insque subroutine inserts the element pointed to by the Element parameter into a queue immediately after the element pointed to by the Pred parameter.

The remque subroutine removes the element defined by the Element parameter from a queue.

Parameters
Pred Points to the element in the queue immediately before the element to be inserted or deleted.
Element Points to the element in the queue immediately after the element to be inserted or deleted.

Related Information
Searching and Sorting Example Program in AIX 5L Version 5.3 General Programming Concepts: Writing and Debugging Programs.
Subroutines Overview in AIX 5L Version 5.3 General Programming Concepts: Writing and Debugging Programs.

install_lwcf_handler Subroutine

Purpose
Registers the signal handler to dump a lightweight core file for signals that normally cause the generation of a core file.

Library
PTools Library (libptools_ptr.a)

Syntax
void install_lwcf_handler (void);

Description
The install_lwcf_handler subroutine registers the signal handler to dump a lightweight core file for signals that normally cause a core file to be generated. The format of lightweight core files complies with the Parallel Tools Consortium Lightweight Core File Format.

The install_lwcf_handler subroutine uses the LIGHTWEIGHT_CORE environment variable to determine the target lightweight core file. If the LIGHTWEIGHT_CORE environment variable is defined, a lightweight core file will be generated. Otherwise, a normal core file will be generated.

If the LIGHTWEIGHT_CORE environment variable is defined without a value, the lightweight core file is assigned the default file name lw_core and is created under the current working directory if it does not already exist.
If the \texttt{LIGHTWEIGHT\_CORE} environment variable is defined with a value of \texttt{STDERR}, the lightweight core file is output to the standard error output device of the process. Keyword \texttt{STDERR} is not case-sensitive.

If the \texttt{LIGHTWEIGHT\_CORE} environment variable is defined with the value of a character string other than \texttt{STDERR}, the string is used as a path name for the lightweight core file generated.

If the target lightweight core file already exists, the traceback information is appended to the file.

The \texttt{install\_lwcf\_handler} subroutine can be called directly from an application to register the signal handler. Alternatively, linker option \texttt{-binitfini:install\_lwcf\_handler} can be used when linking an application, which specifies to execute the \texttt{install\_lwcf\_handler} subroutine when the application is initialized. The advantage of the second method is that the application code does not need to change to invoke the \texttt{install\_lwcf\_handler} subroutine.

\textbf{Related Information}

The \texttt{mt\_trce} and \texttt{sigaction} subroutines.

\textbf{ioc\ldots, ioctlx, ioctl32, or ioctl32x Subroutine}

\textbf{Purpose}
Performs control functions associated with open file descriptors.

\textbf{Library}
Standard C Library (\texttt{libc.a})

BSD Library (\texttt{libbsd.a})

\textbf{Syntax}

```
#include <sys/ioctl.h>
#include <sys/types.h>
#include <unistd.h>
#include <stropts.h>

int ioctl (FileDescriptor, Command, Argument)
int FileDescriptor , Command;
void \* Argument;

int ioctlx (FileDescriptor, Command, Argument, Ext)
int FileDescriptor , Command;
void \* Argument;
int Ext;

int ioctl32 (FileDescriptor, Command, Argument)
int FileDescriptor, Command;
unsigned int Argument;

int ioctl32x (FileDescriptor, Command, Argument, Ext)
int FileDescriptor, Command;
unsigned int Argument;
unsigned int Ext;
```
Description

The **ioctl** subroutine performs a variety of control operations on the object associated with the specified open file descriptor. This function is typically used with character or block special files, sockets or generic device support such as the `termio` general terminal interface.

The control operation provided by this function call is specific to the object being addressed, as are the data type and contents of the `Argument` parameter. The `ioctlx` form of this function can be used to pass an additional extension parameter to objects supporting it. The `ioctl132` and `ioctl132x` forms of this function behave in the same way as `ioctl` and `ioctlx`, but allow 64-bit applications to call the `ioctl` routine for an object that does not normally work with 64-bit applications.

Performing an `ioctl` function on a file descriptor associated with an ordinary file results in an error being returned.

Parameters

- **FileDescriptor**
  Specifies the open file descriptor for which the control operation is to be performed.

- **Command**
  Specifies the control function to be performed. The value of this parameter depends on which object is specified by the `FileDescriptor` parameter.

- **Argument**
  Specifies additional information required by the function requested in the `Command` parameter. The data type of this parameter (a `void` pointer) is object-specific, and is typically used to point to an object device-specific data structure. However, in some device-specific instances, this parameter is used as an integer.

- **Ext**
  Specifies an extension parameter used with the `ioctlx` subroutine. This parameter is passed on to the object associated with the specified open file descriptor. Although normally of type `int`, this parameter can be used as a pointer to a device-specific structure for some devices.

File Input/Output (FIO) ioctl Command Values

A number of file input/output (FIO) ioctl commands are available to enable the `ioctl` subroutine to function similar to the `fcntl` subroutine:

**FIOCLEX** and **FIONCLEX**

Manipulate the `close-on-exec` flag to determine if a file descriptor should be closed as part of the normal processing of the `exec` subroutine. If the flag is set, the file descriptor is closed. If the flag is clear, the file descriptor is left open.

The following code sample illustrates the use of the `fcntl` subroutine to set and clear the `close-on-exec` flag:

```c
/* set the close-on-exec flag for fd1 */
fcntl(fd1,F_SETFD,F_DLOEXEC);

/* clear the close-on-exec flag for fd2 */
fcntl(fd2,F_SETFD,0);
```

Although the `fcntl` subroutine is normally used to set the `close-on-exec` flag, the `ioctl` subroutine may be used if the application program is linked with the Berkeley Compatibility Library (`libbsd.a`) or the Berkeley Thread Safe Library (`libbsd_r.a`) (4.2.1 and later versions). The following ioctl code fragment is equivalent to the preceding `fcntl` fragment:

```c
/* set the close-on-exec flag for fd1 */
ioctl(fd1,FIOCLEX,0);

/* clear the close-on-exec flag for fd2 */
ioctl(fd2,FIONCLEX,0);
```

The third parameter to the `ioctl` subroutine is not used for the `FIOCLEX` and `FIONCLEX` ioctl commands.
FIONBIO Enables nonblocking I/O. The effect is similar to setting the O_NONBLOCK flag with the fcntl subroutine. The third parameter to the ioctl subroutine for this command is a pointer to an integer that indicates whether nonblocking I/O is being enabled or disabled. A value of 0 disables non-blocking I/O. Any nonzero value enables nonblocking I/O. A sample code fragment follows:

```c
int flag;
/* enable NBIO for fd1 */
flag = 1;
ioctl(fd1,FIONBIO,&flag);
/* disable NBIO for fd2 */
flag = 0;
ioctl(fd2,FIONBIO,&flag);
```

FIONREAD Determines the number of bytes that are immediately available to be read on a file descriptor. The third parameter to the ioctl subroutine for this command is a pointer to an integer variable where the byte count is to be returned. The following sample code illustrates the proper use of the FIONREAD ioctl command:

```c
int nbytes;
ioctl(fd,FIONREAD,&nbytes);
```

FIOASYNC Enables a simple form of asynchronous I/O notification. This command causes the kernel to send SIGIO signal to a process or a process group when I/O is possible. Only sockets, ttys, and pseudo-ttys implement this functionality.

The third parameter of the ioctl subroutine for this command is a pointer to an integer variable that indicates whether the asynchronous I/O notification should be enabled or disabled. A value of 0 disables I/O notification; any nonzero value enables I/O notification. A sample code segment follows:

```c
int flag;
/* enable ASYNC on fd1 */
flag = 1;
ioctl(fd, FIOASYNC,&flag);
/* disable ASYNC on fd2 */
flag = 0;
ioctl(fd, FIOASYNC,&flag);
```

FIOSETOWN Sets the recipient of the SIGIO signals when asynchronous I/O notification (FIOASYNC) is enabled. The third parameter to the ioctl subroutine for this command is a pointer to an integer that contains the recipient identifier. If the value of the integer pointed to by the third parameter is negative, the value is assumed to be a process group identifier. If the value is positive, it is assumed to be a process identifier.

Sockets support both process groups and individual process recipients, while ttys and pseudo-ttys support only process groups. Attempts to specify an individual process as the recipient will be converted to the process group to which the process belongs. The following code example illustrates how to set the recipient identifier:

```c
int owner;
owner = -getpgrp();
ioctl(fd, FIOSETOWN,&owner);
```

**Note:** In this example, the asynchronous I/O signals are being enabled on a process group basis. Therefore, the value passed through the owner parameter must be a negative number.

The following code sample illustrates enabling asynchronous I/O signals to an individual process:

```c
int owner;
owner = getpid();
ioctl(fd, FIOSETOWN,&owner);
```
**FIОGETOWN**

Determines the current recipient of the asynchronous I/O signals of an object that has asynchronous I/O notification (FIOASYNC) enabled. The third parameter to the `ioctl` subroutine for this command is a pointer to an integer used to return the owner ID. For example:

```c
int owner;
ioctl(fd, FIОGETOWN, &owner);
```

If the owner of the asynchronous I/O capability is a process group, the value returned in the reference parameter is negative. If the owner is an individual process, the value is positive.

**Return Values**

If the `ioctl` subroutine fails, a value of -1 is returned. The `errno` global variable is set to indicate the error.

The `ioctl` subroutine fails if one or more of the following are true:

- **EBADF**
  The `FileDescriptor` parameter is not a valid open file descriptor.

- **EFAULT**
  The `Argument` or `Ext` parameter is used to point to data outside of the process address space.

- **EINVAL**
  The `Command` or `Argument` parameter is not valid for the specified object.

- **ENOTTY**
  The `FileDescriptor` parameter is not associated with an object that accepts control functions.

- **ENODEV**
  The `FileDescriptor` parameter is associated with a valid character or block special file, but the supporting device driver does not support the `ioctl` function.

- **ENXIO**
  The `FileDescriptor` parameter is associated with a valid character or block special file, but the supporting device driver is not in the configured state.

Object-specific error codes are defined in the documentation for associated objects.

**Related Information**

The `ddioctl` device driver entry point and the `fp_ioctl` kernel service in **AIX 5L Version 5.3 Technical Reference: Kernel and Subsystems**.

The Special Files Overview in **AIX 5L Version 5.3 Files Reference**.

The Input and Output Handling Programmer’s Overview, the tty Subsystem Overview in **AIX 5L Version 5.3 General Programming Concepts: Writing and Debugging Programs**.

The Sockets Overview and Understanding Socket Data Transfer in **AIX 5L Version 5.3 Communications Programming Concepts**.

**isblank Subroutine**

**Purpose**

Tests for a blank character.
#include <ctype.h>

int isblank (c);

### Description

The `isblank` subroutine tests whether the `c` parameter is a character of class `blank` in the program's current locale.

The `c` parameter is a type `int`, the value of which the application shall ensure is a character representable as an `unsigned char` or equal to the value of the macro `EOF`. If the parameter has any other value, the behavior is undefined.

### Parameters

- `c` Specifies the character to be tested.

### Return Values

The `isblank` subroutine returns nonzero if `c` is a `<blank>`; otherwise, it returns 0.

### Related Information

Related to the `ctype, isalpha, isupper, islower, isdigit, isxdigit, isalnum, isspace, ispunct, isprint, isgraph, iscntrl, or isascii Subroutines` on page 208.

These subroutine calls are documented in AIX 5L Version 5.3 Technical Reference: Base Operating System and Extensions Volume 2.

---

### isendwin Subroutine

#### Purpose

Determines whether the `endwin` subroutine was called without any subsequent refresh calls.

#### Library

Curses Library (**libcurses.a**)

#### Syntax

```c
#include <curses.h>
isendwin()
```

#### Description

The `isendwin` subroutine determines whether the `endwin` subroutine was called without any subsequent refresh calls. If the `endwin` was called without any subsequent calls to the `wrefresh` or `doupdate` subroutines, the `isendwin` subroutine returns `TRUE`.

#### Return Values

- **TRUE** Indicates the `endwin` subroutine was called without any subsequent calls to the `wrefresh` or `doupdate` subroutines.
- **FALSE** Indicates subsequent calls to the refresh subroutines.
**Related Information**

The `doupdate` subroutine, `endwin` subroutine, `wrefresh` subroutine.

---

**isfinite Macro**

**Purpose**
Tests for finite value.

**Syntax**

```c
#include <math.h>

int isfinite (x);
```

**Description**

The `isfinite` macro determines whether its argument has a finite value (zero, subnormal, or normal, and not infinite or NaN). An argument represented in a format wider than its semantic type is converted to its semantic type. Determination is based on the type of the argument.

**Parameters**

- `x` Specifies the value to be tested.

**Return Values**

The `isfinite` macro returns a nonzero value if its argument has a finite value.

**Related Information**

- `fpclassify Macro` on page 306,
- `isinf Subroutine` on page 563,
- `class, _class, finite, isnan, or unordered Subroutines` on page 167,
- `isnormal Macro` on page 565.

The `signbit Subroutine` in *AIX 5L Version 5.3 Technical Reference: Base Operating System and Extensions Volume 2*.

**math.h** in *AIX 5L Version 5.3 Files Reference*.

---

**isgreater Macro**

**Purpose**
Tests if `x` is greater than `y`.

**Syntax**

```c
#include <math.h>

int isgreater (x, y);
```
Description
The isgreater macro determines whether its first argument is greater than its second argument. The value of isgreater(x, y) is equal to (x) > (y); however, unlike (x) > (y), isgreater(x, y) does not raise the invalid floating-point exception when x and y are unordered.

Parameters
x Specifies the first value to be compared.
y Specifies the first value to be compared.

Return Values
Upon successful completion, the isgreater macro returns the value of (x) > (y).
If x or y is NaN, 0 is returned.

Related Information
isgreaterequal Subroutine, isless Macro on page 563, islessequal Macro on page 564, islessgreater Macro on page 565, isunordered Macro on page 566.

isgreaterequal Subroutine

Purpose
Tests if x is greater than or equal to y.

Syntax
#include <math.h>

int isgreaterequal (x, y)
real-floating x;
real-floating y;

Description
The isgreaterequal macro determines whether its first argument is greater than or equal to its second argument. The value of isgreaterequal (x, y) is equal to (x) >= (y); however, unlike (x) >= (y), isgreaterequal (x, y) does not raise the invalid floating-point exception when x and y are unordered.

Parameters
x Specifies the first value to be compared.
y Specifies the second value to be compared.

Return Values
Upon successful completion, the isgreaterequal macro returns the value of (x) >= (y).
If x or y is NaN, 0 is returned.
isinf Subroutine

Purpose
Tests for infinity.

Syntax
#include <math.h>

int isinf (x)
real-floating x;

Description
The isinf macro determines whether its argument value is an infinity (positive or negative). An argument represented in a format wider than its semantic type is converted to its semantic type. Determination is based on the type of the argument.

Parameters
x Specifies the value to be checked.

Return Values
The isinf macro returns a nonzero value if its argument has an infinite value.

Related Information


isless Macro

Purpose
Tests if x is less than y.

Syntax
#include <math.h>

int isless (x, y)
real-floating x;
real-floating y;
Description
The isless macro determines whether its first argument is less than its second argument. The value of isless(x, y) is equal to (x) < (y); however, unlike (x) < (y), isless(x, y) does not raise the invalid floating-point exception when x and y are unordered.

Parameters

x Specifies the first value to be compared.
y Specifies the second value to be compared.

Return Values
Upon successful completion, the isless macro returns the value of (x) < (y).

If x or y is NaN, 0 is returned.

Related Information
isgreater Macro” on page 561, “isgreaterequal Subroutine” on page 562, “islessequal Macro,” “islessgreater Macro” on page 565, and “isunordered Macro” on page 566.

islessequal Macro

Purpose
Tests if x is less than or equal to y.

Syntax

#include <math.h>

int islessequal (x, y)
real-floating x;
real-floating y;

Description
The islessequal macro determines whether its first argument is less than or equal to its second argument. The value of islessequal(x, y) is equal to (x) <= (y); however, unlike (x) <= (y), islessequal(x, y) does not raise the invalid floating-point exception when x and y are unordered.

Parameters

x Specifies the first value to be compared.
y Specifies the second value to be compared.

Return Values
Upon successful completion, the islessequal macro returns the value of (x) <= (y).

If x or y is NaN, 0 is returned.
islessgreater Macro

Purpose
Tests if \( x \) is less than or greater than \( y \).

Syntax

```c
#include <math.h>

int islessgreater (x, y);
real-floating x;
real-floating y;
```

Description
The `islessgreater` macro determines whether its first argument is less than or greater than its second argument. The `islessgreater(x, y)` macro is similar to \((x) < (y) \lor (x) > (y)\); however, `islessgreater(x, y)` does not raise the invalid floating-point exception when \( x \) and \( y \) are unordered (nor does it evaluate \( x \) and \( y \) twice).

Parameters

- \( x \) Specifies the first value to be compared.
- \( y \) Specifies the second value to be compared.

Return Values
Upon successful completion, the `islessgreater` macro returns the value of \((x) < (y) \lor (x) > (y)\).

If \( x \) or \( y \) is NaN, 0 is returned.

Related Information
- "isgreater Macro" on page 561, "isgreaterequal Subroutine" on page 562, "islessequal Macro" on page 564, "islessgreater Macro" and "isunordered Macro" on page 566.

math.h in AIX 5L Version 5.3 Files Reference.

isnormal Macro

Purpose
Tests for a normal value.

Syntax

```c
#include <math.h>

int isnormal (x);
real-floating x;
```

Related Information
- "isgreater Macro" on page 561, "isgreaterequal Subroutine" on page 562, "islessequal Macro" on page 563, "islessgreater Macro" on page 564, and "isunordered Macro" on page 566.

math.h in AIX 5L Version 5.3 Files Reference.
Description
The **isnormal** macro determines whether its argument value is normal (neither zero, subnormal, infinite, nor NaN) or not. An argument represented in a format wider than its semantic type is converted to its semantic type. Determination is based on the type of the argument.

Parameters

- **x** Specifies the value to be tested.

Return Values
The **isnormal** macro returns a nonzero value if its argument has a normal value.

Related Information

Related to the **isnormal** macro are:
- "fpclassify Macro" on page 306
- "isfinite Macro" on page 561
- "isnan Subroutine" on page 563
- "class, finite, isnan, or unordered Subroutines" on page 167


The **math.h** in AIX 5L Version 5.3 Files Reference.

isunordered Macro

Purpose
Tests if arguments are unordered.

Syntax

```c
#include <math.h>
int isunordered (x, y)
real-floating x;
real-floating y;
```

Description
The **isunordered** macro determines whether its arguments are unordered.

Parameters

- **x** Specifies the first value in the order.
- **y** Specifies the second value in the order.

Return Values
Upon successful completion, the **isunordered** macro returns 1 if its arguments are unordered, and 0 otherwise.

If **x** or **y** is NaN, 0 is returned.

Related Information
- "isgreater Macro" on page 561
- "isgreaterequal Subroutine" on page 562
- "isless Macro" on page 563
- "islessequal Macro" on page 564
- "islessgreater Macro" on page 565
iswalnum, iswalpha, iswcntrl, iswdigit, iswgraph, iswlower, iswprint, iswpunct, iswspace, iswupper, or iswxdigit Subroutine

Purpose
Tests a wide character for membership in a specific character class.

Library
Standard C Library (libc.a)

Syntax
#include <wchar.h>

int iswalnum (WC)
    wint_t WC;
int iswalpha (WC)
    wint_t WC;
int iswcntrl (WC)
    wint_t WC;
int iswdigit (WC)
    wint_t WC;
int iswgraph (WC)
    wint_t WC;
int iswlower (WC)
    wint_t WC;
int iswprint (WC)
    wint_t WC;
int iswpunct (WC)
    wint_t WC;
int iswspace (WC)
    wint_t WC;
int iswupper (WC)
    wint_t WC;
int iswxdigit (WC)
    wint_t WC;

Description
The isw subroutines check the character class status of the wide character code specified by the WC parameter. Each subroutine tests to see if a wide character is part of a different character class. If the wide character is part of the character class, the isw subroutine returns true; otherwise, it returns false.

Each subroutine is named by adding the isw prefix to the name of the character class that the subroutine tests. For example, the iswalpha subroutine tests whether the wide character specified by the WC parameter is an alphabetic character. The character classes are defined as follows:

<table>
<thead>
<tr>
<th>Character Class</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>alnum</td>
<td>Alphanumeric character.</td>
</tr>
<tr>
<td>alpha</td>
<td>Alphabetic character.</td>
</tr>
<tr>
<td>cntrl</td>
<td>Control character. No characters in the alpha or print classes are included.</td>
</tr>
<tr>
<td>digit</td>
<td>Numeric digit character.</td>
</tr>
<tr>
<td>graph</td>
<td>Graphic character for printing, not including the space character or cntrl characters. Includes all characters in the digit and punct classes.</td>
</tr>
<tr>
<td>lower</td>
<td>Lowercase character. No characters in cntrl, digit, punct, or space are included.</td>
</tr>
<tr>
<td>print</td>
<td>Print character. All characters in the graph class are included, but no characters in cntrl are included.</td>
</tr>
</tbody>
</table>
Parameters

WC Specifies a wide character for testing.

Return Values

If the wide character tested is part of the particular character class, the isw subroutine returns a nonzero value; otherwise it returns a value of 0.

Related Information

The iswctype subroutine, “iswctype or is_wctype Subroutine” on page 569 setlocale subroutine, tolower subroutine, toupper subroutine wctype subroutine.

Subroutines, Example Programs, and Libraries Wide Character Classification Subroutines in AIX 5L Version 5.3 General Programming Concepts: Writing and Debugging Programs.


iswblank Subroutine

Purpose

Tests for a blank wide-character code.

Syntax

#include <wctype.h>

int iswblank (wc)

wint_t wc;

Description

The iswblank subroutine tests whether the wc parameter is a wide-character code representing a character of class blank in the program’s current locale.

The wc parameter is a wint_t, the value of which the application ensures is a wide-character code corresponding to a valid character in the current locale, or equal to the value of the macro WEOF. If the parameter has any other value, the behavior is undefined.

Parameters

wc Specifies the value to be tested.

Return Values

The iswblank subroutine returns a nonzero value if the wc parameter is a blank wide-character code; otherwise, it returns a 0.
Related Information
“iswalnum, iswalpha, iswcntrl, iswdigit, iswgraph, iswlower, iswprint, iswpunct, iswspace, iswupper, or
iswxdigit Subroutine” on page 567 and “iswctype or is_wctype Subroutine.”

setlocale Subroutine in AIX 5L Version 5.3 Technical Reference: Base Operating System and Extensions
Volume 2.

wctype.h in AIX 5L Version 5.3 Files Reference.

iswctype or is_wctype Subroutine

Purpose
Determines properties of a wide character.

Library
Standard C Library (libc.a)

Syntax
#include <wchar.h>

int iswctype ( WC, Property)
  wint_t WC;
  wctype_t Property;

int is_wctype ( WC, Property)
  wint_t WC;
  wctype_t Property;

Description
The iswctype subroutine tests the wide character specified by the WC parameter to determine if it has the
property specified by the Property parameter. The iswctype subroutine is defined for the wide-character
null value and for values in the character range of the current code set, defined in the current locale. The
is_wctype subroutine is identical to the iswctype subroutine.

The iswctype subroutine adheres to X/Open Portability Guide Issue 5.

Parameters
WC Specifies the wide character to be tested.
Property Specifies the property for which to test.

Return Values
If the WC parameter has the property specified by the Property parameter, the iswctype subroutine
returns a nonzero value. If the value specified by the WC parameter does not have the property specified
by the Property parameter, the iswctype subroutine returns a value of zero. If the value specified by the
WC parameter is not in the subroutine’s domain, the result is undefined. If the value specified by the
Property parameter is not valid (that is, not obtained by a call to the wctype subroutine, or the Property
parameter has been invalidated by a subsequent call to the setlocale subroutine that has affected the
LC_CTYPE category), the result is undefined.
jcode Subroutines

Purpose
Perform string conversion on 8-bit processing codes.

Library
Standard C Library (libc.a)

Syntax
#include <jcode.h>

char *jistosj(String1, String2)
char *jistouj(String1, String2)
char *sjtojis(String1, String2)
char *sjtouj(String1, String2)
char *ujtojis(String1, String2)
char *ujtosj(String1, String2)
char *cjistosj(String1, String2)
char *cjistouj(String1, String2)
char *csjtojis(String1, String2)
char *csjtoj(String1, String2)
char *cujtojis(String1, String2)
char *cujtosj(String1, String2)

Description
The jistosj, jistouj, sjtojis, sjtouj, ujtojis, and ujtosj subroutines perform string conversion on 8-bit processing codes. The String2 parameter is converted and the converted string is stored in the String1 parameter. The overflow of the String1 parameter is not checked. Also, the String2 parameter must be a valid string. Code validation is not permitted.
The \texttt{jistosj} subroutine converts JIS to SJIS. The \texttt{jistouj} subroutine converts JIS to UJIS. The \texttt{sjtojis} subroutine converts SJIS to JIS. The \texttt{sjtouj} subroutine converts SJIS to UJIS. The \texttt{ujtojis} subroutine converts UJIS to JIS. The \texttt{ujtosj} subroutine converts UJIS to SJIS.

The \texttt{cjistosj}, \texttt{cjistouj}, \texttt{csjtojis}, \texttt{csjtouj}, \texttt{cujtojis}, and \texttt{cujtosj} macros perform code conversion on 8-bit processing JIS Kanji characters. A character is removed from the \texttt{String2} parameter, and its code is converted and stored in the \texttt{String1} parameter. The \texttt{String1} parameter is returned. The validity of the \texttt{String2} parameter is not checked.

The \texttt{cjistosj} macro converts from JIS to SJIS. The \texttt{cjistouj} macro converts from JIS to UJIS. The \texttt{csjtojis} macro converts from SJIS to JIS. The \texttt{csjtouj} macro converts from SJIS to UJIS. The \texttt{cujtojis} macro converts from UJIS to JIS. The \texttt{cujtosj} macro converts from UJIS to SJIS.

**Parameters**

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>\texttt{String1}</td>
<td>Stores converted string or code.</td>
</tr>
<tr>
<td>\texttt{String2}</td>
<td>Stores string or code to be converted.</td>
</tr>
</tbody>
</table>

**Related Information**

The "Japanese conv Subroutines" and "Japanese ctype Subroutines" on page 573.

List of String Manipulation Services in \textit{AIX 5L Version 5.3 General Programming Concepts: Writing and Debugging Programs.}


Subroutines, Example Programs, and Libraries in \textit{AIX 5L Version 5.3 General Programming Concepts: Writing and Debugging Programs.}

---

**Japanese conv Subroutines**

**Purpose**

Translates predefined Japanese character classes.

**Library**

Standard C Library (\texttt{libc.a})

**Syntax**

```c
#include <ctype.h>
int atojis (Character)
int Character;

int jistoa (Character)
int Character;

int _atojis (Character)
int Character;

int _jistoa (Character)
int Character;
```
int tojupper (Character)
int Character;

int tojlower (Character)
int Character;

int _tojupper (Character)
int Character;

int _tojlower (Character)
int Character;

int toujis (Character)
int Character;

int kutentojis (Character)
int Character;

int tojhira (Character)
int Character;

int tojkata (Character)
int Character;

Description
When running the operating system with Japanese Language Support on your system, the legal value of
the Character parameter is in the range from 0 to NLCOLMAX.

The jistoa subroutine converts an SJIS ASCII equivalent to the corresponding ASCII equivalent. The
atojis subroutine converts an ASCII character to the corresponding SJIS equivalent. Other values are
returned unchanged.

The _jistoa and _atojis routines are macros that function like the jistoa and atojis subroutines, but are
faster and have no error checking function.

The tojlower subroutine converts a SJIS uppercase letter to the corresponding SJIS lowercase letter. The
tojupper subroutine converts an SJIS lowercase letter to the corresponding SJIS uppercase letter. All
other values are returned unchanged.

The _tojlower and _tojupper routines are macros that function like the tojlower and tojupper
subroutines, but are faster and have no error-checking function.

The toujis subroutine sets all parameter bits that are not 16-bit SJIS code to 0.

The kutentojis subroutine converts a kuten code to the corresponding SJIS code. The kutentojis routine
returns 0 if the given kuten code is invalid.

The tojhira subroutine converts an SJIS katakana character to its SJIS hiragana equivalent. Any value
that is not an SJIS katakana character is returned unchanged.

The tojkata subroutine converts an SJIS hiragana character to its SJIS katakana equivalent. Any value
that is not an SJIS hiragana character is returned unchanged.

The _tojhira and _tojkata subroutines attempt the same conversions without checking for valid input.
For all functions except the **toujis** subroutine, the out-of-range parameter values are returned without conversion.

**Parameters**

- **Character**
  - Character to be converted.
- **Pointer**
  - Pointer to the escape sequence.
- **CharacterPointer**
  - Pointer to a single **NLchar** data type.

**Related Information**

The **ctype, isalpha, isupper, islower, isdigit, isxdigit, isalnum, isspace, ispunct, isprint, isgraph, iscntrl, or isascii Subroutines** on page 208, **conv Subroutines** on page 183, **getc, getchar, fgetc, or getw Subroutine** on page 343, **getwc, fgetwc, or getwchar Subroutine** on page 472, and **setlocale subroutine**.

List of Character Manipulation Subroutines and Subroutines, Example Programs, and Libraries in AIX 5L Version 5.3 General Programming Concepts: Writing and Debugging Programs.

**Japanese ctype Subroutines**

**Purpose**

Classify characters.

**Library**

Standard Character Library (**libc.a**)

**Syntax**

```c
#include <ctype.h>

int isjalpha (Character)
int Character;

int isjupper (Character)
int Character;

int isjlower (Character)
int Character;

int isjbytekana (Character)
int Character;

int isjdigit (Character)
int Character;

int isjxdigit (Character)
int Character;

int isjalnum (Character)
int Character;
```

Base Operating System (BOS) Runtime Services (A-P)  573
Description
The Japanese ctype subroutines classify character-coded integer values specified in a table. Each of these subroutines returns a nonzero value for True and 0 for False.

Parameters
Character Character to be tested.

Return Values
The isjprint and isjgraph subroutines return a 0 value for user-defined characters.

Related Information
The ctype, isalpha, isupper, islower, isdigit, isxdigit, isalnum, isspace, ispunct, isprint, isgraph, iscntrl, or isascii Subroutines on page 208, and setlocale subroutine.

List of Character Manipulation Services and Subroutines, Example Programs, and Libraries in AIX 5L Version 5.3 General Programming Concepts: Writing and Debugging Programs.

**kill or killpg Subroutine**

**Purpose**
Sends a signal to a process or to a group of processes.

**Library**
Standard C Library (*libc.a*)

**Syntax**
```c
#include <sys/types.h>
#include <signal.h>

int kill(
    Process,  
    Signal)  
pid_t Process;
int Signal;

int killpg(
    ProcessGroup, Signal)  
int ProcessGroup, Signal;
```

**Description**
The `kill` subroutine sends the signal specified by the `Signal` parameter to the process or group of processes specified by the `Process` parameter.

To send a signal to another process, either the real or the effective user ID of the sending process must match the real or effective user ID of the receiving process, and the calling process must have root user authority.

The processes that have the process IDs of 0 and 1 are special processes and are sometimes referred to here as `proc0` and `proc1`, respectively.

Processes can send signals to themselves.

**Note:** Sending a signal does not imply that the operation is successful. All signal operations must pass the access checks prescribed by each enforced access control policy on the system.

The following interface is provided for BSD Compatibility:
```c
int killpg(ProcessGroup, Signal)  
int ProcessGroup, Signal;
```

This interface is equivalent to:
```c
if (ProcessGroup < 0)  
{    
    errno = ESRCH;    
    return (-1);  
}  
return (kill(-ProcessGroup, Signal));
```
Parameters

Process
Specifies the ID of a process or group of processes.

If the Process parameter is greater than 0, the signal specified by the Signal parameter is sent to the process identified by the Process parameter.

If the Process parameter is 0, the signal specified by the Signal parameter is sent to all processes, excluding proc0 and proc1, whose process group ID matches the process group ID of the sender.

If the value of the Process parameter is a negative value other than -1 and if the calling process passes the access checks for the process to be signaled, the signal specified by the Signal parameter is sent to all processes, excluding proc0 and proc1, whose process group ID matches the absolute value of the Process parameter.

If the value of the Process parameter is -1, the signal specified by the Signal parameter is sent to all processes which the process has permission to send that signal.

Signal
Specifies the signal. If the Signal parameter is a null value, error checking is performed but no signal is sent. This parameter is used to check the validity of the Process parameter.

ProcessGroup
Specifies the process group.

Return Values
Upon successful completion, the kill subroutine returns a value of 0. Otherwise, a value of -1 is returned and the errno global variable is set to indicate the error.

Error Codes
The kill subroutine is unsuccessful and no signal is sent if one or more of the following are true:

EINVAL
The Signal parameter is not a valid signal number.
EINVAL
The Signal parameter specifies the SIGKILL, SIGSTOP, SIGTSTP, or SIGCONT signal, and the Process parameter is 1 (proc1).
ESRCH
No process can be found corresponding to that specified by the Process parameter.
EPERM
The real or effective user ID does not match the real or effective user ID of the receiving process, or else the calling process does not have root user authority.

Related Information
The getpid, getpgrp, or getppid subroutine, sigaction, sigvec, or signal subroutine.

Signal Management in AIX 5L Version 5.3 General Programming Concepts: Writing and Debugging Programs provides more information about signal management in multi-threaded processes.

kleenup Subroutine

Purpose
Cleans up the run-time environment of a process.
Library

Syntax

```c
int kleenup(int FileDescriptor, int SigIgn, int SigKeep);
int FileDescriptor;
int SigIgn[];
int SigKeep[];
```

Description

The `kleenup` subroutine cleans up the run-time environment for a trusted process by:

- Closing unnecessary file descriptors.
- Resetting the alarm time.
- Resetting signal handlers.
- Clearing the value of the `real directory read` flag described in the `ulimit` subroutine.
- Resetting the `ulimit` value, if it is less than a reasonable value (8192).

Parameters

- **FileDescriptor**
  Specifies a file descriptor. The `kleenup` subroutine closes all file descriptors greater than or equal to the `FileDescriptor` parameter.

- **SigIgn**
  Points to a list of signal numbers. If these are nonnull values, this list is terminated by 0s. Any signals specified by the `SigIgn` parameter are set to `SIG_IGN`. The handling of all signals not specified by either this list or the `SigKeep` list are set to `SIG_DFL`. Some signals cannot be reset and are left unchanged.

- **SigKeep**
  Points to a list of signal numbers. If these are nonnull values, this list is terminated by 0s. The handling of any signals specified by the `SigKeep` parameter is left unchanged. The handling of all signals not specified by either this list or the `SigIgn` list are set to `SIG_DFL`. Some signals cannot be reset and are left unchanged.

Return Values

The `kleenup` subroutine is always successful and returns a value of 0. Errors in closing files are not reported. It is not an error to attempt to modify a signal that the process is not allowed to handle.

Related Information

The `ulimit` subroutine.

List of Security and Auditing Subroutines and Subroutines Overview in AIX 5L Version 5.3 General Programming Concepts: Writing and Debugging Programs.

knlist Subroutine

Purpose

Translates names to addresses in the running system.

Syntax

```c
#include <nlist.h>
```
int knlist(NList, NumberOfElements, Size);

struct nlist *NList;
int NumberOfElements;
int Size;

Description
The knlist subroutine allows a program to examine the list of symbols exported by kernel routines to other kernel modules.

The first field in the nlist structure is an input parameter to the knlist subroutine. The n_value field is modified by the knlist subroutine, and all the others remain unchanged. The nlist structure consists of the following fields:

- char *n_name: Specifies the name of the symbol whose attributes are to be retrieved.
- long n_value: Indicates the virtual address of the object. This will also be the offset when using segment descriptor 0 as the extension parameter of the readx or writex subroutines against the /dev/mem file.

If the name is not found, all fields, other than n_name, are set to 0.

The nlist.h file is automatically included by the a.out.h file for compatibility. However, do not include the a.out.h file if you only need the information necessary to use the knlist subroutine. If you do include the a.out.h file, follow the #include statement with the line:

```
#undef n_name
```

Notes:
1. If both the nlist.h and netdb.h files are to be included, the netdb.h file should be included before the nlist.h file in order to avoid a conflict with the n_name structure member. Likewise, if both the a.out.h and netdb.h files are to be included, the netdb.h file should be included before the a.out.h file to avoid a conflict with the n_name structure.

2. If the netdb.h file and either the nlist.h or syms.h file are included, the n_name field will be defined as _n._n_name. This definition allows you to access the n_name field in the nlist or syment structure. If you need to access the n_name field in the netent structure, undefine the n_name field by entering:

```
#undef n_name
```

before accessing the n_name field in the netent structure. If you need to access the n_name field in a syment or nlist structure after undefining it, redefine the n_name field with:

```
#define n_name _n._n_name
```

Parameters

- NList: Points to an array of nlist structures.
- NumberOfElements: Specifies the number of structures in the array of nlist structures.
- Size: Specifies the size of each structure.

Return Values
Upon successful completion, knlist returns a value of 0. Otherwise, a value of -1 is returned, and the errno global variable is set to indicate the error.

Error Codes
The knlist subroutine fails when the following is true:

- EFAULT: Indicates that the NList parameter points outside the limit of the array of nlist structures.
kpidstate Subroutine

Purpose
Returns the status of a process.

Syntax

```c
kpidstate(pid)
pid_t pid;
```

Description
The kpidstate subroutine returns the state of a process specified by the pid parameter. The kpidstate subroutine can only be called by a process.

Parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>pid</td>
<td>Specifies the product ID.</td>
</tr>
</tbody>
</table>

Return Values
If the pid parameter is not valid, KP_NOTFOUND is returned. If the pid parameter is valid, the following settings in the process state determine what is returned:

- SNONE: Return KP_NOTFOUND.
- SIDL: Return KP_INITING.
- SZOMB: Return KP_EXITING, also if SEXIT in pv_flag.
- SSTOP: Return KP_STOPPED.

Otherwise the pid is alive and KP_ALIVE is returned.

Error Codes

_lazySetErrorHandler Subroutine

Purpose
Installs an error handler into the lazy loading runtime system for the current process.

Library
Standard C Library (libc.a)

Syntax

```c
#include <sys/ldr.h>
#include <sys/errno.h>
typedef void (*_handler_t)(
    char *module,
    char *symbol,
    unsigned int _errVal )();

handler_t *_lazySetErrorHandler(err_handler)
handler_t *err_handler;
```
**Description**

This function allows a process to install a custom error handler to be called when a lazy loading reference fails to find the required module or function. This function should only be used when the main program or one of its dependent modules was linked with the `-blazy` option. To call `_lazySetErrorHandler` from a module that is not linked with the `-blazy` option, you must use the `-lrtl` option. If you use `-blazy`, you do not need to specify `-lrtl`.

This function is not thread safe. The calling program should ensure that `_lazySetErrorHandler` is not called by multiple threads at the same time.

The user-supplied error handler may print its own error message, provide a substitute function to be used in place of the called function, or call the `longjmp` subroutine. To provide a substitute function that will be called instead of the originally referenced function, the error handler should return a pointer to the substitute function. This substitute function will be called by all subsequent calls to the intended function from the same module. If the value returned by the error handler appears to be invalid (for example, a NULL pointer), the default error handler will be used.

Each calling module resolves its lazy references independent of other modules. That is, if module A and B both call `foo` subroutine in module C, but module C does not export `foo` subroutine, the error handler will be called once when `foo` subroutine is called for the first time from A, and once when `foo` subroutine is called for the first time from B.

The default lazy loading error handler will print a message containing: the name of module that the program required; the name of the symbol being accessed; and the error value generated by the failure. Since the default handler considers a lazy load error to be fatal, the process will exit with a status of 1.

During execution of a program that utilizes lazy loading, there are a few conditions that may cause an error to occur. In all cases the current error handler will be called.

1. The referenced module (which is to be loaded upon function invocation) is unavailable or cannot be loaded. The `errVal` parameter will probably indicate the reason for the error if a system call failed.
2. A function is referenced, but the loaded module does not contain a definition for the function. In this case, `errVal` parameter will be `EINVAL`.

Some possibilities as to why either of these errors might occur:

1. The `LIBPATH` environment variable may contain a set of search paths that cause the application to load the wrong version of a module.
2. A module has been changed and no longer provides the same set of symbols that it did when the application was built.
3. The `load` subroutine fails due to a lack of resources available to the process.

**Parameters**

- `err_handler` A pointer to the new error handler function. The new function should accept 3 arguments:
  - `module` The name of the referenced module.
  - `symbol` The name of the function being called at the time the failure occurred.
  - `errVal` The value of `errno` at the time the failure occurred, if a system call used to load the module fails. For other failures, `errval` may be `EINVAL` or `ENOMEM`.

Note that the value of module or symbol may be NULL if the calling module has somehow been corrupted.

If the `err_handler` parameter is NULL, the default error handler is restored.
Return Value
The function returns a pointer to the previous user-supplied error handler, or NULL if the default error handler was in effect.

Related Information
The load ("load Subroutine" on page 721) subroutine.

The ld command.

The Shared Library Overview and Subroutines Overview in AIX 5L Version 5.3 General Programming Concepts.

The Shared Library and Lazy Loading in AIX 5L Version 5.3 General Programming Concepts.

l3tol or ltol3 Subroutine

Purpose
Converts between 3-byte integers and long integers.

Library
Standard C Library (libc.a)

Syntax

```c
void l3tol (LongPointer, CharacterPointer, Number)
long *LongPointer;
char *CharacterPointer;
int Number;
void ltol3 (CharacterPointer, LongPointer, Number)
char *CharacterPointer;
long *LongPointer;
int Number;
```

Description
The l3tol subroutine converts a list of the number of 3-byte integers specified by the Number parameter packed into a character string pointed to by the CharacterPointer parameter into a list of long integers pointed to by the LongPointer parameter.

The ltol3 subroutine performs the reverse conversion, from long integers (the LongPointer parameter) to 3-byte integers (the CharacterPointer parameter).

These functions are useful for file system maintenance where the block numbers are 3 bytes long.

Parameters

- **LongPointer**
  - Specifies the address of a list of long integers.
- **CharacterPointer**
  - Specifies the address of a list of 3-byte integers.
- **Number**
  - Specifies the number of list elements to convert.

Related Information
The filsys.h file format.
Subroutines Overview in AIX 5L Version 5.3 General Programming Concepts: Writing and Debugging Programs.

I64a_r Subroutine

Purpose
Converts base-64 long integers to strings.

Library
Thread-Safe C Library (libc_r.a)

Syntax
#include <stdlib.h>

int l64a_r (Convert, Buffer, Length)
long Convert;
char *Buffer;
int Length;

Description
The l64a_r subroutine converts a given long integer into a base-64 ASCII string.

Programs using this subroutine must link to the libpthread.a library.

For base-64 characters, the following ASCII characters are used:

<table>
<thead>
<tr>
<th>Character</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>.</td>
<td>Represents 0.</td>
</tr>
<tr>
<td>/</td>
<td>Represents 1.</td>
</tr>
<tr>
<td>0-9</td>
<td>Represents the numbers 2-11.</td>
</tr>
<tr>
<td>A-Z</td>
<td>Represents the numbers 12-37.</td>
</tr>
<tr>
<td>a-z</td>
<td>Represents the numbers 38-63.</td>
</tr>
</tbody>
</table>

The l64a_r subroutine places the converted base-64 string in the buffer pointed to by the Buffer parameter.

Parameters

Convert Specifies the long integer that is to be converted into a base-64 ASCII string.
Buffer Specifies a working buffer to hold the converted long integer.
Length Specifies the length of the Buffer parameter.

Return Values

0 Indicates that the subroutine was successful.
-1 Indicates that the subroutine was not successful. If the l64a_r subroutine is not successful, the errno global variable is set to indicate the error.
Error Codes
If the l64a_r subroutine is not successful, it returns the following error code:

EINVAL The Buffer parameter value is invalid or too small to hold the resulting ASCII string.

Related Information
Subroutines Overview in AIX 5L Version 5.3 General Programming Concepts: Writing and Debugging Programs.

List of Multithread Subroutines in AIX 5L Version 5.3 General Programming Concepts: Writing and Debugging Programs.

LAPI_Addr_get Subroutine

Purpose
Retrieves a function address that was previously registered using LAPI_Addr_set.

Library
Availability Library (liblapi_r.a)

C Syntax
```c
#include <lapi.h>

int LAPI_Addr_get(hndl, addr, addr_hndl)
  lapi_handle_t hndl;
  void **addr;
  int addr_hndl;
```

FORTRAN Syntax
```fortran
include 'lapif.h'

LAPI_ADDR_GET(hndl, addr, addr_hndl, ierror)
INTEGER hndl
INTEGER (KIND=LAPI_ADDR_TYPE) :: addr
INTEGER addr_hndl
INTEGER ierror
```

Description
Type of call: local address manipulation

Use this subroutine to get the pointer that was previously registered with LAPI and is associated with the index addr_hndl. The value of addr_hndl must be in the range 1 <= addr_hndl < LOC_ADDRTBL_SZ.

Parameters

INPUT

hndl Specifies the LAPI handle.

addr_hndl Specifies the index of the function address to retrieve. You should have previously registered the address at this index using LAPI_Addr_set. The value of this parameter must be in the range 1 <= addr_hndl < LOC_ADDRTBL_SZ.

OUTPUT
addr Returns a function address that the user registered with LAPI.

ierror Specifies a FORTRAN return code. This is always the last parameter.

C Examples
To retrieve a header handler address that was previously registered using `LAPI_Addr_set`:

```c
lapi_handle_t hndl; /* the LAPI handle */
void **addr; /* the address to retrieve */
int addr_hndl; /* the index returned from LAPI_Addr_set */

: :

addr_hndl = 1;
LAPI_Addr_get(hndl, &addr, addr_hndl);

/* addr now contains the address that was previously registered */
/* using LAPI_Addr_set */
```

Return Values

- **LAPI_SUCCESS** Indicates that the function call completed successfully.
- **LAPI_ERR_ADDR_HNDL_RANGE** Indicates that the value of `addr_hndl` is not in the range \(1 \leq addr\_hndl < LOC\_ADDRTBL\_SZ\).
- **LAPI_ERR_HNDL_INVALID** Indicates that the `hndl` passed in is not valid (not initialized or in terminated state).
- **LAPI_ERR_RET_PTR_NULL** Indicates that the value of the `addr` pointer is NULL (in C) or that the value of `addr` is `LAPI_ADDR_NULL` (in FORTRAN).

Location

/etc/lib/liblapi_r.a

Related Information
Subroutines: `LAPI_Addr_set`, `LAPI_Qenv`

---

**LAPI_Addr_set** Subroutine

Purpose
Registers the address of a function.

Library
Availability Library (`liblapi_r.a`)

C Syntax

```c
#include <lapi.h>

int LAPI_Addr_set(hndl, addr, addr_hndl)
    lapi_handle_t hndl;
    void *addr;
    int addr_hndl;
```

584  Technical Reference, Volume 1: Base Operating System and Extensions
FORTRAN Syntax

#include 'lapif.h'

LAPI_ADDR_SET(hndl, addr, addr_hndl, ierror)

INTEGER hndl
INTEGER (KIND=LAPI_ADDR_TYPE) :: addr
INTEGER addr_hndl
INTEGER ierror

Description

Type of call: local address manipulation

Use this subroutine to register the address of a function (addr). LAPI maintains the function address in an internal table. The function address is indexed at location addr_hndl. In subsequent LAPI calls, addr_hndl can be used in place of addr. The value of addr_hndl must be in the range 1 <= addr_hndl < LOC_ADDRTBL_SZ.

For active message communication, you can use addr_hndl in place of the corresponding header handler address. LAPI only supports this indexed substitution for remote header handler addresses (but not other remote addresses, such as target counters or base data addresses). For these other types of addresses, the actual address value must be passed to the API call.

Parameters

INPUT

hndl Specifies the LAPI handle.
addr Specifies the address of the function handler that the user wants to register with LAPI.
addr_hndl Specifies a user function address that can be passed to LAPI calls in place of a header handler address. The value of this parameter must be in the range 1 <= addr_hndl < LOC_ADDRTBL_SZ.

OUTPUT

ierror Specifies a FORTRAN return code. This is always the last parameter.

C Examples

To register a header handler address:

lapi_handle_t hndl; /* the LAPI handle */
void *addr; /* the remote header handler address */
int addr_hndl; /* the index to associate */

...

addr = my_func;
addr_hndl = 1;
LAPI.Addr_set(hndl, addr, addr_hndl);

/* addr_hndl can now be used in place of addr in LAPI_Amsend, */
/* LAPI_Amsendv, and LAPI_Xfer calls */

...

Return Values

LAPI_SUCCESS Indicates that the function call completed successfully.
LAPI_ERR_ADDR_HNDL_RANGE
Indicates that the value of addr_hndl is not in the range 1 <= addr_hndl < LOC_ADDRTBL_SZ.

LAPI_ERR_HNDL_INVALID
Indicates that the hndl passed in is not valid (not initialized or in terminated state).

Location
/usr/lib/liblapi_r.a

Related Information
Subroutines: LAPI_Addr_get, LAPI_Amsend, LAPI_Amsendv, LAPI_Qenv, LAPI_Xfer

LAPI_Address Subroutine

Purpose
Returns an unsigned long value for a specified user address.

Library
Availability Library (liblapi_r.a)

C Syntax
#include <lapi.h>

int LAPI_Address(my_addr, ret_addr)
void *my_addr;
ulong *ret_addr;

Note: This subroutine is meant to be used by FORTRAN programs. The C version of LAPI_Address is provided for compatibility purposes only.

FORTRAN Syntax
include 'lapif.h'

LAPI_ADDRESS(my_addr, ret_addr, ierror)
INTEGER (KIND=any_type) :: my_addr
INTEGER (KIND=LAPI_ADDR_TYPE) :: ret_addr
INTEGER ierror

where:
any_type Is any FORTRAN datatype. This type declaration has the same meaning as the type void *
in C.

Description
Type of call: local address manipulation

Use this subroutine in FORTRAN programs when you need to store specified addresses in an array. In FORTRAN, the concept of address (&) does not exist as it does in C. LAPI_Address provides FORTRAN programmers with this function.

Parameters

INPUT
**my_addr** Specifies the address to convert. The value of this parameter cannot be NULL (in C) or **LAPI_ADDR_NULL** (in FORTRAN).

**OUTPUT**

**ret_addr** Returns the address that is stored in **my_addr** as an unsigned long for use in LAPI calls. The value of this parameter cannot be NULL (in C) or **LAPI_ADDR_NULL** (in FORTRAN).

**ierror** Specifies a FORTRAN return code. This is always the last parameter.

### FORTRAN Examples

To retrieve the address of a variable:

```fortran
! Contains the address of the target counter
integer (KIND=LAPI_ADDR_TYPE) :: cntr_addr

! Target Counter
type (LAPI_CNTR_T) :: tgt_cntr

! Return code
type :: ierror

call LAPI_ADDRESS(tgt_cntr, cntr_addr, ierror)

! cntr_addr now contains the address of tgt_cntr
```

### Return Values

- **LAPI_SUCCESS** Indicates that the function call completed successfully.
- **LAPI_ERR_ORG_ADDR_NULL** Indicates that the value of **my_addr** is NULL (in C) or **LAPI_ADDR_NULL** (in FORTRAN).
- **LAPI_ERR_TGT_ADDR_NULL** Indicates that the value of **ret_addr** is NULL (in C) or **LAPI_ADDR_NULL** (in FORTRAN).

### Location

/usr/lib/liblapi_r.a

### Related Information

Subroutines: **LAPI_Address_init**, **LAPI_Address_init64**

### LAPI_Address_init Subroutine

#### Purpose

Creates a remote address table.

#### Library

Availability Library (liblapi_r.a)
C Syntax
#include <lapi.h>

int LAPI_Address_init(hndl, my_addr, add_tab)
lapi_handle_t hndl;
void *my_addr;
void *add_tab[];

FORTRAN Syntax
include 'lapif.h'

LAPI_ADDRESS_INIT(hndl, my_addr, add_tab, ierror)
INTEGER hndl
INTEGER (KIND=LAPI_ADDR_TYPE) :: my_addr
INTEGER (KIND=LAPI_ADDR_TYPE) :: add_tab(*)
INTEGER ierror

Description
Type of call: collective communication (blocking)

LAPI_Address_init exchanges virtual addresses among tasks of a parallel application. Use this subroutine to create tables of such items as header handlers, target counters, and data buffer addresses.

LAPI_Address_init is a collective call over the LAPI handle hndl, which fills the table add_tab with the virtual address entries that each task supplies. Collective calls must be made in the same order at all participating tasks.

The addresses that are stored in the table add_tab are passed in using the my_addr parameter. Upon completion of this call, add_tab[i] contains the virtual address entry that was provided by task i. The array is opaque to the user.

Parameters

INPUT
hndl Specifies the LAPI handle.
my_addr Specifies the entry supplied by each task. The value of this parameter can be NULL (in C) or LAPI_ADDR_NULL (in FORTRAN).

OUTPUT
add_tab Specifies the address table containing the addresses that are to be supplied by all tasks. add_tab is an array of pointers, the size of which is greater than or equal to NUM_TASKS. The value of this parameter cannot be NULL (in C) or LAPI_ADDR_NULL (in FORTRAN).
ierror Specifies a FORTRAN return code. This is always the last parameter.

C Examples
To collectively transfer target counter addresses for use in a communication API call, in which all nodes are either 32-bit or 64-bit:

lapi_handle_t hndl; /* the LAPI handle */
void *addr_tbl[NUM_TASKS]; /* the table for all tasks' addresses */
lapi_cntr_t tgt_cntr; /* the target counter */

LAPI_Address_init(hndl, (void *)&tgt_cntr, addr_tbl);
/* for communication with task t, use addr_tbl[t] */
/* as the address of the target counter */

For a combination of 32-bit and 64-bit nodes, use LAPI_Address_init64.

**Return Values**

- **LAPI_SUCCESS**
  Indicates that the function call completed successfully.

- **LAPI_ERR_COLLECTIVE_PSS**
  Indicates that a collective call was made while in persistent subsystem (PSS) mode.

- **LAPI_ERR_HNDL_INVALID**
  Indicates that the `hndl` passed in is not valid (not initialized or in terminated state).

- **LAPI_ERR_RET_PTR_NULL**
  Indicates that the value of the add_tab pointer is NULL (in C) or that the value of `add_tab` is LAPI_ADDR_NULL (in FORTRAN).

**Location**

`/usr/lib/liblapi_r.a`

**Related Information**

Subroutines: LAPI_Address, LAPI_Address_init64

---

**LAPI_Address_init64 Subroutine**

**Purpose**

Creates a 64-bit remote address table.

**Library**

Availability Library (`liblapi_r.a`)

**C Syntax**

```c
#include <lapi.h>

int LAPI_Address_init64(hndl, my_addr, add_tab)
    lapi_handle_t hndl;
    lapi_long_t my_addr;
    lapi_long_t *add_tab;
```

**FORTRAN Syntax**

```
include 'lapif.h'

LAPI_ADDRESS_INIT64(hndl, my_addr, add_tab, ierror)
INTEGER hndl
INTEGER (KIND=LAPI_ADDR_TYPE) :: my_addr
INTEGER (KIND=LAPI_LONG_LONG_TYPE) :: add_tab(*)
INTEGER ierror
```

**Description**

Type of call: collective communication (blocking)
**LAPI_Address_init64** exchanges virtual addresses among a mixture of 32-bit and 64-bit tasks of a parallel application. Use this subroutine to create 64-bit tables of such items as header handlers, target counters, and data buffer addresses.

**LAPI_Address_init64** is a collective call over the LAPI handle *hndl*, which fills the 64-bit table *add_tab* with the virtual address entries that each task supplies. Collective calls must be made in the same order at all participating tasks.

The addresses that are stored in the table *add_tab* are passed in using the *my_addr* parameter. Upon completion of this call, *add_tab[i]* contains the virtual address entry that was provided by task *i*. The array is opaque to the user.

### Parameters

**INPUT**

- **hndl** Specifies the LAPI handle.
- **my_addr** Specifies the address entry that is supplied by each task. The value of this parameter can be NULL (in C) or **LAPI_ADDR_NULL** (in FORTRAN). To ensure 32-bit/64-bit interoperability, it is passed as a *lapi_long_t* type in C.

**OUTPUT**

- **add_tab** Specifies the 64-bit address table that contains the 64-bit values supplied by all tasks. *add_tab* is an array of type *lapi_long_t* (in C) or **LAPI_LONG_LONG_TYPE** (in FORTRAN). The size of *add_tab* is greater than or equal to **NUM_TASKS**. The value of this parameter cannot be NULL (in C) or **LAPI_ADDR_NULL** (in FORTRAN).
- **ierror** Specifies a FORTRAN return code. This is always the last parameter.

### C Examples

To collectively transfer target counter addresses for use in a communication API call with a mixed task environment (any combination of 32-bit and 64-bit):

```c
lapi_handle_t hndl; /* the LAPI handle */
lapi_long_t addr_tbl[NUM_TASKS]; /* the table for all tasks' addresses */
lapi_long_t tgt_cntr; /* the target counter */

... LAPI_Address_init64(hndl, (lapi_long_t)&tgt_cntr, addr_tbl);

/* For communication with task t, use addr_tbl[t] as the address */
/* of the target counter. For mixed (32-bit and 64-bit) jobs, */
/* use the LAPI_Xfer subroutine for communication. */
```

### Return Values

- **LAPI_SUCCESS** Indicates that the function call completed successfully.
- **LAPI_ERR_COLLECTIVE_PSS** Indicates that a collective call was made while in persistent subsystem (PSS) mode.
- **LAPI_ERR_HNDL_INVALID** Indicates that the *hndl* passed in is not valid (not initialized or in terminated state).
- **LAPI_ERR_RET_PTR_NULL** Indicates that the value of the *add_tab* pointer is NULL (in C) or that the value of *add_tab* is **LAPI_ADDR_NULL** (in FORTRAN).
Location
/usr/lib/liblapi_r.a

Related Information
Subroutines: LAPI_Address, LAPI_Address_init, LAPI_Xfer

LAPI_Amsend Subroutine

Purpose
Transfers a user message to a remote task, obtaining the target address on the remote task from a
user-specified header handler.

Library
Availability Library (liblapi_r.a)

C Syntax
#include <lapi.h>

typedef void (compl_hndlr_t) (hdl, user_info);

lapi_handle_t *hdl;       /* pointer to LAPI context passed in from LAPI_Amsend */
void *user_info;          /* buffer (user_info) pointer passed in */
/* from header handler (void *(hdr_hndlr_t)) */

typedef void *(hdr_hndlr_t)(hndl, uhdr, uhdr_len, msg_len, comp_h, user_info);

lapi_handle_t *hdl;       /* pointer to LAPI context passed in from LAPI_Amsend */
void *uhdr;               /* uhdr passed in from LAPI_Amsend */
uint *uhdr_len;           /* uhdr_len passed in from LAPI_Amsend */
ulong *msg_len;           /* udata_len passed in from LAPI_Amsend */
compl_hndlr_t **comp_h;   /* function address of completion handler */
/* (void (compl_hndlr_t)) that needs to be filled */
/* out by this header handler function. */
void **user_info;         /* pointer to the parameter to be passed */
/* in to the completion handler */

int LAPI_Amsend(hndl, tgt, hdr_hdl, uhdr, uhdr_len, udata, udata_len,
    tgt_cntr, org_cntr, cmpl_cntr)

lapi_handle_t hndl;
uint tgt;
void *hdr_hdl;
void *uhdr;
uint *uhdr_len;
void *udata;
ulong udata_len;
lapi_cntr_t *tgt_cntr;
lapi_cntr_t *org_cntr;
lapi_cntr_t *cmpl_cntr;

FORTRAN Syntax
include 'lapif.h'

INTEGER SUBROUTINE COMPL_H (hdl, user_info)
INTEGER hndl
INTEGER user_info
INTEGER FUNCTION HDR_HDL (hndl, uhdr, uhdr_len, msg_len, comp_h, user_info)
INTEGER hndl
INTEGER uhdr
INTEGER uhdr_len
INTEGER (KIND=LAPI_LONG_TYPE) :: msg_len
EXTERNAL INTEGER FUNCTION comp_h
TYPE (LAPI_ADDR_T) :: user_info

LAPI_Amsend(hndl, tgt, hdr_hdl, uhdr, uhdr_len, udata, udata_len,
tgt_cntr, org_cntr, cmpl_cntr, ierror)
INTEGER hndl
EXTERNAL INTEGER FUNCTION hdr_hdl
EXTERNAL INTEGER FUNCTION comp_h
EXTERNAL INTEGER FUNCTION hdr_hdl

Description

Type of call: point-to-point communication (non-blocking)

Use this subroutine to transfer data to a target task, where it is desirable to run a handler on the target
task before message delivery begins or after delivery completes. LAPI_Amsend allows the user to provide
a header handler and optional completion handler. The header handler is used to specify the target buffer
address for writing the data, eliminating the need to know the address on the origin task when the
subroutine is called.

User data (uhdr and udata) are sent to the target task. Once these buffers are no longer needed on the
origin task, the origin counter is incremented, which indicates the availability of origin buffers for
modification. Using the LAPI_Xfer call with the LAPI_AM_XFER type provides the same type of transfer,
with the option of using a send completion handler instead of the origin counter to specify buffer
availability.

Upon arrival of the first data packet at the target, the user’s header handler is invoked. Note that a header
handler must be supplied by the user because it returns the base address of the buffer in which LAPI will
write the data sent from the origin task (udata). See RSCT for AIX 5L: LAPI Programming Guide for an
optimization exception to this requirement that a buffer address be supplied to LAPI for single-packet
messages.

The header handler also provides additional information to LAPI about the message delivery, such as the
completion handler. LAPI_Amsend and similar calls (such as LAPI_Amsendv and corresponding
LAPI_Xfer transfers) also allow the user to specify their own message header information, which is
available to the header handler. The user may also specify a completion handler parameter from within the
header handler. LAPI will pass the information to the completion handler at execution.

Note that the header handler is run inline by the thread running the LAPI dispatcher. For this reason, the
header handler must be non-blocking because no other progress on messages will be made until it
returns. It is also suggested that execution of the header handler be simple and quick. The completion
handler, on the other hand, is normally enqueued for execution by a separate thread. It is possible to
request that the completion handler be run inline. See RSCT for AIX 5L: LAPI Programming Guide for
more information on inline completion handlers.

If a completion handler was not specified (that is, set to LAPI_ADDR_NULL in FORTRAN or its pointer
set to NULL in C), the arrival of the final packet causes LAPI to increment the target counter on the
remote task and send an internal message back to the origin task. The message causes the completion
counter (if it is not NULL in C or LAPI_ADDR_NULL in FORTRAN) to increment on the origin task.

If a completion handler was specified, the above steps take place after the completion handler returns. To
guarantee that the completion handler has executed on the target, you must wait on the completion
counter. See RSCT for AIX 5L: LAPI Programming Guide for a time-sequence diagram of events in a
LAPI_Amsend call.

User details

As mentioned above, the user must supply the address of a header handler to be executed on the target
upon arrival of the first data packet. The signature of the header handler is as follows:

```c
void *hdr_hndlr(lapi_handle_t *hndl, void *uhdr, uint *uhdr_len, ulong *msg_len,
                compl_hndlr_t **cmpl_hndlr, void **user_info);
```

The value returned by the header handler is interpreted by LAPI as an address for writing the user data
(udata) that was passed to the LAPI_Amsend call. The uhdr and uhdr_len parameters are passed by
LAPI into the header handler and contain the information passed by the user to the corresponding
parameters of the LAPI_Amsend call.

Use of LAPI_Addr_set

Remote addresses are commonly exchanged by issuing a collective LAPI_Address_init call within a few
steps of initializing LAPI. LAPI also provides the LAPI_Addr_set mechanism, whereby users can register
one or more header handler addresses in a table, associating an index value with each address. This
index can then be passed to LAPI_Amsend instead of an actual address. On the target side, LAPI will
use the index to get the header handler address. Note that, if all tasks use the same index for their header
handler, the initial collective communication can be avoided. Each task simply registers its own header
handler address using the well-known index. Then, on any LAPI_Amsend calls, the reserved index can be
passed to the header handler address parameter.

Role of the header handler

The user optionally returns the address of a completion handler function through the cmpl_hndlr parameter
and a completion handler parameter through the user_info parameter. The address passed through the
user_info parameter can refer to memory containing a datatype defined by the user and then cast to the
appropriate type from within the completion handler if desired.

The signature for a user completion handler is as follows:

```c
typedef void (*compl_hndlr_t)(lapi_handle_t *hndl, void *completion_param);
```

The argument returned by reference through the user_info member of the user’s header handler will be
passed to the completion_param argument of the user’s completion handler. See the C Examples for an
example of setting the completion handler and parameter in the header handler.

As mentioned above, the value returned by the header handler must be an address for writing the user
data sent from the origin task. There is one exception to this rule. In the case of a single-packet message,
LAPI passes the address of the packet in the receive FIFO, allowing the entire message to be consumed
within the header handler. In this case, the header handler should return NULL (in C) or
LAPI_ADDR_NULL (in FORTRAN) so that LAPI does not copy the message to a target buffer. See RSCT
for AIX 5L: LAPI Programming Guide for more information (including a sample header handler that uses
this method for fast retrieval of a single-packet message).

Passing additional information through lapi_return_info_t
LAPI allows additional information to be passed to and returned from the header handler by passing a pointer to `lapi_return_info_t` through the `msg_len` argument. On return from a header handler that is invoked by a call to `LAPI_Amsend`, the `ret_flags` member of `lapi_return_info_t` can contain one of these values: `LAPI_NORMAL` (the default), `LAPI_SEND_REPLY` (to run the completion handler inline), or `LAPI_LOCAL_STATE` (no reply is sent). The `dgsp_handle` member of `lapi_return_info_t` should not be used in conjunction with `LAPI_Amsend`.

For a complete description of the `lapi_return_info_t` type, see RSCT for AIX 5L: LAPI Programming Guide

**Inline execution of completion handlers**

Under normal operation, LAPI uses a separate thread for executing user completion handlers. After the final packet arrives, completion handler pointers are placed in a queue to be handled by this thread. For performance reasons, the user may request that a given completion handler be run inline instead of being placed on this queue behind other completion handlers. This mechanism gives users a greater degree of control in prioritizing completion handler execution for performance-critical messages.

LAPI places no restrictions on completion handlers that are run “normally” (that is, by the completion handler thread). Inline completion handlers should be short and should not block, because no progress can be made while the main thread is executing the handler. The user must use caution with inline completion handlers so that LAPI’s internal queues do not fill up while waiting for the handler to complete. I/O operations must not be performed with an inline completion handler.

**Parameters**

**INPUT**

- **hndl**: Specifies the LAPI handle.
- **tgt**: Specifies the task ID of the target task. The value of this parameter must be in the range $0 \leq tgt < \text{NUM\_TASKS}$.
- **hdr_hdl**: Specifies the pointer to the remote header handler function to be invoked at the target. The value of this parameter cannot be `NULL` (in C) or `LAPI\_ADDR\_NULL` (in FORTRAN).
- **uhdr**: Specifies the pointer to the user header data. This data will be passed to the user header handler on the target. If `uhdr_len` is 0, the value of this parameter can be `NULL` (in C) or `LAPI\_ADDR\_NULL` (in FORTRAN).
- **uhdr_len**: Specifies the length of the user’s header. The value of this parameter must be a multiple of the processor’s word size in the range $0 \leq \text{uhdr\_len} \leq \text{MAX\_UHDR\_SZ}$.
- **udata**: Specifies the pointer to the user data. If `udata_len` is 0, the value of this parameter can be `NULL` (in C) or `LAPI\_ADDR\_NULL` (in FORTRAN).
- **udata_len**: Specifies the length of the user data in bytes. The value of this parameter must be in the range $0 \leq \text{udata\_len} \leq \text{value of LAPI constant LAPI\_MAX\_MSG\_SZ}$.

**INPUT/OUTPUT**

- **tgt_cntr**: Specifies the target counter address. The target counter is incremented after the completion handler (if specified) completes or after the completion of data transfer. If the value of this parameter is `NULL` (in C) or `LAPI\_ADDR\_NULL` (in FORTRAN), the target counter is not updated.
- **org_cntr**: Specifies the origin counter address (in C) or the origin counter (in FORTRAN). The origin counter is incremented after data is copied out of the origin address (in C) or the origin (in
FORTRAN). If the value of this parameter is NULL (in C) or LAPI_ADDR_NULL (in FORTRAN), the origin counter is not updated.

**cmpl_cntr**

Specifies the counter at the origin that signifies completion of the completion handler. It is updated once the completion handler completes. If no completion handler is specified, the counter is incremented at the completion of message delivery. If the value of this parameter is NULL (in C) or LAPI_ADDR_NULL (in FORTRAN), the completion counter is not updated.

**OUTPUT**

ierror

Specifies a FORTRAN return code. This is always the last parameter.

**Return Values**

**LAPI_SUCCESS**

Indicates that the function call completed successfully.

**LAPI_ERR_DATA_LEN**

Indicates that the value of udata_len is greater than the value of LAPI constant LAPI_MAX_MSG_SZ.

**LAPI_ERR_HDR_HNDLR_NULL**

Indicates that the value of the hdr_hdl passed in is NULL (in C) or LAPI_ADDR_NULL (in FORTRAN).

**LAPI_ERR_HNDL_INVALID**

Indicates that the hndl passed in is not valid (not initialized or in terminated state).

**LAPI_ERR_ORG_ADDR_NULL**

Indicates that the value of the udata parameter passed in is NULL (in C) or LAPI_ADDR_NULL (in FORTRAN), but the value of udata_len is greater than 0.

**LAPI_ERR_TGT**

Indicates that the tgt passed in is outside the range of tasks defined in the job.

**LAPI_ERR_TGT_PURGED**

Indicates that the subroutine returned early because LAPI_Purge_totask() was called.

**LAPI_ERR_UHDR_LEN**

Indicates that the uhdr_len value passed in is greater than MAX_UHDR_SZ or is not a multiple of the processor’s doubleword size.

**LAPI_ERR_UHDR_NULL**

Indicates that the uhdr passed in is NULL (in C) or LAPI_ADDR_NULL (in FORTRAN), but uhdr_len is not 0.

**C Examples**

To send an active message and then wait on the completion counter:

```c
/* header handler routine to execute on target task */
void *hdr_hndlr(lapi_handle_t *hndl, void *uhdr, uint *uhdr_len,
               ulong *msg_len, compl_hndlr_t **cmpl_hndlr,
               void **user_info)
{
    /* set completion handler pointer and other information */
    /* return base address for LAPI to begin its data copy */
}

lapi_handle_t hndl;    /* the LAPI handle */
int task_id;           /* the LAPI task ID */
int num_tasks;         /* the total number of tasks */
void *hdr_hndlr_list[NUM_TASKS]; /* the table of remote header handlers */
int buddy;             /* the communication partner */
ulong cmpl_cntr;       /* the completion counter */
int data_buffer[DATA_LEN]; /* the data to transfer */
```
/* retrieve header handler addresses */
LAPI_Address_init(hndl, (void *)&hdr_hndlr, hdr_hndlr_list);

/*
** up to this point, all instructions have executed on all
** tasks. we now begin differentiating tasks.
*/
if (sender) { /* origin task */
    /* initialize data buffer, cmpl_cntr, etc. */
    .
    .
    /* synchronize before starting data transfer */
    LAPI_Gfence(hndl);
    LAPI_Amsend(hndl, buddy, (void *)hdr_hndlr_list[buddy], NULL,
               NULL, (char *)&data_buffer[0], DATA_LEN*(sizeof(int)),
               NULL, NULL, cmpl_cntr);
    /* Wait on completion counter before continuing. Completion */
    /* counter will update when message completes at target. */
} else { /* receiver */
    .
    .
    /* to match the origin's synchronization before data transfer */
    LAPI_Gfence(hndl);
}

For a complete program listing, see RSCT for AIX 5L: LAPI Programming Guide. Sample code illustrating the LAPI_Amsend call can be found in the LAPI sample files. See RSCT for AIX 5L: LAPI Programming Guide for more information about the sample programs that are shipped with LAPI.

Location
/usr/lib/liblapi_r.a

Related Information
Subroutines: LAPI_Addr_get, LAPI_Addr_set, LAPI_Getcntr, LAPI_Msgpoll, LAPI_Qenv, LAPI_Setcntr, LAPI_Waitcntr, LAPI_Xfer

LAPI_Amsendv Subroutine

Purpose
Transfers a user vector to a remote task, obtaining the target address on the remote task from a user-specified header handler.
Library
Availability Library (liblapi_r.a)

C Syntax
#include <lapi.h>

typedef void (compl_hndlr_t) (hndl, user_info);

lapi_handle_t *hndl; /* the LAPI handle passed in from LAPI_Amsendv */
void *user_info; /* the buffer (user_info) pointer passed in */

/* from vhdr_hndlr (void *(vhdr_hndlr_t)) */

typedef lapi_vec_t *(vhdr_hndlr_t) (hndl, uhdr, uhdr_len, len_vec, comp_h, user_info);

lapi_handle_t *hndl; /* pointer to the LAPI handle passed in from LAPI_Amsendv */
void *uhdr; /* uhdr passed in from LAPI_Amsendv */
uint *uhdr_len; /* uhdr_len passed in from LAPI_Amsendv */
ulong *len_vec[]; /* vector of lengths passed in LAPI_Amsendv */
compl_hndlr_t **comp_h; /* function address of completion handler */
/* (void (compl_hndlr_t)) that needs to be */
/* filled out by this header handler function */
void **user_info; /* pointer to the parameter to be passed */
/* in to the completion handler */

int LAPI_Amsendv(hndl, tgt, hdr_hdl, uhdr, uhdr_len, org_vec, tgt_cntr, org_cntr, cmpl_cntr);

FORTRAN Syntax
#include 'lapif.h'

INTEGER SUBROUTINE COMPL_H (hndl, user_info)
INTEGER hndl
INTEGER user_info(*)

INTEGER FUNCTION VHDR_HDL (hndl, uhdr, uhdr_len, len_vec, comp_h, user_info)
INTEGER hndl
INTEGER uhdr
INTEGER uhdr_len
INTEGER (KIND=LAPI_LONG_TYPE) :: len_vec
EXTERNAL INTEGER FUNCTION comp_h
TYPE (LAPI_ADDR_T) :: user_info

LAPI_AMSENDV(hndl, tgt, hdr_hdl, uhdr, uhdr_len, org_vec, tgt_cntr, org_cntr, cmpl_cntr, ierror)
INTEGER hndl
INTEGER tgt
EXTERNAL INTEGER FUNCTION hdr_hdl
INTEGER uhdr
INTEGER uhdr_len
TYPE (LAPI_VEC_T) :: org_vec
INTEGER (KIND=LAPI_ADDR_TYPE) :: tgt_cntr
TYPE (LAPI_CNTR_T) :: org_cntr
TYPE (LAPI_CNTR_T) :: cmpl_cntr
INTEGER ierror
**Description**

**Type of call:** point-to-point communication (non-blocking)

**LAPI_Amsendv** is the vector-based version of the **LAPI_Amsend** call. You can use it to specify multi-dimensional and non-contiguous descriptions of the data to transfer. Whereas regular LAPI calls allow the specification of a single data buffer address and length, the vector versions allow the specification of a vector of address and length combinations. Additional information is allowed in the data description on the origin task and the target task.

Use this subroutine to transfer a vector of data to a target task, when you want a handler to run on the target task before message delivery begins or after message delivery completes.

To use **LAPI_Amsendv**, you must provide a header handler, which returns the address of the target vector description that LAPI uses to write the data that is described by the origin vector. The header handler is used to specify the address of the vector description for writing the data, which eliminates the need to know the description on the origin task when the subroutine is called. The header handler is called upon arrival of the first data packet at the target.

Optionally, you can also provide a completion handler. The header handler provides additional information to LAPI about the message delivery, such as the completion handler. You can also specify a completion handler parameter from within the header handler. LAPI passes the information to the completion handler at execution.

With the exception of the address that is returned by the completion handler, the use of counters, header handlers, and completion handlers in **LAPI_Amsendv** is identical to that of **LAPI_Amsend**. In both cases, the user header handler returns information that LAPI uses for writing at the target. See **LAPI_Amsend** for more information. This section presents information that is specific to the vector version of the call (**LAPI_Amsendv**).

LAPI vectors are structures of type **lapi_vec_t**, defined as follows:

```c
typedef struct {
    lapi_vectype_t vec_type;
    uint num_vecs;
    void **info;
    ulong *len;
} lapi_vec_t;
```

vec_type is an enumeration that describes the type of vector transfer, which can be: **LAPI_GEN_GENERIC**, **LAPI_GEN_IOVECTOR**, or **LAPI_GEN_STRIDED_XFER**.

For transfers of type **LAPI_GEN_GENERIC** and **LAPI_GEN_IOVECTOR**, the fields are used as follows:

- **num_vecs** indicates the number of data vectors to transfer. Each data vector is defined by a base address and data length.
- **info** is the array of addresses.
- **len** is the array of data lengths.

For example, consider the following vector description:

```c
vec_type = LAPI_GEN_IOVECTOR
num_vecs = 3
info = {addr_0, addr_1, addr_2}
len = {len_0, len_1, len_2}
```

On the origin side, this example would tell LAPI to read len_0 bytes from addr_0, len_1 bytes from addr_1, and len_2 bytes from addr_2. As a target vector, this example would tell LAPI to write len_0 bytes to addr_0, len_1 bytes to addr_1, and len_2 bytes to addr_2.
Recall that vector transfers require an origin and target vector. For \texttt{LAPI\_Amsendv} calls, the origin vector is passed to the API call on the origin task. The address of the target vector is returned by the header handler.

For transfers of type \texttt{LAPI\_GEN\_GENERIC}, the target vector description must also have type \texttt{LAPI\_GEN\_GENERIC}. The contents of the \texttt{info} and \texttt{len} arrays are unrestricted in the generic case; the number of vectors and the length of vectors on the origin and target do not need to match. In this case, LAPI transfers a given number of bytes in noncontiguous buffers specified by the origin vector to a set of noncontiguous buffers specified by the target vector.

If the sum of target vector data lengths (say \texttt{TGT\_LEN}) is less than the sum of origin vector data lengths (say \texttt{ORG\_LEN}), only the first \texttt{TGT\_LEN} bytes from the origin buffers are transferred and the remaining bytes are discarded. If \texttt{TGT\_LEN} is greater than \texttt{ORG\_LEN}, all \texttt{ORG\_LEN} bytes are transferred. Consider the following example:

Origin vector:

```c
{  
    num_vecs = 3;  
    info = {orgaddr_0, orgaddr_1, orgaddr_2};  
    len = {5, 10, 5}  
}
```

Target vector:

```c
{  
    num_vecs = 4;  
    info = {tgtaddr_0, tgtaddr_1, tgtaddr_2, tgtaddr_3};  
    len = {12, 2, 4, 2}  
}
```

LAPI copies data as follows:

1. 5 bytes from orgaddr_0 to tgtaddr_0 (leaves 7 bytes of space at a 5-byte offset from tgtaddr_0)
2. 7 bytes from orgaddr_1 to remaining space in tgtaddr_0 (leaves 3 bytes of data to transfer from orgaddr_1)
3. 2 bytes from orgaddr_1 to tgtaddr_1 (leaves 1 byte to transfer from orgaddr_1)
4. 1 byte from orgaddr_1 followed by 3 bytes from orgaddr_2 to tgt_addr_2 (leaves 3 bytes to transfer from orgaddr_2)
5. 2 bytes from orgaddr_2 to tgtaddr_3

LAPI will copy data from the origin until the space described by the target is filled. For example:

Origin vector:

```c
{  
    num_vecs = 1;  
    info = {orgaddr_0};  
    len = {20}  
}
```

Target vector:

```c
{  
    num_vecs = 2;  
    info = {tgtaddr_0, tgtaddr_1};  
    len = {5, 10}  
}
```

LAPI will copy 5 bytes from orgaddr_0 to tgtaddr_0 and the next 10 bytes from orgaddr_0 to tgtaddr_1. The remaining 5 bytes from orgaddr_0 will not be copied.

For transfers of type \texttt{LAPI\_GEN\_IOVECTOR}, the lengths of the vectors must match and the target vector description must match the origin vector description. More specifically, the target vector description must:

- also have type \texttt{LAPI\_GEN\_IOVECTOR}
- have the same \texttt{num_vecs} as the origin vector
- initialize the info array with \texttt{num_vecs} addresses in the target address space. For LAPI vectors \texttt{origin\_vector} and \texttt{target\_vector} described similarly to the example above, data is copied as follows:
1. transfer origin_vector.len[0] bytes from the address at origin_vector.info[0] to the address at target_vector.info[0]
2. transfer origin_vector.len[1] bytes from the address at origin_vector.info[1] to the address at target_vector.info[1]
3. transfer origin_vector.len[n] bytes from the address at origin_vector.info[n] to the address at target_vector.info[n], for $n = 2$ to $n = [num_vecs-3]$
4. transfer origin_vector.len[num_vecs-2] bytes from the address at origin_vector.info[num_vecs-2] to the address at target_vector.info[num_vecs-2]
5. copy origin_vector.len[num_vecs-1] bytes from the address at origin_vector.info[num_vecs-1] to the address at target_vector.info[num_vecs-1]

**Strided vector transfers**

For transfers of type **LAPI_GEN_STRIDED_XFER**, the target vector description must match the origin vector description. Rather than specifying the set of address and length pairs, the *info* array of the origin and target vectors is used to specify a data block "template", consisting of a base address, block size and stride. LAPI thus expects the *info* array to contain three integers. The first integer contains the base address, the second integer contains the block size to copy, and the third integer contains the byte stride. In this case, *num_vecs* indicates the number of blocks of data that LAPI should copy, where the first block begins at the base address. The number of bytes to copy in each block is given by the block size and the starting address for all but the first block is given by previous address + stride. The total amount of data to be copied will be $num_vecs*block_size$. Consider the following example:

```c
Origin_vector {
    num_vecs = 3;
    info = {orgaddr, 5, 8}
}
```

Based on this description, LAPI will transfer 5 bytes from orgaddr, 5 bytes from orgaddr+8 and 5 bytes from orgaddr+16.

**Call details**

As mentioned above, counter and handler behavior in **LAPI_Amsendv** is nearly identical to that of **LAPI_Amsend**. A short summary of that behavior is provided here. See the **LAPI_Amsend** description for full details.

This is a non-blocking call. The calling task cannot change the *uhdr* (origin header) and *org_vec* data until completion at the origin is signaled by the *org_cntr* being incremented. The calling task cannot assume that the *org_vec* structure can be changed before the origin counter is incremented. The structure (of type **lapi_vec_t**) that is returned by the header handler cannot be modified before the target counter has been incremented. Also, if a completion handler is specified, it may execute asynchronously, and can only be assumed to have completed after the target counter increments (on the target) or the completion counter increments (at the origin).

The length of the user-specified header (*uhdr_len*) is constrained by the implementation-specified maximum value **MAX_UHDR_SZ**. *uhdr_len* must be a multiple of the processor’s doubleword size. To get the best bandwidth, *uhdr_len* should be as small as possible.

If the following requirement is not met, an error condition occurs:

- If a strided vector is being transferred, the size of each block must not be greater than the stride size in bytes.

LAPI does not check for any overlapping regions among vectors either at the origin or the target. If the overlapping regions exist on the target side, the contents of the target buffer are undefined after the operation.
Parameters

**hdl** Specifies the LAPI handle.

**tgt** Specifies the task ID of the target task. The value of this parameter must be in the range \( 0 \leq tgt < \text{NUM_TASKS} \).

**hdr_hdl** Points to the remote header handler function to be invoked at the target. The value of this parameter can take an address handle that had been previously registered using the \text{LAPI_Addr_set/LAPI_Addr_get} mechanism. The value of this parameter cannot be NULL (in C) or \text{LAPI_ADDR_NULL} (in FORTRAN).

**uhdr** Specifies the pointer to the local header (parameter list) that is passed to the handler function. If **uhdr_len** is 0, The value of this parameter can be NULL (in C) or \text{LAPI_ADDR_NULL} (in FORTRAN).

**uhdr_len** Specifies the length of the user's header. The value of this parameter must be a multiple of the processor's doubleword size in the range \( 0 \leq \text{uhdr_len} \leq \text{MAX_UHDR_SZ} \).

**org_vec** Points to the origin vector.

**tgt_cntr** Specifies the target counter address. The target counter is incremented after the completion handler (if specified) completes or after the completion of data transfer. If the value of this parameter is NULL (in C) or \text{LAPI_ADDR_NULL} (in FORTRAN), the target counter is not updated.

**org_cntr** Specifies the origin counter address (in C) or the origin counter (in FORTRAN). The origin counter is incremented after data is copied out of the origin address (in C) or the origin (in FORTRAN). If the value of this parameter is NULL (in C) or \text{LAPI_ADDR_NULL} (in FORTRAN), the origin counter is not updated.

**cmpl_cntr** Specifies the counter at the origin that signifies completion of the completion handler. It is updated once the completion handler completes. If no completion handler is specified, the counter is incremented at the completion of message delivery. If the value of this parameter is NULL (in C) or \text{LAPI_ADDR_NULL} (in FORTRAN), the completion counter is not updated.

**ierror** Specifies a FORTRAN return code. This is always the last parameter.

**INPUT/OUTPUT**

**tgt_cntr** Specifies the target counter address. The target counter is incremented after the completion handler (if specified) completes or after the completion of data transfer. If the value of this parameter is NULL (in C) or \text{LAPI_ADDR_NULL} (in FORTRAN), the target counter is not updated.

**org_cntr** Specifies the origin counter address (in C) or the origin counter (in FORTRAN). The origin counter is incremented after data is copied out of the origin address (in C) or the origin (in FORTRAN). If the value of this parameter is NULL (in C) or \text{LAPI_ADDR_NULL} (in FORTRAN), the origin counter is not updated.

**cmpl_cntr** Specifies the counter at the origin that signifies completion of the completion handler. It is updated once the completion handler completes. If no completion handler is specified, the counter is incremented at the completion of message delivery. If the value of this parameter is NULL (in C) or \text{LAPI_ADDR_NULL} (in FORTRAN), the completion counter is not updated.

**OUTPUT**

**ierror** Specifies a FORTRAN return code. This is always the last parameter.

**C Examples**

1. To send a \text{LAPI_GEN_IOVECTOR} using active messages:

```c
/* header handler routine to execute on target task */
lapi_vec_t *hdr_hndlr(lapi_handle_t *handle, void *uhdr, uint *uhdr_len,
                     ulong *len_vec[], compl_hdlr_t **completion_handler,
                     void **user_info)
{
    /* set up the vector to return to LAPI */
    /* * for a LAPI_GEN_IOVECTOR: num_vecs, vec_type, and len must all have */
    /* the same values as the origin vector. The info array should */
    /* contain the buffer addresses for LAPI to write the data */
    vec->num_vecs = NUM_VECS;
    vec->vec_type = LAPI_GEN_IOVECTOR;
    vec->len = (unsigned long *)malloc(NUM_VECS*sizeof(unsigned long));
    vec->info = (void **) malloc(NUM_VECS*sizeof(void *));
    for( i=0; i < NUM_VECS; i++ ) {
        vec->info[i] = (void *) &data_buffer[i];
        vec->len[i] = (unsigned long)(sizeof(int));
    }
    /* set completion handler pointer and other info */
    completion_handler = completion_hdlr;
    /* set the target counter address */
    tgt_cntr = address_of_target_counter;
    /* set the origin counter address */
    org_cntr = address_of_origin_counter;
    /* set the completion counter address */
    cmpl_cntr = address_of_completion_counter;
    /* other information */
    /* return success */
    return NULL;
}
```
The above example could also illustrate the LAPI_GEN_GENERIC type, with the following modifications:

- Both vectors would need LAPI_GEN_GENERIC as the vec_type.
- There are no restrictions on symmetry of number of vectors and lengths between the origin and target sides.

2. To send a LAPI_STRIDED_VECTOR using active messages:

   /* header handler routine to execute on target task */
   lapi_vec_t *hdr_hdlr(lapi_handle_t *handle, void *uhdr, uint *uhdr_len,
                        ulong *len_vec[], compl_hdlr_t **completion_handler,
                        void **user_info)
   {
      int block_size;    /* block size */
      int data_size;     /* stride */
      .
      .
      vec->num_vecs = NUM_VECS;    /* NUM_VECS = number of vectors to transfer */
      vec->vec_type = LAPI_GEN_STRIDED_XFER;    /* same as origin vector */
      /* see comments in origin vector setup for a description of how data */
      /* will be copied based on these settings. */
      vec->info[0] = buffer_address;    /* starting address for data copy */
      vec->info[1] = block_size;      /* bytes of data to copy */
      vec->info[2] = stride;         /* distance from copy block to copy block */
      .
return vec;
}
}
lapi_vec_t *vec; /* data for data transfer */
vec->num_vecs = NUM_VECS; /* NUM_VECS = number of vectors to transfer */
vec->vec_type = LAPI_GEN_STRIDED_XFER; /* must match that of the target vector */
vec->info[0] = buffer_address; /* starting address for data copy */
vec->info[1] = block_size; /* bytes of data to copy */
vec->info[2] = stride; /* distance from copy block to copy block */
/* data will be copied as follows: */
/* block_size bytes will be copied from buffer_address */
/* block_size bytes will be copied from buffer_address+stride */
/* block_size bytes will be copied from buffer_address+(2*stride) */
/* block_size bytes will be copied from buffer_address+(3*stride) */
/* block_size bytes will be copied from buffer_address+((NUM_VECS-1)*stride) */
/* if uhdr isn't used, uhdr should be NULL and uhdr_len should be 0 */
/* tgt_cntr, org_cntr and cmpl_cntr can all be NULL */
LAPI_Amsendv(hndl, tgt, (void *)hdr_hdl_list[buddy], uhdr, uhdr_len,
vec, tgt_cntr, org_cntr, cmpl_cntr);

For complete examples, see the sample programs shipped with LAPI.

**Return Values**

<table>
<thead>
<tr>
<th>Error Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>LAPI_SUCCESS</td>
<td>Indicates that the function call completed successfully.</td>
</tr>
<tr>
<td>LAPI_ERR_HDR_HNDLR_NULL</td>
<td>Indicates that the <code>hdr_hdl</code> passed in is NULL (in C) or LAPI_ADDR_NULL</td>
</tr>
<tr>
<td></td>
<td>(in FORTRAN).</td>
</tr>
<tr>
<td>LAPI_ERR_HNDL_INVALID</td>
<td>Indicates that the <code>hndl</code> passed in is not valid (not initialized or in</td>
</tr>
<tr>
<td></td>
<td>terminated state).</td>
</tr>
<tr>
<td>LAPI_ERR_ORG_EXTENT</td>
<td>Indicates that the <code>org_vec</code>'s extent (stride * num_vecs) is greater than</td>
</tr>
<tr>
<td></td>
<td>the value of LAPI constant LAPI_MAX_MSG_SZ.</td>
</tr>
<tr>
<td>LAPI_ERR_ORG_STRIDE</td>
<td>Indicates that the <code>org_vec</code> stride is less than block.</td>
</tr>
<tr>
<td>LAPI_ERR_ORG_VEC_ADDR</td>
<td>Indicates that the <code>org_vec-&gt;info[i]</code> is NULL (in C) or LAPI_ADDR_NULL (in</td>
</tr>
<tr>
<td></td>
<td>FORTRAN), but its length (<code>org_vec-&gt;len[i]</code>) is not 0.</td>
</tr>
<tr>
<td>LAPI_ERR_ORG_VEC_LEN</td>
<td>Indicates that the sum of <code>org_vec-&gt;len</code> is greater than the value of LAPI</td>
</tr>
<tr>
<td></td>
<td>constant LAPI_MAX_MSG_SZ.</td>
</tr>
</tbody>
</table>
**LAPI_ERR_ORG_VEC_NULL** Indicates that `org_vec` is NULL (in C) or **LAPI_ADDR_NULL** (in FORTRAN).

**LAPI_ERR_ORG_VEC_TYPE** Indicates that the `org_vec->vec_type` is not valid.

**LAPI_ERR_STRIDE_ORG_VEC_ADDR_NULL** Indicates that the strided vector address `org_vec->info[0]` is NULL (in C) or **LAPI_ADDR_NULL** (in FORTRAN).

**LAPI_ERR_TGT** Indicates that the `tgt` passed in is outside the range of tasks defined in the job.

**LAPI_ERR_TGT_PURGED** Indicates that the subroutine returned early because **LAPI_Purge_totask()** was called.

**LAPI_ERR_UHDR_LEN** Indicates that the `uhdr_len` value passed in is greater than **MAX_UHDR_SZ** or is not a multiple of the processor's doubleword size.

**LAPI_ERR_UHDR_NULL** Indicates that the `uhdr` passed in is NULL (in C) or **LAPI_ADDR_NULL** (in FORTRAN), but `uhdr_len` is not 0.

**Location**
/usrl/lib/liblapi_r.a

**Related Information**

**LAPI_Fence Subroutine**

**Purpose**
Enforces order on LAPI calls.

**Library**
Availability Library (liblapi_r.a)

**C Syntax**
```c
#include <lapi.h>

int LAPI_Fence(hndl)
    lapi_handle_t hndl;
```

**FORTRAN Syntax**
```fortran
include 'lapif.h'

LAPI_FENCE(hndl, ierror)
INTEGER hndl
INTEGER ierror
```

**Description**
**Type of call**: Local data synchronization (blocking) (may require progress on the remote task)

Use this subroutine to enforce order on LAPI calls. If a task calls **LAPI_Fence**, all the LAPI operations that were initiated by that task, before the fence using the LAPI context `hndl`, are guaranteed to complete at the target tasks. This occurs before any of its communication operations using `hndl`, initiated after the
**LAPI_Fence**, start transmission of data. This is a data fence which means that the data movement is complete. This is not an operation fence which would need to include active message completion handlers completing on the target.

**LAPI_Fence** may require internal protocol processing on the remote side to complete the fence request.

**Parameters**

**INPUT**

`hndl` Specifies the LAPI handle.

**OUTPUT**

`ierror` Specifies a FORTRAN return code. This is always the last parameter.

**Return Values**

**LAPI_SUCCESS** Indicates that the function call completed successfully.

**LAPI_ERR_HNDL_INVALID** Indicates that the `hndl` passed in is not valid (not initialized or in terminated state).

**C Examples**

To establish a data barrier in a single task:

```c
lapi_handle_t hndl; /* the LAPI handle */

... /* API communication call 1 */
/* API communication call 2 */
...

/* API communication call n */
LAPI_Fence(hndl);

/* all data movement from above communication calls has completed by this point */
/* any completion handlers from active message calls could still be running. */
```

**Location**

`/usr/lib/liblapi_r.a`

**Related Information**

Subroutines: **LAPI_Amsend**, **LAPI_Gfence**

---

**LAPI_Get Subroutine**

**Purpose**

Copies data from a remote task to a local task.

**Library**

Availability Library (`liblapi_r.a`)
C Syntax

#include <lapi.h>

int LAPI_Get(hndl, tgt, len, tgt_addr, org_addr, tgt_cntr, org_cntr)

lapi_handle_t hndl;
uint tgt;
ulong len;
void *tgt_addr;
void *org_addr;
lapi_cntr_t *tgt_cntr;
lapi_cntr_t *org_cntr;

FORTRAN Syntax

include 'lapif.h'

LAPI_GET(hndl, tgt, len, tgt_addr, org_addr, tgt_cntr, org_cntr, ierror)

INTEGER hndl
INTEGER tgt
INTEGER (KIND=LAPI_LONG_TYPE) :: len
INTEGER (KIND=LAPI_ADDR_TYPE) :: tgt_addr
INTEGER (KIND=LAPI_ADDR_TYPE) :: org_addr
INTEGER (KIND=LAPI_ADDR_TYPE) :: tgt_cntr
TYPE (LAPI_CNTR_T) :: org_cntr
INTEGER ierror

Description

Type of call: point-to-point communication (non-blocking)

Use this subroutine to transfer data from a remote (target) task to a local (origin) task. Note that in this case the origin task is actually the receiver of the data. This difference in transfer type makes the counter behavior slightly different than in the normal case of origin sending to target.

The origin buffer will still increment on the origin task upon availability of the origin buffer. But in this case, the origin buffer becomes available once the transfer of data is complete. Similarly, the target counter will increment once the target buffer is available. Target buffer availability in this case refers to LAPI no longer needing to access the data in the buffer.

This is a non-blocking call. The caller cannot assume that data transfer has completed upon the return of the function. Instead, counters should be used to ensure correct buffer addresses as defined above.

Note that a zero-byte message does not transfer data, but it does have the same semantic with respect to counters as that of any other message.

Parameters

INPUT

hndl Specifies the LAPI handle.

tgt Specifies the task ID of the target task that is the source of the data. The value of this parameter must be in the range 0 <= tgt < NUM_TASKS.

len Specifies the number of bytes of data to be copied. This parameter must be in the range 0 <= len <= the value of LAPI constant LAPI_MAX_MSG_SZ.

tgt_addr Specifies the target buffer address of the data source. If len is 0, The value of this parameter can be NULL (in C) or LAPI_ADDR_NULL (in FORTRAN).

INPUT/OUTPUT

tgt_cntr Specifies the target counter address. The target counter is incremented once the data
buffer on the target can be modified. If the value of this parameter is NULL (in C) or LAPI_ADDR_NULL (in FORTRAN), the target counter is not updated.

**org_cntr**  
Specifies the origin counter address (in C) or the origin counter (in FORTRAN). The origin counter is incremented after data arrives at the origin. If the value of this parameter is NULL (in C) or LAPI_ADDR_NULL (in FORTRAN), the origin counter is not updated.

**OUTPUT**  
**org_addr**  
Specifies the local buffer address into which the received data is copied. If \( len \) is 0, The value of this parameter can be NULL (in C) or LAPI_ADDR_NULL (in FORTRAN).

**ierror**  
Specifies a FORTRAN return code. This is always the last parameter.

### C Examples

```c
{  
  /* initialize the table buffer for the data addresses */  
  /* get remote data buffer addresses */  
  LAPI_Address_init(hndl,(void *)data_buffer,data_buffer_list);  
  .  
  .  
  LAPI_Get(hndl, tgt, (ulong) data_len, (void *) (data_buffer_list[tgt]),  
          (void *) data_buffer, tgt_cntr, org_cntr);  
  .  
  /* retrieve data_len bytes from address data_buffer_list[tgt] on task tgt. */  
  /* write the data starting at address data_buffer. Tgt_cntr and org_cntr  */  
  /* can be NULL. */  
}
```

### Return Values

- **LAPI_SUCCESS**  
  Indicates that the function call completed successfully.

- **LAPI_ERR_DATA_LEN**  
  Indicates that the value of \( udata.len \) is greater than the value of LAPI constant LAPI_MAX_MSG_SZ.

- **LAPI_ERR_HNDL_INVALID**  
  Indicates that the \( hndl \) passed in is not valid (not initialized or in terminated state).

- **LAPI_ERR_ORG_ADDR_NULL**  
  Indicates that the \( org_addr \) passed in is NULL (in C) or LAPI_ADDR_NULL (in FORTRAN), but \( len \) is greater than 0.

- **LAPI_ERR_TGT**  
  Indicates that the \( tgt \) passed in is outside the range of tasks defined in the job.

- **LAPI_ERR_TGT_ADDR_NULL**  
  Indicates that the \( tgt_addr \) passed in is NULL (in C) or LAPI_ADDR_NULL (in FORTRAN), but \( len \) is greater than 0.

- **LAPI_ERR_TGT_PURGED**  
  Indicates that the subroutine returned early because LAPI_Purge_totask() was called.

### Location

/usr/lib/liblapi_r.a

### Related Information

Subroutines: LAPI_Address_init, LAPI_Getcntr, LAPI_Put, LAPI_Qenv, LAPI_Waitcntr, LAPI_Xfer
LAPI_Getcntr Subroutine

Purpose
Gets the integer value of a specified LAPI counter.

Library
Availability Library (liblapi_r.a)

C Syntax
#include <lapi.h>

int LAPI_Getcntr(hndl, cntr, val)
lapi_handle_t hndl;
lapi_cntr_t *cntr;
int *val;

FORTRAN Syntax
include 'lapif.h'
LAPI_GETCNTR(hndl, cntr, val, ierror)
INTEGER hndl
TYPE (LAPI_CNTR_T) :: cntr
INTEGER val
INTEGER ierror

Description
Type of call: Local counter manipulation

This subroutine gets the integer value of cntr. It is used to check progress on hndl.

Parameters

INPUT

hndl Specifies the LAPI handle.

cntr Specifies the address of the counter. The value of this parameter cannot be NULL (in C) or LAPI_ADDR_NULL (in FORTRAN).

OUTPUT

val Returns the integer value of the counter cntr. The value of this parameter cannot be NULL (in C) or LAPI_ADDR_NULL (in FORTRAN).

ierror Specifies a FORTRAN return code. This is always the last parameter.

C Examples

{   lapi_cntr_t cntr;
    int val;

    /* cntr is initialized */

    /* processing/communication takes place */
LAPI_Getcntr(hndl, &cntr, &val)

/* val now contains the current value of cntr */
}

Return Values

LAPI_SUCCESS Indicates that the function call completed successfully.
LAPI_ERR_CNTR_NULL Indicates that the cntr pointer is NULL (in C) or that the value of cntr is LAPI_ADDR_NULL (in FORTRAN).
LAPI_ERR_HNDL_INVALID Indicates that the hndl passed in is not valid (not initialized or in terminated state).
LAPI_ERR_RET_PTR_NULL Indicates that the value of the val pointer is NULL (in C) or that the value of val is LAPI_ADDR_NULL (in FORTRAN).

Location

/usr/lib/liblapi_r.a

Related Information

Subroutines: LAPI_Amsend, LAPI_Amsendv, LAPI_Get, LAPI_Getv, LAPI_Put, LAPI_Putv, LAPI_Rmw, LAPI_Setcntr, LAPI_Waitcntr, LAPI_Xfer

LAPI_Getv Subroutine

Purpose
Copies vectors of data from a remote task to a local task.

Library
Availability Library (liblapi_r.a)

C Syntax
#include <lapi.h>

int LAPI_Getv(hndl, tgt, tgt_vec, org_vec, tgt_cntr, org_cntr)

lapi_handle_t hndl;
uint tgt;
lapi_vec_t *tgt_vec;
lapi_vec_t *org_vec;
lapi_cntr_t *tgt_cntr;
lapi_cntr_t *org_cntr;

typedef struct {
    lapi_vectype_t vec_type; /* operation code */
    uint num_vecs; /* number of vectors */
    void **info; /* vector of information */
    ulong *len; /* vector of lengths */
} lapi_vec_t;

FORTRAN Syntax

include 'lapif.h'

LAPI_GETV(hndl, tgt, tgt_vec, org_vec, tgt_cntr, org_cntr, ierror)

INTEGER hndl
INTEGER tgt
The 32-bit version of the `LAPI_VEC_T` type is defined as:

```
TYPE LAPI_VEC_T
  SEQUENCE
    INTEGER(KIND = 4) :: vec_type
    INTEGER(KIND = 4) :: num_vecs
    INTEGER(KIND = 4) :: info
    INTEGER(KIND = 4) :: len
END TYPE LAPI_VEC_T
```

The 64-bit version of the `LAPI_VEC_T` type is defined as:

```
TYPE LAPI_VEC_T
  SEQUENCE
    INTEGER(KIND = 4) :: vec_type
    INTEGER(KIND = 4) :: num_vecs
    INTEGER(KIND = 8) :: info
    INTEGER(KIND = 8) :: len
END TYPE LAPI_VEC_T
```

### Description

**Type of call:** point-to-point communication (non-blocking)

This subroutine is the vector version of the `LAPI_Get` call. Use `LAPI_Getv` to transfer vectors of data from the target task to the origin task. Both the origin and target vector descriptions are located in the address space of the origin task. But, the values specified in the `info` array of the target vector must be addresses in the address space of the target task.

The calling program *cannot* assume that the origin buffer can be changed or that the contents of the origin buffers on the origin task are ready for use upon function return. After the origin counter (`org_cntr`) is incremented, the origin buffers can be modified by the origin task. After the target counter (`tgt_cntr`) is incremented, the target buffers can be modified by the target task. If you provide a completion counter (`cmpl_cntr`), it is incremented at the origin after the target counter (`tgt_cntr`) has been incremented at the target. If the values of any of the counters or counter addresses are NULL (in C) or `LAPI_ADDR_NULL` (in FORTRAN), the data transfer occurs, but the corresponding counter increments do not occur.

If any of the following requirements are not met, an error condition occurs:

- The vector types `org_vec`->`vec_type` and `tgt_vec`->`vec_type` must be the same.
- If a strided vector is being transferred, the size of each block must not be greater than the stride size in bytes.
- The length of any vector that is pointed to by `tgt_vec` must be equal to the length of the corresponding vector that is pointed to by `org_vec`.

LAPI does not check for any overlapping regions among vectors either at the origin or the target. If the overlapping regions exist on the origin side, the contents of the origin buffer are undefined after the operation.

See `LAPI_Amsendv` for details about communication using different LAPI vector types. (`LAPI_Getv` does not support the `LAPI_GEN_GENERIC` type.)

### Parameters

**INPUT**
**hndl**
Specifies the LAPI handle.

**tgt**
Specifies the task ID of the target task. The value of this parameter must be in the range 0 <= tgt < NUM_TASKS.

**tgt_vec**
Points to the target vector description.

**org_vec**
Points to the origin vector description.

**INPUT/OUTPUT**

**tgt_cntr**
Specifies the target counter address. The target counter is incremented once the data buffer on the target can be modified. If the value of this parameter is NULL (in C) or LAPI_ADDR_NULL (in FORTRAN), the target counter is not updated.

**org_cntr**
Specifies the origin counter address (in C) or the origin counter (in FORTRAN). The origin counter is incremented after data arrives at the origin. If the value of this parameter is NULL (in C) or LAPI_ADDR_NULL (in FORTRAN), the origin counter is not updated.

**OUTPUT**

**ierror**
Specifies a FORTRAN return code. This is always the last parameter.

**C Examples**
To get a LAPI_GEN_IOVECTOR:

```c
{
    ...
    LAPI_Getv(hndl, tgt, tgt_vec, org_vec, tgt_cntr, org_cntr);
    ...
}
```

Base Operating System (BOS) Runtime Services (A-P)  611
For examples of other vector types, see LAPI_Amsendv.

**Return Values**

- **LAPI_SUCCESS** Indicates that the function call completed successfully.
- **LAPI_ERR_HNDL_INVALID** Indicates that the \textit{hndl} passed in is not valid (not initialized or in terminated state).
- **LAPI_ERR_ORG_EXTENT** Indicates that the \textit{org_vec}'s extent (stride \* \textit{num_vecs}) is greater than the value of LAPI constant \texttt{LAPI\_MAX\_MSG\_SZ}.
- **LAPI_ERR_ORG_STRIDE** Indicates that the \textit{org_vec} stride is less than block size.
- **LAPI_ERR_ORG_VEC_ADDR** Indicates that some \textit{org_vec}->\texttt{info}[\textit{i}] is NULL (in C) or \texttt{LAPI\_ADDR\_NULL} (in FORTRAN), but the corresponding length (\textit{org_vec}->\texttt{len}[\textit{i}]) is not 0.
- **LAPI_ERR_ORG_VEC_LEN** Indicates that the total sum of all \textit{org_vec}->\texttt{len}[\textit{i}] (where \textit{[i]} is in the range 0 <= \textit{i} <= \textit{org_vec}->\texttt{num_vecs}) is greater than the value of LAPI constant \texttt{LAPI\_MAX\_MSG\_SZ}.
- **LAPI_ERR_ORG_VEC_NULL** Indicates that the \textit{org_vec} is NULL (in C) or \texttt{LAPI\_ADDR\_NULL} (in FORTRAN).
- **LAPI_ERR_ORG_VEC_TYPE** Indicates that the \textit{org_vec}->\texttt{vec_type} is not valid.
- **LAPI_ERR_STRIDE_ORG_VEC_ADDR_NULL** Indicates that the strided vector base address \textit{org_vec}->\texttt{info}[0] is NULL (in C) or \texttt{LAPI\_ADDR\_NULL} (in FORTRAN).
- **LAPI_ERR_STRIDE_TGT_VEC_ADDR_NULL** Indicates that the strided vector address \textit{tgt_vec}->\texttt{info}[0] is NULL (in C) or \texttt{LAPI\_ADDR\_NULL} (in FORTRAN).
- **LAPI_ERR_TGT** Indicates that the \textit{tgt} passed in is outside the range of tasks defined in the job.
- **LAPI_ERR_TGT_EXTENT** Indicates that \textit{tgt_vec}'s extent (stride \* \textit{num_vecs}) is greater than the value of LAPI constant \texttt{LAPI\_MAX\_MSG\_SZ}.
- **LAPI_ERR_TGT_PURGED** Indicates that the subroutine returned early because \texttt{LAPI\_Purge\_totask()} was called.
- **LAPI_ERR_TGT_STRIDE** Indicates that the \textit{tgt_vec}'s stride is less than its block size.
- **LAPI_ERR_TGT_VEC_ADDR** Indicates that the \textit{tgt_vec}->\texttt{info}[\textit{i}] is NULL (in C) or \texttt{LAPI\_ADDR\_NULL} (in FORTRAN), but its length (\textit{tgt_vec}->\texttt{len}[\textit{i}]) is not 0.
- **LAPI_ERR_TGT_VEC_LEN** Indicates that the sum of \textit{tgt_vec}->\texttt{len} is greater than the value of LAPI constant \texttt{LAPI\_MAX\_MSG\_SZ}.
- **LAPI_ERR_TGT_VEC_NULL** Indicates that \textit{tgt_vec} is NULL (in C) or \texttt{LAPI\_ADDR\_NULL} (in FORTRAN).
- **LAPI_ERR_TGT_VEC_TYPE** Indicates that the \textit{tgt_vec}->\texttt{vec_type} is not valid.
- **LAPI_ERR_VEC_LEN_DIFF** Indicates that \textit{org_vec} and \textit{tgt_vec} have different lengths (\texttt{len[]}).
- **LAPI_ERR_VEC_NUM_DIFF** Indicates that \textit{org_vec} and \textit{tgt_vec} have different \textit{num_vecs}.
**LAPI_ERR_VEC_TYPE_DIFF**

Indicates that `org_vec` and `tgt_vec` have different vector types (`vec_type`).

**Location**

`/usr/lib/liblapi_r.a`

**Related Information**

Subroutines: LAPI_Amsendv, LAPI_Getcntr, LAPI_Putv, LAPI_Qenv, LAPI_Waitcntr

---

**LAPI_Gfence Subroutine**

**Purpose**

Enforces order on LAPI calls across all tasks and provides barrier synchronization among them.

**Library**

Availability Library (`liblapi_r.a`)

**C Syntax**

```c
#include <lapi.h>

int LAPI_Gfence(hndl)
    lapi_handle_t hndl;
```

**FORTRAN Syntax**

```fortran
#include 'lapif.h'

LAPI_GFENCE(hndl, ierror)
   INTEGER hndl
   INTEGER ierror
```

**Description**

**Type of call:** collective data synchronization (blocking)

Use this subroutine to enforce global order on LAPI calls. This is a collective call. Collective calls must be made in the same order at all participating tasks.

On completion of this call, it is assumed that all LAPI communication associated with `hndl` from all tasks has quiesced. Although `hndl` is local, it represents a set of tasks that were associated with it at **LAPI_Init**, all of which must participate in this operation for it to complete. This is a data fence, which means that the data movement is complete. This is not an operation fence, which would need to include active message completion handlers completing on the target.

**Parameters**

**INPUT**

`hndl` Specifies the LAPI handle.

**OUTPUT**

`ierror` Specifies a FORTRAN return code. This is always the last parameter.
Return Values

LAPI_SUCCESS Indicates that the function call completed successfully.
LAPI_ERR_HNDL_INVALID Indicates that the \textit{hndl} passed in is not valid (not initialized or in terminated state).

Location

/\texttt{usr/lib/liblapi_r.a}

Related Information

Subroutines: LAPI_Fence

LAPI_Init Subroutine

Purpose

Initializes a LAPI context.

Library

Availability Library (\texttt{liblapi_r.a})

C Syntax

#include \texttt{<lapi.h>}

\begin{verbatim}
int LAPI_Init(hndl, lapi_info)
lapi_handle_t *hndl;
lapi_info_t *lapi_info;
\end{verbatim}

FORTRAN Syntax

include 'lapif.h'

\begin{verbatim}
LAPI_INIT(hndl, lapi_info, ierror)
INTEGER hndl
TYPE (LAPI_INFO_T) :: lapi_info
INTEGER ierror
\end{verbatim}

Description

Type of call: Local initialization

Use this subroutine to instantiate and initialize a new LAPI context. A handle to the newly-created LAPI context is returned in \texttt{hndl}. All subsequent LAPI calls can use \texttt{hndl} to specify the context of the LAPI operation. Except for \texttt{LAPI_Address()} and \texttt{LAPI_Msg_string()}, the user cannot make any LAPI calls before calling \texttt{LAPI_Init()}. The \texttt{lapi_info} structure (\texttt{lapi_info_t}) must be “zeroed out” before any fields are filled in. To do this in C, use this statement: \texttt{bzero (lapi_info, size of (lapi_info_t))}. In FORTRAN, you need to “zero out” each field manually in the \texttt{LAPI_INFO_T} type. Fields with a description of Future support should not be used because the names of those fields might change.

The \texttt{lapi_info_t} structure is defined as follows:

\begin{verbatim}
typedef struct {
lapi_dev_t protocol; /* Protocol device returned */
lapi-lib_t lib_vers; /* LAPI library version -- user-supplied */
uint epoch_num; /* No longer used */
} ...
\end{verbatim}
int num_compl_hdlr_thrd; /* Number of completion handler threads */
uint instance_no; /* Instance of LAPI to initialize [1-16] */
int info6; /* Future support */
LAPI_err_hdlr *err_hdlr; /* User-registered error handler */
com_thread_info_t *lapi_thread_attr; /* Support thread att and init function */
void *adapter_name; /* What adapter to initialize, i.e. css0, ml0 */
lapi_extend_t *add_info; /* Additional structure extension */
} lapi_info_t;

The fields are used as follows:

**protocol**
LAPI sets this field to the protocol that has been initialized.

**lib_vers**
Is used to indicate a library version to LAPI for compatibility purposes. Valid values for this field are:

- **L1_LIB** Provides basic functionality (this is the default).
- **L2_LIB** Provides the ability to use counters as structures.
- **LAST_LIB** Provides the most current level of functionality. For new users of LAPI, lib_vers should be set to **LAST_LIB**.

This field must be set to **L2_LIB** or **LAST_LIB** to use **LAPI_Nopoll_wait** and **LAPI_Setcntr_wstatus**.

**epoch_num**
This field is no longer used.

**num_compl_hdlr_thrd**
Indicates to LAPI the number of completion handler threads to initialize.

**instance_no**
Specifies the instance of LAPI to initialize (1 to 16).

**info6**
This field is for future use.

**err_hdlr**
Use this field to optionally pass a callback pointer to an error-handler routine.

**lapi_thread_attr**
Supports thread attributes and initialization function.

**adapter_name**
Is used in persistent subsystem (PSS) mode to pass an adapter name.

**add_info**
Is used for additional information in standalone UDP mode.

### Parameters

**INPUT/OUTPUT**

**lapi_info**
Specifies a structure that provides the parallel job information with which this LAPI context is associated. The value of this parameter cannot be NULL (in C) or **LAPI_ADDR_NULL** (in FORTRAN).

**OUTPUT**

**hndl**
Specifies a pointer to the LAPI handle to initialize.

**ierror**
Specifies a FORTRAN return code. This is always the last parameter.

### Return Values

**LAPI_SUCCESS**
Indicates that the function call completed successfully.

**LAPI_ERR_ALL_HNDL_IN_USE**
All available LAPI instances are in use.
LAPI_ERR_BOTH_NETSTR_SET
Both the MP_LAPI_NETWORK and MP_LAPI_INET statements are set (only one should be set).

LAPI_ERR_CSS_LOAD_FAILED
LAPI is unable to load the communication utility library.

LAPI_ERR_HNDL_INVALID
The lapi_handle_t * passed to LAPI for initialization is NULL (in C) or LAPI_ADDR_NULL (in FORTRAN).

LAPI_ERR_INFO_NONZERO_INFO
The future support fields in the lapi_info_t structure that was passed to LAPI are not set to zero (and should be).

LAPI_ERR_INFO_NULL
The lapi_info_t pointer passed to LAPI is NULL (in C) or LAPI_ADDR_NULL (in FORTRAN).

LAPI_ERR_MEMORY_EXHAUSTED
LAPI is unable to obtain memory from the system.

LAPI_ERR_MSG_API
Indicates that the MP_MSG_API environment variable is not set correctly.

LAPI_ERR_NO_NETSTR_SET
No network statement is set. Note that if running with POE, this will be returned if MP_MSG_API is not set correctly.

LAPI_ERR_NO_UDP_HNDLR
You passed a value of NULL (in C) or LAPI_ADDR_NULL (in FORTRAN) for both the UDP handler and the UDP list. One of these (the UDP handler or the UDP list) must be initialized for standalone UDP initialization. This error is returned in standalone UDP mode only.

LAPI_ERR_PSS_NON_ROOT
You tried to initialize the persistent subsystem (PSS) protocol as a non-root user.

LAPI_ERR_SHM_KE_NOT_LOADED
LAPI's shared memory kernel extension is not loaded.

LAPI_ERR_SHM_SETUP
LAPI is unable to set up shared memory. This error will be returned if LAPI_USE_SHM=only and tasks are assigned to more than one node.

LAPI_ERR_UDP_PKT_SZ
The UDP packet size you indicated is not valid.

LAPI_ERR_UNKNOWN
An internal error has occurred.

LAPI_ERR_USER_UDP_HNDLR_FAIL
The UDP handler you passed has returned a non-zero error code. This error is returned in standalone UDP mode only.

C Examples
The following environment variable must be set before LAPI is initialized:

MP_MSG_API=[ lapi | [ lapi,mpi | mpi,lapi ] | mpi_lapi ]

The following environment variables are also commonly used:

MP_EUILIB=[ ip | us ] (ip is the default)

MP_PROCS=number_of_tasks_in_job

LAPI_USE_SHM=[ yes | no | only ] (no is the default)

To initialize LAPI, follow these steps:
1. Set environment variables (as described in *RSCT for AIX 5L: LAPI Programming Guide*) before the user application is invoked. The remaining steps are done in the user application.
2. Clear `lapi_info_t`, then set any fields.
3. Call `LAPI_Init`.

**For systems running PE**

Both US and UDP/IP are supported for shared handles as long as they are the same for both handles. Mixed transport protocols such as LAPI IP and shared user space (US) are not supported.

To initialize a LAPI handle:

```c
{lapi_handle_t hndl;
lapi_info_t info;
bzero(&info, sizeof(lapi_info_t)); /* clear lapi_info */
LAPI_Init(&hndl, &info);
}
```

To initialize a LAPI handle and register an error handler:

```c
void my_err_hndlr(lapi_handle_t *hndl, int *error_code, lapi_err_t *err_type,
int *task_id, int *src)
{
/* examine passed parameters and delete desired information */
if (user wants to terminate ) {
   LAPI_Term(*hndl); /* will terminate LAPI */
   exit(some_return_code);
}
/* any additional processing */
return; /* signals to LAPI that error is non-fatal; execution should continue */
}
```

```c
{lapi_handle_t hndl;
lapi_info_t info;

bzero(&info, sizeof(lapi_info_t)); /* clear lapi_info */
/* set error handler pointer */
info.err_hndlr = (LAPI_err_hndlr) my_err_hndlr;
LAPI_Init(&hndl, &info);
}
```

**For standalone systems (not running PE)**

To initialize a LAPI handle for UDP/IP communication using a user handler:

```c
int my_udp_hndlr(lapi_handle_t *hndl, lapi_udp_t *local_addr, lapi_udp_t *addr_list,
                    lapi_udpinfo_t *info)
{
/* LAPI will allocate and free addr_list pointer when using */
/* a user handler */
```
/* use the AIX inet_addr call to convert an IP address */
/* from a dotted quad to a long */
task_0_ip_as_long = inet_addr(task_0_ip_as_string);
addr_list[0].ip_addr = task_0_ip_as_long;
addr_list[0].port_no = task_0_port_as_unsigned;

// Task 1

// Repeat for each task...

// Task N

task_num_tasks-1_ip_as_long = inet_addr(task_num_tasks-1_ip_as_string);
addr_list[num_tasks-1].ip_addr = task_num_tasks-1_ip_as_long;
addr_list[num_tasks-1].port_no = task_num_tasks-1_port_as_unsigned;

{ lapi_handle_t hndl;
  lapi_info_t info;
  lapi_extend_t extend_info;

  addr_list = malloc(num_tasks);

  // Populate addr_list
  // use the AIX inet_addr call to convert an IP address
  // from a dotted quad to a long
  task_0_ip_as_long = inet_addr(task_0_ip_as_string);
  addr_list[0].ip_addr = task_0_ip_as_long;
  addr_list[0].port_no = task_0_port_as_unsigned;

  // Task 1

  // Repeat for each task...

  // Task N

  task_num_tasks-1_ip_as_long = inet_addr(task_num_tasks-1_ip_as_string);
  addr_list[num_tasks-1].ip_addr = task_num_tasks-1_ip_as_long;
  addr_list[num_tasks-1].port_no = task_num_tasks-1_port_as_unsigned;

  LAPI_Init(&hndl, &info);
}

To initialize a LAPI handle for UDP/IP communication using a user list:

{ lapi_handle_t hndl;
  lapi_info_t info;
  lapi_extend_t extend_info;

  bzero(&info, sizeof(lapi_info_t)); /* clear lapi_info */
  bzero(&extend_info, sizeof(lapi_extend_t)); /* clear lapi_extend_info */

  extend_info.udp_hndlr = (udp_init_hndlr *) my_udp_hndlr;
  info.add_info = &extend_info;

  LAPI_Init(&hndl, &info);
}
task_num_tasks-1_ip_as_long = inet_addr(task_num_tasks-1_ip_as_string);
addr_list[num_tasks-1].ip_addr = task_num_tasks-1_ip_as_long;
addr_list[num_tasks-1].port_no = task_num_tasks-1_port_as_unsigned;

/* then assign to extend pointer */
extend_info.add_udp_addrs = addr_list;

info.add_info = &extend_info;

LAPI_Init(&hndl, &info);

/* user's responsibility only in the case of user list */
free(addr_list);
}

See the LAPI sample programs for complete examples of initialization in standalone mode.

To initialize a LAPI handle for user space (US) communication in standalone mode:

export MP_MSG_API=lapi
export MP_EUILIB=us
export MP_PROCS= /* number of tasks in job */
export MP_PARTITION= /* unique job key */
export MP_CHILD= /* unique task ID */
export MP_LAPI_NETWORK=01:164,sn0 /* LAPI network information */

run LAPI jobs as normal

See the README.LAPI.STANDALONE.US file in the standalone/us directory of the LAPI sample files for complete details.

Location
/usr/lib/liblapi_r.a

Related Information
Books: RSCT for AIX 5L: LAPI Programming Guide for information about
• Initializing LAPI on systems running PE
• Initializing LAPI on standalone systems
• Bulk message transfer

Subroutines: LAPI_Msg_string, LAPI_Term

LAPI_Msg_string Subroutine

Purpose
Retrieves the message that is associated with a subroutine return code.

Library
Availability Library (liblapi_r.a)
**C Syntax**

```c
#include <lapi.h>

LAPI_Msg_string(error_code, buf)
int   error_code;
void *buf;
```

**FORTRAN Syntax**

```fortran
include 'lapif.h'

LAPI_MSG_STRING(error_code, buf, ierror)
INTEGER error_code
CHARACTER buf(LAPI_MAX_ERR_STRING)
INTEGER ierror
```

**Description**

*Type of call: local queries*

Use this subroutine to retrieve the message string that is associated with a LAPI return code. LAPI tries to find the messages of any return codes that come from the AIX operating system or its communication subsystem.

**Parameters**

**INPUT**

- `error_code`: Specifies the return value of a previous LAPI call.

**OUTPUT**

- `buf`: Specifies the buffer to store the message string.
- `ierror`: Specifies a FORTRAN return code. This is always the last parameter.

**C Examples**

To get the message string associated with a LAPI return code:

```c
{
    char msg_buf[LAPI_MAX_ERR_STRING]; /* constant defined in lapi.h */
    int rc, errc;

    rc = some_LAPI_call();
    errc = LAPI_Msg_string(rc, msg_buf);
    /* msg_buf now contains the message string for the return code */
}
```

**Return Values**

- `LAPI_SUCCESS`: Indicates that the function call completed successfully.
- `LAPI_ERR_CATALOG_FAIL`: Indicates that the message catalog cannot be opened. An English-only string is copied into the user's message buffer (`buf`).
- `LAPI_ERR_CODE_UNKNOWN`: Indicates that `error_code` is outside of the range known to LAPI.
- `LAPI_ERR_RET_PTR_NULL`: Indicates that the value of the `buf` pointer is NULL (in C) or that the value of `buf` is `LAPI_ADDR_NULL` (in FORTRAN).
Location
/usr/lib/liblapi_r.a

Related Information
RSCT for AIX 5L: LAPI Programming Guide contains information about
- Initializing LAPI
- Bulk message transfer

Subroutines: LAPI_Msg_string, LAPI_Term

LAPI_Msgpoll Subroutine

Purpose
Allows the calling thread to check communication progress.

Library
Availability Library (liblapi_r.a)

C Syntax
#include <lapi.h>

int LAPI_Msgpoll(hndl, cnt, info)
lapi_handle_t hndl;
uint cnt;
lapi_msg_info_t *info;

typedef struct {
    lapi_msg_state_t status; /* Message status returned from LAPI_Msgpoll */
    ulong reserve[10]; /* Reserved */
} lapi_msg_info_t;

FORTRAN Syntax
include 'lapif.h'

LAPI_MSGPOLL(hndl, cnt, info, ierror)
INTEGER hndl
INTEGER cnt
TYPE (LAPI_MSG_STATE_T) :: info
INTEGER ierror

Description
Type of call: local progress monitor (blocking)

The LAPI_Msgpoll subroutine allows the calling thread to check communication progress. With this subroutine, LAPI provides a means of running the dispatcher several times until either progress is made or a specified maximum number of dispatcher loops have executed. Here, progress is defined as the completion of either a message send operation or a message receive operation.

LAPI_Msgpoll is intended to be used when interrupts are turned off. If the user has not explicitly turned interrupts off, LAPI temporarily disables interrupt mode while in this subroutine because the dispatcher is called, which will process any pending receive operations. If the LAPI dispatcher loops for the specified
maximum number of times, the call returns. If progress is made before the maximum count, the call will return immediately. In either case, LAPI will report status through a data structure that is passed by reference.

The `lapi_msg_info_t` structure contains a flags field (status), which is of type `lapi_msg_state_t`. Flags in the status field are set as follows:

- **LAPI_DISP_CNTR**: If the dispatcher has looped `cnt` times without making progress
- **LAPI_SEND_COMPLETE**: If a message send operation has completed
- **LAPI_RECV_COMPLETE**: If a message receive operation has completed
- **LAPI_BOTH_COMPLETE**: If both a message send operation and a message receive operation have completed
- **LAPI_POLLING_NET**: If another thread is already polling the network or shared memory completion

### Parameters

#### INPUT

- **hndl**: Specifies the LAPI handle.
- **cnt**: Specifies the maximum number of times the dispatcher should loop with no progress before returning.
- **info**: Specifies a status structure that contains the result of the `LAPI_Msgpoll()` call.

#### OUTPUT

- **ierror**: Specifies a FORTRAN return code. This is always the last parameter.

### C Examples

To loop through the dispatcher no more than 1000 times, then check what progress has been made:

```c
{
    lapi_msg_info_t msg_info;
    int cnt = 1000;
    
    LAPI_Msgpoll(hndl, cnt, &msg_info);

    if (msg_info.status & LAPI_BOTH_COMPLETE) {
        /* both a message receive and a message send have been completed */
    } else if (msg_info.status & LAPI_RECV_COMPLETE) {
        /* just a message receive has been completed */
    } else if (msg_info.status & LAPI_SEND_COMPLETE) {
        /* just a message send has been completed */
    } else {
        /* cnt loops and no progress */
    }
}
```

### Return Values

- **LAPI_SUCCESS**: Indicates that the function call completed successfully.
- **LAPI_ERR_HNDL_INVALID**: Indicates that the `hndl` passed in is not valid (not initialized or in terminated state).
LAPI_ERR_MSG_INFO_NULL
Indicates that the info pointer is NULL (in C) or that the value of info is LAPI_ADDR_NULL (in FORTRAN).

Location
/usr/lib/liblapi_r.a

Related Information
Subroutines: LAPI_Getcntr, LAPI_Probe, LAPI_Setcntr, LAPI_Waitcntr

LAPI_Nopoll_wait Subroutine

Purpose
Waits for a counter update without polling.

Library
Availability Library (liblapi_r.a)

C Syntax
#include <lapi.h>

void LAPI_Nopoll_wait(hndl, cntr_ptr, val, cur_cntr_val)
    lapi_handle_t hndl;
    lapi_cntr_t *cntr_ptr;
    int val;
    int *cur_cntr_val;

FORTRAN Syntax
include 'lapif.h'

int LAPI_NOPOLL_WAIT(hndl, cntr, val, cur_cntr_val, ierr)
    INTEGER hndl
    TYPE (LAPI_CNTR_T) :: cntr
    INTEGER val
    INTEGER cur_cntr_val
    INTEGER ierr

Description
Type of call: recovery (blocking)

This subroutine waits for a counter update without polling (that is, without explicitly invoking LAPI’s internal communication dispatcher). This call may or may not check for message arrivals over the LAPI context hndl. The cur_cntr_val variable is set to the current counter value. Although it has higher latency than LAPI_Waitcntr, LAPI_Nopoll_wait frees up the processor for other uses.

Note: To use this subroutine, the lib_vers field in the lapi_info_t structure must be set to L2_LIB or LAST_LIB.

Parameters
INPUT
    hndl Specifies the LAPI handle.
val Specifies the relative counter value (starting from 1) that the counter needs to reach before returning.

cur_cntr_val Specifies the integer value of the current counter. The value of this parameter can be NULL (in C) or LAPI_ADDR_NULL (in FORTRAN).

INPUT/OUTPUT

cntr_ptr Points to the lapi_cntr_t structure in C.

cntr Is the lapi_cntr_t structure in FORTRAN.

OUTPUT

ierror Specifies a FORTRAN return code. This is always the last parameter.

Return Values

LAPI_SUCCESS Indicates that the function call completed successfully.

LAPI_ERR_CNTR_NULL Indicates that the cntr_ptr pointer is NULL (in C) or that the value of cntr is LAPI_ADDR_NULL (in FORTRAN).

LAPI_ERR_CNTR_VAL Indicates that the val passed in is less than or equal to 0.

LAPI_ERR_HNDL_INVALID Indicates that the hndl passed in is not valid (not initialized or in terminated state).

LAPI_ERR_MULTIPLE_WAITERS Indicates that more than one thread is waiting for the counter.

LAPI_ERR_TGT_PURGED Indicates that the subroutine returned early because LAPI_Purge_totask() was called.

Restrictions

Use of this subroutine is not recommended on a system that is running Parallel Environment (PE).

Location

/usr/lib/liblapi_r.a

Related Information

Subroutines: LAPI_Init, LAPI_Purge_totask, LAPI_Resume_totask, LAPI_Setcntr_wstatus

LAPI_Probe Subroutine

Purpose

Transfers control to the communication subsystem to check for arriving messages and to make progress in polling mode.

Library

Availability Library (liblapi_r.a)

C Syntax

#include <lapi.h>

int LAPI_Probe(hndl)
    lapi_handle_t hndl;
FORTRAN Syntax

```
include 'lapif.h'

int LAPI_PROBE(hndl, ierror)
INTEGER hndl
INTEGER ierror
```

Description

**Type of call:** local progress monitor (non-blocking)

This subroutine transfers control to the communication subsystem in order to make progress on messages associated with the context `hndl`. A **LAPI_Probe** operation lasts for one round of the communication dispatcher.

**Note:** There is no guarantee about receipt of messages on the return from this function.

Parameters

**INPUT**

`hndl` Specifies the LAPI handle.

**OUTPUT**

`ierror` Specifies a FORTRAN return code. This is always the last parameter.

Return Values

- **LAPI_SUCCESS** Indicates that the function call completed successfully.
- **LAPI_ERR_HNDL_INVALID** Indicates that the `hndl` passed in is not valid (not initialized or in terminated state).

Location

/usr/lib/liblapi_r.a

Related Information

Subroutines: LAPI_Getcntr, LAPI_Msgpoll, LAPI_Nopoll_wait, LAPI_Waitcntr

---

**LAPI_Purge_totask Subroutine**

**Purpose**

Allows a task to cancel messages to a given destination.

**Library**

Availability Library (liblapi_r.a)

**C Syntax**

```
#include <lapi.h>

int LAPI_Purge_totask(hndl, dest)
  lapi_handle_t hndl;
  uint dest;
```
FORTRAN Syntax

#include 'lapif.h'

int LAPI_PURGE_TOTASK(hndl, dest, ierror)
INTEGER hndl
INTEGER dest
INTEGER ierror

Description

Type of call: recovery

This subroutine cancels messages and resets the state corresponding to messages in flight or submitted to be sent to a particular target task. This is an entirely local operation. For correct behavior a similar invocation is expected on the destination (if it exists). This function cleans up all the state associated with pending messages to the indicated target task. It is assumed that before the indicated task starts communicating with this task again, it also purges this instance (or that it was terminated and initialized again). It will also wake up all threads that are in LAPI_Nopoll_wait depending on how the arguments are passed to the LAPI_Nopoll_wait function. The behavior of LAPI_Purge_totask is undefined if LAPI collective functions are used.

Note: This subroutine should not be used when the parallel application is running in a PE/LoadLeveler environment.

LAPI_Purge_totask is normally used after connectivity has been lost between two tasks. If connectivity is restored, the tasks can restored for LAPI communication by calling LAPI_Resume_totask.

Parameters

INPUT

hndl Specifies the LAPI handle.
dest Specifies the destination instance ID to which pending messages need to be cancelled.

OUTPUT

ierror Specifies a FORTRAN return code. This is always the last parameter.

Restrictions

Use of this subroutine is not recommended on a system that is running Parallel Environment (PE).

Return Values

LAPI_SUCCESS Indicates that the function call completed successfully.
LAPI_ERR_HNDL_INVALID Indicates that the hndl passed in is not valid (not initialized or in terminated state).
LAPI_ERR_TGT Indicates that dest is outside the range of tasks defined in the job.

Location

/usr/lib/liblapi_r.a

Related Information

Subroutines: LAPI_Init, LAPI_Nopoll_wait, LAPI_Resume_totask, LAPI_Term
LAPI_Put Subroutine

Purpose
Transfers data from a local task to a remote task.

Library
Availability Library (liblapi_r.a)

C Syntax
#include <lapi.h>

int LAPI_Put(hndl, tgt, len, tgt_addr, org_addr, tgt_cntr, org_cntr, cmpl_cntr)
  lapi_handle_t hndl;
  int tgt;
  ulong len;
  void *tgt_addr;
  void *org_addr;
  lapi_cntr_t *tgt_cntr;
  lapi_cntr_t *org_cntr;
  lapi_cntr_t *cmpl_cntr;

FORTRAN Syntax
include 'lapif.h'

int LAPI_PUT(hndl, tgt, len, tgt_addr, org_addr, tgt_cntr, org_cntr, ierror)
  INTEGER hndl
  INTEGER tgt
  INTEGER (KIND=LAPI_LONG_TYPE) :: len
  INTEGER (KIND=LAPI_ADDR_TYPE) :: tgt_addr
  INTEGER org_addr
  INTEGER (KIND=LAPI_ADDR_TYPE) :: tgt_cntr
  TYPE (LAPI_CNTR_T) :: org_cntr
  TYPE (LAPI_CNTR_T) :: cmpl_cntr
  INTEGER ierror

Description
Type of call: point-to-point communication (non-blocking)

Use this subroutine to transfer data from a local (origin) task to a remote (target) task. The origin counter will increment on the origin task upon origin buffer availability. The target counter will increment on the target and the completion counter will increment at the origin task upon message completion. Because there is no completion handler, message completion and target buffer availability are the same in this case.

This is a non-blocking call. The caller cannot assume that the data transfer has completed upon the return of the function. Instead, counters should be used to ensure correct buffer accesses as defined above.

Note that a zero-byte message does not transfer data, but it does have the same semantic with respect to counters as that of any other message.

Parameters

INPUT

hndl  Specifies the LAPI handle.
tgt specifies the task ID of the target task. The value of this parameter must be in the range 0 <= tgt < NUM_TASKS.

len specifies the number of bytes to be transferred. This parameter must be in the range 0 <= len <= the value of LAPI constant LAPI_MAX_MSG_SZ.

tgt_addr specifies the address on the target task where data is to be copied into. If len is 0, the value of this parameter can be NULL (in C) or LAPI_ADDR_NULL (in FORTRAN).

org_addr specifies the address on the origin task from which data is to be copied. If len is 0, the value of this parameter can be NULL (in C) or LAPI_ADDR_NULL (in FORTRAN).

**Input/Output**

tgt_cntr specifies the target counter address. The target counter is incremented upon message completion. If this parameter is NULL (in C) or LAPI_ADDR_NULL (in FORTRAN), the target counter is not updated.

org_cntr specifies the origin counter address (in C) or the origin counter (in FORTRAN). The origin counter is incremented at buffer availability. If this parameter is NULL (in C) or LAPI_ADDR_NULL (in FORTRAN), the origin counter is not updated.

cmpl_cntr specifies the completion counter address (in C) or the completion counter (in FORTRAN) that is a reflection of tgt_cntr. The completion counter is incremented at the origin after tgt_cntr is incremented. If this parameter is NULL (in C) or LAPI_ADDR_NULL (in FORTRAN), the completion counter is not updated.

**Output**

ierror specifies a FORTRAN return code. This is always the last parameter.

**Examples**

```c
  /* initialize the table buffer for the data addresses */
  /* get remote data buffer addresses */
  LAPI_Address_init(hndl,(void *)data_buffer,data_buffer_list);
  .
  .
  LAPI_Put(hndl, tgt, (ulong) data_len, (void *)(data_buffer_list[tgt]),
            (void *) data_buffer, tgt_cntr, org_cntr, compl_cntr);

  /* transfer data_len bytes from local address data_buffer. */
  /* write the data starting at address data_buffer_list[tgt] on */
  /* task tgt, tgt_cntr, org_cntr, and compl_cntr can be NULL. */
```

**Return Values**

**LAPI_SUCCESS** indicates that the function call completed successfully.

**LAPI_ERR_DATA_LEN** indicates that the value of len is greater than the value of LAPI constant LAPI_MAX_MSG_SZ.

**LAPI_ERR_HNDL_INVALID** indicates that the hndl passed in is not valid (not initialized or in terminated state).

**LAPI_ERR_ORG_ADDR_NULL** indicates that the org_addr parameter passed in is NULL (in C) or LAPI_ADDR_NULL (in FORTRAN), but len is greater than 0.
LAPI_ERR_TGT Indicates that the tgt passed in is outside the range of tasks defined in the job.

LAPI_ERR_TGT_ADDR_NULL Indicates that the tgt_addr parameter passed in is NULL (in C) or LAPI_ADDR_NULL (in FORTRAN), but len is greater than 0.

LAPI_ERR_TGT_PURGED Indicates that the subroutine returned early because LAPI_Purge_totask() was called.

Location
/usr/lib/liblapi_r.a

Related Information
Subroutines: LAPI_Get, LAPI_Getcntr, LAPI_Qenv, LAPI_Setcntr, LAPI_Waitcntr, LAPI_Xfer

LAPI_Putv Subroutine

Purpose
Transfers vectors of data from a local task to a remote task.

Library
Availability Library (liblapi_r.a)

C Syntax
#include <lapi.h>

int LAPI_Putv(hndl, tgt, tgt_vec, org_vec, tgt_cntr, org_cntr, cmpl_cntr)

typedef struct {
    lapi_vectype_t vec_type; /* operation code */
    uint num_vecs; /* number of vectors */
    void **info; /* vector of information */
    ulong *len; /* vector of lengths */
} lapi_vec_t;

FORTRAN Syntax
include 'lapif.h'

LAPI_PUTV(hndl, tgt, tgt_vec, org_vec, tgt_cntr, org_cntr, cmpl_cntr, ierror)
INTEGER hndl
INTEGER tgt
INTEGER (KIND=LAPI_ADDR_TYPE) :: tgt_vec
TYPE (LAPI_VEC_T) :: org_vec
INTEGER (KIND=LAPI_ADDR_TYPE) :: tgt_cntr
TYPE (LAPI_CNTR_T) :: org_cntr
TYPE (LAPI_CNTR_T) :: cmpl_cntr
INTEGER ierror
The 32-bit version of the LAPI_VEC_T type is defined as:

```fortran
TYPE LAPI_VEC_T
  SEQUENCE
    INTEGER(KIND = 4) :: vec_type
    INTEGER(KIND = 4) :: num_vecs
    INTEGER(KIND = 4) :: info
    INTEGER(KIND = 4) :: len
END TYPE LAPI_VEC_T
```

The 64-bit version of the LAPI_VEC_T type is defined as:

```fortran
TYPE LAPI_VEC_T
  SEQUENCE
    INTEGER(KIND = 4) :: vec_type
    INTEGER(KIND = 4) :: num_vecs
    INTEGER(KIND = 8) :: info
    INTEGER(KIND = 8) :: len
END TYPE LAPI_VEC_T
```

**Description**

**Type of call:** point-to-point communication (non-blocking)

**LAPI_Putv** is the vector version of the **LAPI_Put** call. Use this subroutine to transfer vectors of data from the origin task to the target task. The origin vector descriptions and the target vector descriptions are located in the address space of the origin task. However, the values specified in the info array of the target vector must be addresses in the address space of the target task.

The calling program cannot assume that the origin buffer can be changed or that the contents of the target buffers on the target task are ready for use upon function return. After the origin counter (org_cntr) is incremented, the origin buffers can be modified by the origin task. After the target counter (tgt_cntr) is incremented, the target buffers can be modified by the target task. If you provide a completion counter (cmpc_cntr), it is incremented at the origin after the target counter (tgt_cntr) has been incremented at the target. If the values of any of the counters or counter addresses are NULL (in C) or **LAPI_ADDR_NULL** (in FORTRAN), the data transfer occurs, but the corresponding counter increments do not occur.

If a strided vector is being transferred, the size of each block must not be greater than the stride size in bytes.

The length of any vector pointed to by org_vec must be equal to the length of the corresponding vector pointed to by tgt_vec.

LAPI does not check for any overlapping regions among vectors either at the origin or the target. If the overlapping regions exist on the target side, the contents of the target buffer are undefined after the operation.

See **LAPI_Amsendv** for more information about using the various vector types. (**LAPI_Putv** does not support the LAPI_GEN_GENERIC type.)

**Parameters**

**INPUT**

- **hndl** Specifies the LAPI handle.
- **tgt** Specifies the task ID of the target task. The value of this parameter must be in the range 0 <= tgt < **NUM_TASKS**.
- **tgt_vec** Points to the target vector description.
- **org_vec** Points to the origin vector description.
INPUT/OUTPUT

tgt_cntr  Specifies the target counter address. The target counter is incremented upon message completion. If this parameter is NULL (in C) or LAPI_ADDR_NULL (in FORTRAN), the target counter is not updated.

org_cntr  Specifies the origin counter address (in C) or the origin counter (in FORTRAN). The origin counter is incremented at buffer availability. If this parameter is NULL (in C) or LAPI_ADDR_NULL (in FORTRAN), the origin counter is not updated.

cmpl_cntr  Specifies the completion counter address (in C) or the completion counter (in FORTRAN) that is a reflection of tgt_cntr. The completion counter is incremented at the origin after tgt_cntr is incremented. If this parameter is NULL (in C) or LAPI_ADDR_NULL (in FORTRAN), the completion counter is not updated.

OUTPUT

ierror  Specifies a FORTRAN return code. This is always the last parameter.

C Examples
To put a LAPI_GEN_IOVECTOR:

```c
{ /* retrieve a remote data buffer address for data to transfer, */ /* such as through LAPI_Address_init */
    /* task that calls LAPI_Putv sets up both org_vec and tgt_vec */
    org_vec->num_vecs = NUM_VECS;
    org_vec->vec_type = LAPI_GEN_IOVECTOR;
    org_vec->len = (unsigned long *) malloc(NUM_VECS*sizeof(unsigned long));
    org_vec->info = (void **) malloc(NUM_VECS*sizeof(void *));
    /* each org_vec->info[i] gets a base address on the origin task */
    /* each org_vec->len[i] gets the number of bytes to transfer */
    /* from org_vec->info[i] */
    tgt_vec->num_vecs = NUM_VECS;
    tgt_vec->vec_type = LAPI_GEN_IOVECTOR;
    tgt_vec->len = (unsigned long *) malloc(NUM_VECS*sizeof(unsigned long));
    tgt_vec->info = (void **) malloc(NUM_VECS*sizeof(void *));
    /* each tgt_vec->info[i] gets a base address on the target task */
    /* each tgt_vec->len[i] gets the number of bytes to write to vec->info[i] */
    /* For LAPI_GEN_IOVECTOR, num_vecs, vec_type, and len must be the same */
    LAPI_Putv(hndl, tgt, tgt_vec, org_vec, tgt_cntr, org_cntr, compl_cntr);
    /* tgt_cntr, org_cntr and compl_cntr can all be NULL */
    /* data will be transferred as follows: */
    /* org_vec->len[0] bytes will be retrieved from */
    /* org_vec->info[0] and written to tgt_vec->info[0] */
    /* org_vec->len[1] bytes will be retrieved from */
    /* org_vec->info[1] and written to tgt_vec->info[1] */
    .
    .
    /* org_vec->len[NUM_VECS-1] bytes will be retrieved */
    /* from org_vec->info[NUM_VECS-1] and written to */
    /* tgt_vec->info[NUM_VECS-1] */
}
```

See the example in LAPI_Amsendv for information on other vector types.
Return Values

**LAPI_SUCCESS**
Indicates that the function call completed successfully.

**LAPI_ERR_HNDL_INVALID**
Indicates that the `hndl` passed in is not valid (not initialized or in terminated state).

**LAPI_ERR_ORG_EXTENT**
Indicates that the `org_vec`'s extent (stride * num_vecs) is greater than the value of LAPI constant `LAPI_MAX_MSG_SZ`.

**LAPI_ERR_ORG_STRIDE**
Indicates that the `org_vec` stride is less than block.

**LAPI_ERR_ORG_VEC_ADDR**
Indicates that the `org_vec->info[i]` is NULL (in C) or `LAPI_ADDR_NULL` (in FORTRAN), but its length (`org_vec->len[i]`) is not 0.

**LAPI_ERR_ORG_VEC_LEN**
Indicates that the sum of `org_vec->len` is greater than the value of LAPI constant `LAPI_MAX_MSG_SZ`.

**LAPI_ERR_ORG_VEC_NULL**
Indicates that the `org_vec` is NULL (in C) or `LAPI_ADDR_NULL` (in FORTRAN).

**LAPI_ERR_ORG_VEC_TYPE**
Indicates that the `org_vec->vec_type` is not valid.

**LAPI_ERR_STRIDE_ORG_VEC_ADDR_NULL**
Indicates that the strided vector address `org_vec->info[0]` is NULL (in C) or `LAPI_ADDR_NULL` (in FORTRAN).

**LAPI_ERR_STRIDE_TGT_VEC_ADDR_NULL**
Indicates that the strided vector address `tgt_vec->info[0]` is NULL (in C) or `LAPI_ADDR_NULL` (in FORTRAN).

**LAPI_ERR_TGT**
Indicates that the `tgt` passed in is outside the range of tasks defined in the job.

**LAPI_ERR_TGT_EXTENT**
Indicates that `tgt_vec`'s extent (stride * num_vecs) is greater than the value of LAPI constant `LAPI_MAX_MSG_SZ`.

**LAPI_ERR_TGT_PURGED**
Indicates that the subroutine returned early because `LAPI_Purge_totask()` was called.

**LAPI_ERR_TGT_STRIDE**
Indicates that the `tgt_vec` stride is less than block.

**LAPI_ERR_TGT_VEC_ADDR**
Indicates that the `tgt_vec->info[i]` is NULL (in C) or `LAPI_ADDR_NULL` (in FORTRAN), but its length (`tgt_vec->len[i]`) is not 0.

**LAPI_ERR_TGT_VEC_LEN**
Indicates that the sum of `tgt_vec->len` is greater than the value of LAPI constant `LAPI_MAX_MSG_SZ`.

**LAPI_ERR_TGT_VEC_NULL**
Indicates that `tgt_vec` is NULL (in C) or `LAPI_ADDR_NULL` (in FORTRAN).

**LAPI_ERR_TGT_VEC_TYPE**
Indicates that the `tgt_vec->vec_type` is not valid.

**LAPI_ERR_VEC_LEN_DIFF**
Indicates that `org_vec` and `tgt_vec` have different lengths (`len[i]`).

**LAPI_ERR_VEC_NUM_DIFF**
Indicates that `org_vec` and `tgt_vec` have different `num_vecs`.

**LAPI_ERR_VEC_TYPE_DIFF**
Indicates that `org_vec` and `tgt_vec` have different vector types (`vec_type`).

Location

/user/lib/liblapi_r.a
Related Information
Subroutines: LAPI_Amsendv, LAPI_Getcntr, LAPI_Getv, LAPI_Qenv, LAPI_Setcntr, LAPI_Waitcntr, LAPI_Xfer

LAPI_Qenv Subroutine

Purpose
Used to query LAPI for runtime task information.

Library
Availability Library (liblapi_r.a)

C Syntax
#include <lapif.h>

int LAPI_Qenv(hndl, query, ret_val)
lapi_handle_t hndl;
int *ret_val; /* ret_val's type varies (see Additional query types) */

FORTRAN Syntax
include 'lapif.h'

LAPI_QENV(hndl, query, ret_val, ierror)
INTEGER hndl
INTEGER query
INTEGER ret_val /* ret_val's type varies (see Additional query types) */
INTEGER ierror

Description
Type of call: local queries

Use this subroutine to query runtime settings and statistics from LAPI. LAPI defines a set of query types as an enumeration in lapif.h for C and explicitly in the 32-bit and 64-bit versions of lapif.h for FORTRAN.

For example, you can query the size of the table that LAPI uses for the LAPI_Addr_set subroutine using a query value of LOC_ADDRTBL_SZ:

LAPI_Qenv(hndl, LOC_ADDRTBL_SZ, &ret_val);

ret_val will contain the upper bound on the table index. A subsequent call to LAPI_Addr_set (hndl, addr, addr_hndl); could then ensure that the value of addr_hndl is between 0 and ret_val.

When used to show the size of a parameter, a comparison of values, or a range of values, valid values for the query parameter of the LAPI_Qenv subroutine appear in SMALL, BOLD capital letters. For example:

NUM_TASKS

is a shorthand notation for:

LAPI_Qenv(hndl, NUM_TASKS, ret_val)
In C, `lapi_query_t` defines the valid types of LAPI queries:

```c
typedef enum {
    TASK_ID=0,    /* Query the task ID of the current task in the job */
    NUM_TASKS,   /* Query the number of tasks in the job */
    MAX_UHDR_SZ, /* Query the maximum user header size for active messaging */
    MAX_DATA_SZ, /* Query the maximum data size that can be sent */
    ERROR_CHK,   /* Query and set parameter checking on (1) or off (0) */
    TIMEOUT,     /* Query and set the current communication timeout setting */
    INTERUPT_SET,/* Query and set interrupt mode on (1) or off (0) */
    MAX_PORTS,   /* Query the maximum number of available communication ports */
    MAX_PKTS_SZ, /* This is the payload size of 1 packet */
    NUM_REX_BUFS,/* Number of retransmission buffers */
    REX_BUF_SZ,  /* Size of each retransmission buffer in bytes */
    LOC_ADDR_TBL_SZ,/* Size of address store table used by LAPI_Addr_set */
    EPOCH_NUM,   /* No longer used by LAPI (supports legacy code) */
    USE_THRESH,  /* No longer used by LAPI (supports legacy code) */
    RCV_FIFO_SIZE,/* No longer used by LAPI (supports legacy code) */
    ACK_THRESHOLD,/* Query and set the threshold of acknowledgments going */
    LAST_QUERY  /* back to the source */
} lapi_query_t;
```

In FORTRAN, the valid types of LAPI queries are defined in `lapif.h` as follows:

```fortran
integer TASK_ID,NUM_TASKS,MAX_UHDR_SZ,MAX_DATA_SZ,ERROR_CHK
integer TIMEOUT,MIN_TIMEOUT,MX_MAX_TIMEOUT
integer INTERRUPT_SET,MX_MAX_PORTS,MX_PKT_SZ,MX_PKT_SZ,NUM_REX_BUFS
integer REX_BUF_SZ,LOC_ADDR_TBL_SZ,EPOCH_NUM,USE_THRESH
integer RCV_FIFO_SIZE,MX_ATOM_SIZE,BUF_CP_SIZE
integer MAX_PKTS_OUT,ACK_THRESHOLD,QUERY_SHM_ENABLED
```
Integer QUERY_SHM_NUM_TASKS, QUERY_SHM_TASKS
Integer QUERY_STATISTICS, PRINT_STATISTICS
Integer QUERY_SHM_STATISTICS, QUERY_LOCAL_SEND_STATISTICS
Integer BULK_XFER, BULK_MIN_MSG_SIZE,
Integer LAST_QUERY
Parameter (TASK_ID=0, NUM_TASKS=1, MAX_UHDR_SZ=2, MAX_DATA_SZ=3)
Parameter (ERROR_CHK=4, TIMEOUT=5, MIN_TIMEOUT=6)
Parameter (MAX_TIMEOUT=7, INTERRUPT_SET=8, MAX_PORTS=9)
Parameter (MAX_PKT_SZ=10, NUM_REX_BUFS=11, REX_BUF_SZ=12)
Parameter (LOC_ADDRTBL_SZ=13, EPOCH_NUM=14, USE_THRESH=15)
Parameter (RCV_FIFO_SIZE=16, MAX_ATOM_SIZE=17, BUF_CP_SIZE=18)
Parameter (MAX_PKTS_OUT=19, ACK_THRESHOLD=20)
Parameter (QUERY_SHM_ENABLED=21, QUERY_SHM_NUM_TASKS=22)
Parameter (QUERY_SHM_TASKS=23, QUERY_STATISTICS=24)
Parameter (PRINT_STATISTICS=25)
Parameter (QUERY_SHM_STATISTICS=26, QUERY_LOCAL_SEND_STATISTICS=27)
Parameter (BULK_XFER=28, BULK_MIN_MSG_SIZE=29)
Parameter (LAST_QUERY=30)

Additional query types
LAPI provides additional query types for which the behavior of LAPI_Qenv is slightly different:

PRINT_STATISTICS When passed this query type, LAPI sends data transfer statistics to standard output. In this case, ret_val is unaffected. However, LAPI's error checking requires that the value of ret_val is not NULL (in C) or LAPI_ADDR_NULL (in FORTRAN) for all LAPI_Qenv types (including PRINT_STATISTICS).

QUERY_LOCAL_SEND_STATISTICS When passed this query type, LAPI_Qenv interprets ret_val as a pointer to type lapi_statistics_t. Upon function return, the fields of the structure contain LAPI's data transfer statistics for data transferred through intra-task local copy. The packet count will be 0.

QUERY_SHM_STATISTICS When passed this query type, LAPI_Qenv interprets ret_val as a pointer to type lapi_statistics_t. Upon function return, the fields of the structure contain LAPI's data transfer statistics for data transferred through shared memory.

QUERY_SHM_TASKS When passed this query type, LAPI_Qenv returns a list of task IDs with which this task can communicate using shared memory. ret_val must be an int * with enough space to hold NUM_TASKS integers. For each task i, if it is possible to use shared memory, ret_val[i] will contain the shared memory task ID. If it is not possible to use shared memory, ret_val[i] will contain -1.

QUERY_STATISTICS When passed this query type, LAPI_Qenv interprets ret_val as a pointer to type lapi_statistics_t. Upon function return, the fields of the structure contain LAPI's data transfer statistics for data transferred using the user space (US) protocol or UDP/IP.

Parameters

Input

hndl Specifies the LAPI handle.
query Specifies the type of query you want to request. In C, the values for query are defined by the lapi_query_t enumeration in lapi.h. In FORTRAN, these values are defined explicitly in the 32-bit version and the 64-bit version of lapif.h.

Output


ret_val Specifies the reference parameter for LAPI to store as the result of the query. The value of this parameter cannot be NULL (in C) or \texttt{LAPI_ADDR_NULL} (in FORTRAN).

ierror Specifies a FORTRAN return code. This is always the last parameter.

Return values

\texttt{LAPI\_SUCCESS} Indicates that the function call completed successfully.

\texttt{LAPI\_ERR\_HNDL\_INVALID} Indicates that the \texttt{hndl} passed in is not valid (not initialized or in terminated state).

\texttt{LAPI\_ERR\_QUERY\_TYPE} Indicates that the query passed in is not valid.

\texttt{LAPI\_ERR\_RET\_PTR\_NULL} Indicates that the value of the \texttt{ret\_val} pointer is NULL (in C) or that the value of \texttt{ret\_val} is \texttt{LAPI\_ADDR\_NULL} (in FORTRAN).

C Examples

To query runtime values from LAPI:

\begin{verbatim}
{  int task_id;
   lapi_statistics_t stats;
   ...
   LAPI_Qenv(hndl, TASK_ID, &task_id);
   /* task_id now contains the task ID */
   ...
   LAPI_Qenv(hndl, QUERY_STATISTICS, (int *)&stats);
   /* the fields of the stats structure are now filled in with runtime values */
   ...
}
\end{verbatim}

Location

\texttt{/usr/lib/liblapi_r.a}

Related Information

Subroutines: \texttt{LAPI\_Amsend, LAPI\_Get, LAPI\_Put, LAPI\_Senv, LAPI\_Xfer}

---

\textbf{LAPI\_Resume\_totask} Subroutine

Purpose

Re-enables the sending of messages to the task.

Library

Availability Library (\texttt{liblapi_r.a})

C Syntax

\begin{verbatim}
#include <lapi.h>

int LAPI\_Resume\_totask(hndl, dest)
   lapi_handle_t hndl;
   uint dest;
\end{verbatim}
FORTRAN Syntax

```fortran
include 'lapif.h'

int LAPI_RESUME_TOTASK(hndl, dest, ierror)
INTEGER hndl
INTEGER dest
INTEGER ierror
```

Description

**Type of call:** recovery

This subroutine is used in conjunction with **LAPI_Purge_totask**. It enables LAPI communication to be reestablished for a task that had previously been purged. The purged task must either restart LAPI or execute a **LAPI_Purge_totask/LAPI_Resume_totask** sequence for this task.

Parameters

**INPUT**

- **hndl** Specifies the LAPI handle.
- **dest** Specifies the destination instance ID with which to resume communication.

**OUTPUT**

- **ierror** Specifies a FORTRAN return code. This is always the last parameter.

Restrictions

Use of this subroutine is *not* recommended on a system that is running Parallel Environment (PE).

Return Values

- **LAPI_SUCCESS** Indicates that the function call completed successfully.
- **LAPI_ERR_HNDL_INVALID** Indicates that the *hndl* passed in is not valid (not initialized or in terminated state).
- **LAPI_ERR_TGT** Indicates that the *tgt* passed in is outside the range of tasks defined in the job.

Location

/usr/lib/liblapi_r.a

Related Information

Subroutines: **LAPI_Init, LAPI_Nopoll_wait, LAPI_Purge_totask, LAPI_Term**

**LAPI_Rmw Subroutine**

**Purpose**

Provides data synchronization primitives.

**Library**

Availability Library (liblapl_r.a)
C Syntax

```c
#include <lapi.h>

int LAPI_Rmw(hndl, op, tgt, tgt_var, in_val, prev_tgt_val, org_cntr)
```

```c
lapi_handle_t hndl;
RMW_ops_t op;
uint tgt;
int *tgt_var;
int *in_val;
int *prev_tgt_val;
lapi_cntr_t *org_cntr;
```

FORTRAN Syntax

```fortran
include 'lapif.h'
```

```fortran
LAPI_RMW(hndl, op, tgt, tgt_var, in_val, prev_tgt_val, org_cntr, ierr)
```

```fortran
INTEGER hndl
INTEGER op
INTEGER tgt
INTEGER (KIND=LAPI_ADDR_TYPE) :: tgt_var
INTEGER in_val
INTEGER prev_tgt_val
TYPE (LAPI_CNTR_T) :: org_cntr
INTEGER ierr
```

Description

**Type of call:** point-to-point communication (non-blocking)

Use this subroutine to synchronize two independent pieces of data, such as two tasks sharing a common data structure. The operation is performed at the target task (tgt) and is atomic. The operation takes an input value (in_val) from the origin and performs one of four operations (op) on a variable (tgt_var) at the target (tgt), and then replaces the target variable (tgt_var) with the results of the operation (op). The original value (prev_tgt_val) of the target variable (tgt_var) is returned to the origin.

The operations (op) are performed over the context referred to by hndl. The outcome of the execution of these calls is as if the following code was executed atomically:

```
*prev_tgt_val = *tgt_var;
*tgt_var = f(*tgt_var, *in_val);
```

where:

- \( f(a, b) = a + b \) for **FETCH_AND_ADD**
- \( f(a, b) = a | b \) for **FETCH_AND_OR** (bitwise or)
- \( f(a, b) = b \) for **SWAP**

For **COMPARE_AND_SWAP**, in_val is treated as a pointer to an array of two integers, and the op is the following atomic operation:

```
if(*tgt_var == in_val[0]) {
  *prev_tgt_val = TRUE;
  *tgt_var = in_val[1];
} else {
  *prev_tgt_val = FALSE;
}
```
All LAPI_Rmw calls are non-blocking. To test for completion, use the LAPI_Getcntr and LAPI_Waitcntr subroutines. LAPI_Rmw does not include a target counter (tgt_cntr), so LAPI_Rmw calls do not provide any indication of completion on the target task (tgt).

Parameters

INPUT

hdl  Specifies the LAPI handle.

op  Specifies the operation to be performed. The valid operations are:
  • COMPARE_AND_SWAP
  • FETCH_AND_ADD
  • FETCH_AND_OR
  • SWAP

tgt  Specifies the task ID of the target task where the read-modify-write (Rmw) variable resides. The value of this parameter must be in the range 0 <= tgt < NUM_TASKS.

tgt_var  Specifies the target read-modify-write (Rmw) variable (in FORTRAN) or its address (in C). The value of this parameter cannot be NULL (in C) or LAPI_ADDR_NULL (in FORTRAN).

in_val  Specifies the value that is passed in to the operation (op). This value cannot be NULL (in C) or LAPI_ADDR_NULL (in FORTRAN).

INPUT/OUTPUT

prev_tgt_val  Specifies the location at the origin in which the previous tgt_var on the target task is stored before the operation (op) is executed. The value of this parameter can be NULL (in C) or LAPI_ADDR_NULL (in FORTRAN).

org_cntr  Specifies the origin counter address (in C) or the origin counter (in FORTRAN). If prev_tgt_val is set, the origin counter (org_cntr) is incremented when prev_tgt_val is returned to the origin side. If prev_tgt_val is not set, the origin counter (org_cntr) is updated after the operation (op) is completed at the target side.

OUTPUT

ierror  Specifies a FORTRAN return code. This is always the last parameter.

Restrictions

LAPI statistics are not reported for shared memory communication and data transfer, or for messages that a task sends to itself.

C Examples

1. To synchronize a data value between two tasks (with FETCH_AND_ADD):

```c
int local_var;
int *addr_list;

/* both tasks initialize local_var to a value */
/* local_var addresses are exchanged and stored */
/* in addr_list (using LAPI_Address_init). */
/* addr_list[tgt] now contains the address of */
/* local_var on tgt */
/* add value to local_var on some task */
```
2. To synchronize a data value between two tasks (with **SWAP**):

```c
int local_var;
int *addr_list;

/* local_var addresses are exchanged and stored */
/* in addr_list (using LAPI_Address_init). */
/* addr_list[tgt] now contains the address of */
/* local_var on tgt. */

/* local_var is assigned some value */

/* assign local_var to local_var on remote task */
LAPI_Rmw(hndl, SWAP, tgt, addr_list[tgt],
         local_var, prev_tgt_val, &org_cntr);

/* local_var on the remote task is now equal to */
/* local_var on the local task. prev_tgt_val now */
/* contains the value of local_var on the remote */
/* task before the swap. */
```

3. To conditionally swap a data value (with **COMPARE_AND_SWAP**):

```c
int local_var;
int *addr_list;
int in_val[2];

/* local_var addresses are exchanged and stored */
/* in addr_list (using LAPI_Address_init). */
/* addr_list[tgt] now contains the address of */
/* local_var on tgt. */

/* if local_var on remote_task is equal to comparator, */
/* assign value to local_var on remote task */

in_val[0] = comparator;
in_val[1] = value;

LAPI_Rmw(hndl, COMPARE_AND_SWAP, tgt, addr_list[tgt],
         in_val, prev_tgt_val, &org_cntr);

/* local_var on the remote task is now in_val[1] if it */
/* had previously been equal to in_val[0]. If the swap */
/* was performed, prev_tgt_val now contains TRUE; */
/* otherwise, it contains FALSE. */
```
Return Values

**LAPI_SUCCESS**
Indicates that the function call completed successfully.

**LAPI_ERR_HNDL_INVALID**
Indicates that the *hndl* passed in is not valid (not initialized or in terminated state).

**LAPI_ERR_IN_VAL_NULL**
Indicates that the *in_val* pointer is NULL (in C) or that the value of *in_val* is **LAPI_ADDR_NULL** (in FORTRAN).

**LAPI_ERR_RMW_OP**
Indicates that *op* is not valid.

**LAPI_ERR_TGT**
Indicates that the *tgt* passed in is outside the range of tasks defined in the job.

**LAPI_ERR_TGT_PURGED**
Indicates that the subroutine returned early because **LAPI_Purge_totask()** was called.

**LAPI_ERR_TGT_VAR_NULL**
Indicates that the *tgt_var* address is NULL (in C) or that the value of *tgt_var* is **LAPI_ADDR_NULL** (in FORTRAN).

Location

/usr/lib/liblapi_r.a

Related Information

Subroutines: **LAPI_Address_init**, **LAPI_Getcntr**, **LAPI_Qenv**, **LAPI_Rmw64**, **LAPI_Setcntr**, **LAPI_Waitcntr**, **LAPI_Xfer**

__LAPI_Rmw64 Subroutine__

**Purpose**
Provides data synchronization primitives for 64-bit applications.

**Library**
Availability Library (liblapi_r.a)

**C Syntax**

```c
#include <lapi.h>

int LAPI_Rmw64(hndl, op, tgt, tgt_var, in_val, prev_tgt_val, org_cntr)

lapi_handle_t hndl;
Rmw_ops_t op;
uint tgt;
long long *tgt_var;
long long *in_val;
long long *prev_tgt_val;
lapi_cntr_t *org_cntr;
```

**FORTRAN Syntax**

```fortran
include 'lapif.h'

LAPI_RMW64(hndl, op, tgt, tgt_var, in_val, prev_tgt_val, org_cntr, ierror)

INTEGER hndl
INTEGER op
INTEGER tgt
```
INTEGER (KIND=LAPI_ADDR_TYPE) :: tgt_var
INTEGER (KIND=LAPI_LONG_LONG_TYPE) :: in_val, prev_tgt_val
TYPE (LAPI_CNTR_T) :: org_cntr
INTEGER ierror

Description

Type of call: point-to-point communication (non-blocking)

This subroutine is the 64-bit version of LAPI_Rmw. It is used to synchronize two independent pieces of 64-bit data, such as two tasks sharing a common data structure. The operation is performed at the target task (tgt) and is atomic. The operation takes an input value (in_val) from the origin and performs one of four operations (op) on a variable (tgt_var) at the target (tgt), and then replaces the target variable (tgt_var) with the results of the operation (op). The original value (prev_tgt_val) of the target variable (tgt_var) is returned to the origin.

The operations (op) are performed over the context referred to by hndl. The outcome of the execution of these calls is as if the following code was executed atomically:

\[
*\text{prev\_tgt\_val} = *\text{tgt\_var};
*\text{tgt\_var} = f(*\text{tgt\_var}, *\text{in\_val});
\]

where:

\[f(a,b) = a + b \text{ for } \text{FETCH\_AND\_ADD}\]

\[f(a,b) = a | b \text{ for } \text{FETCH\_AND\_OR} \text{ (bitwise or)}\]

\[f(a,b) = b \text{ for } \text{SWAP}\]

For COMPARE\_AND\_SWAP, in_val is treated as a pointer to an array of two integers, and the op is the following atomic operation:

\[
\text{if}(*\text{tgt\_var} == \text{in\_val}[0]) \{
  *\text{prev\_tgt\_val} = \text{TRUE};
  *\text{tgt\_var} = \text{in\_val}[1];
\} \text{ else } \{
  *\text{prev\_tgt\_val} = \text{FALSE};
\}
\]

This subroutine can also be used on a 32-bit processor.

All LAPI_Rmw64 calls are non-blocking. To test for completion, use the LAPI_Getcntr and LAPI_Waitcntr subroutines. LAPI_Rmw64 does not include a target counter (tgt_cntr), so LAPI_Rmw64 calls do not provide any indication of completion on the target task (tgt).

Parameters

INPUT

\textit{hndl} \quad \text{Specifies the LAPI handle.}

\textit{op} \quad \text{Specifies the operation to be performed. The valid operations are:}

\begin{itemize}
  \item COMPARE\_AND\_SWAP
  \item FETCH\_AND\_ADD
  \item FETCH\_AND\_OR
  \item SWAP
\end{itemize}
tgt  Specifies the task ID of the target task where the read-modify-write (Rmw64) variable resides. The value of this parameter must be in the range $0 \leq tgt < \text{NUM_TASKS}$.

tgt_var  Specifies the target read-modify-write (Rmw64) variable (in FORTRAN) or its address (in C). The value of this parameter cannot be NULL (in C) or LAPI_ADDR_NULL (in FORTRAN).

in_val  Specifies the value that is passed in to the operation ($op$). This value cannot be NULL (in C) or LAPI_ADDR_NULL (in FORTRAN).

INPUT/OUTPUT

prev_tgt_val  Specifies the location at the origin in which the previous $tgt \_var$ on the target task is stored before the operation ($op$) is executed. The value of this parameter can be NULL (in C) or LAPI_ADDR_NULL (in FORTRAN).

org_cntr  Specifies the origin counter address (in C) or the origin counter (in FORTRAN). If $prev \_tgt \_val$ is set, the origin counter ($org \_cntr$) is incremented when $prev \_tgt \_val$ is returned to the origin side. If $prev \_tgt \_val$ is not set, the origin counter ($org \_cntr$) is updated after the operation ($op$) is completed at the target side.

OUTPUT

ierror  Specifies a FORTRAN return code. This is always the last parameter.

Restrictions
LAPI statistics are not reported for shared memory communication and data transfer, or for messages that a task sends to itself.

C Examples
1. To synchronize a data value between two tasks (with FETCH\_AND\_ADD):

   ```c
   { 
   long long local_var;
   long long *addr_list;
   
   /* both tasks initialize local_var to a value */
   /* local_var addresses are exchanged and stored */
   /* in addr_list (using LAPI Address_init64) */
   /* addr_list[tgt] now contains address of */
   /* local_var on tgt */
   .
   .
   .
   /* add value to local_var on some task */
   .
   .
   .
   /* use LAPI to add value to local_var on remote task */
   LAPI_Rmw64(hndl, FETCH\_AND\_ADD, tgt, addr_list[tgt],
                value, prev_tgt_val, &org_cntr);
   
   /* local_var on remote task has been increased */
   /* by value. prev_tgt_val now contains value of */
   /* local_var on remote task before the addition */
   }
   ```

2. To synchronize a data value between two tasks (with SWAP):

   ```c
   { 
   long long local_var;
   long long *addr_list;
   ```
3. To conditionally swap a data value (with COMPARE_AND_SWAP):

```c
{
    long long local_var;
    long long *addr_list;
    long long in_val[2];

    /* local_var addresses are exchanged and stored */
    /* in addr_list (using LAPI_Address_init64). */
    /* addr_list[tgt] now contains the address of */
    /* local_var on tgt. */
    .
    .
    /* local_var is assigned some value */
    /* assign local_var to local_var on the remote task */
    LAPI_Rmw64(hndl, SWAP, tgt, addr_list[tgt],
               local_var, prev_tgt_val, &org_cntr);

    /* local_var on the remote task is now equal to local_var */
    /* on the local task. prev_tgt_val now contains the value */
    /* of local_var on the remote task before the swap. */
}
```

**Return Values**

**LAPI_SUCCESS**
Indicates that the function call completed successfully.

**LAPI_ERR_HNDL_INVALID**
Indicates that the `hndl` passed in is not valid (not initialized or in terminated state).

**LAPI_ERR_IN_VAL_NULL**
Indicates that the `in_val` pointer is NULL (in C) or that the value of `in_val` is `LAPI_ADDR_NULL` (in FORTRAN).

**LAPI_ERR_RMW_OP**
Indicates that `op` is not valid.

**LAPI_ERR_TGT**
Indicates that the `tgt` passed in is outside the range of tasks defined in the job.

**LAPI_ERR_TGT_PURGED**
Indicates that the subroutine returned early because `LAPI_Purge_totask()` was called.
**LAPI_ERR_TGT_VAR_NULL** Indicates that the `tgt_var` address is NULL (in C) or that the value of `tgt_var` is `LAPI_ADDR_NULL` (in FORTRAN).

**Location**

/usr/lib/liblapi_r.a

**Related Information**

Subroutines: `LAPI_Address_init64`, `LAPI_Getcntr`, `LAPI_Qenv`, `LAPI_Rmw`, `LAPI_Setcntr`, `LAPI_Waitcntr`, `LAPI_Xfer`

---

**LAPI_Senv Subroutine**

**Purpose**

Used to set a runtime variable.

**Library**

Availability Library (`liblapi_r.a`)

**C Syntax**

```c
#include <lapif.h>

int LAPI_Senv(hndl, query, set_val)
  lapi_handle_t hndl;
  lapi_query_t query;
  int set_val;
```

**FORTRAN Syntax**

```fortran
include 'lapif.h'

LAPI_SENV(hndl, query, set_val, ierror)
  INTEGER hndl
  INTEGER query
  INTEGER set_val
  INTEGER ierror
```

**Description**

*Type of call:* local queries

Use this subroutine to set runtime attributes for a specific LAPI instance. In C, the `lapi_query_t` enumeration defines the attributes that can be set at runtime. These attributes are defined explicitly in FORTRAN. See `LAPI_Qenv` for more information.

You can use `LAPI_Senv` to set these runtime attributes: `ACK_THRESHOLD`, `ERROR_CHK`, `INTERRUPT_SET`, and `TIMEOUT`.

**Parameters**

**INPUT**

`hndl` Specifies the LAPI handle.

`query` Specifies the type of query that you want to set. In C, the values for `query` are defined by the `lapi_query_t` enumeration in `lapif.h`. In FORTRAN, these values are defined explicitly in the 32-bit version and the 64-bit version of `lapif.h`.

set_val  Specifies the integer value of the query that you want to set.

OUTPUT
ierror  Specifies a FORTRAN return code. This is always the last parameter.

Restrictions
LAPI statistics are *not* reported for shared memory communication and data transfer, or for messages that a task sends to itself.

C Examples
The following values can be set using LAPI_Senv:

ACK_THRESHOLD:
int value;
LAPI_Senv(hndl, ACK_THRESHOLD, value);
/* LAPI sends packet acknowledgements (acks) in groups, waiting until */
/* ACK_THRESHOLD packets have arrived before returning a group of acks */
/* The valid range for ACK_THRESHOLD is (1 <= value <= 30) */
/* The default is 30. */

ERROR_CHK:
boolean toggle;
LAPI_Senv(hndl, ERROR_CHK, toggle);
/* Indicates whether LAPI should perform error checking. If set, LAPI */
/* calls will perform bounds-checking on parameters. Error checking */
/* is disabled by default. */

INTERRUPT_SET:
boolean toggle;
LAPI_Senv(hndl, INTERRUPT_SET, toggle);
/* Determines whether LAPI will respond to interrupts. If interrupts */
/* are disabled, LAPI will poll for message completion. */
/* toggle==True will enable interrupts, False will disable. */
/* Interrupts are enabled by default. */

TIMEOUT:
int value;
LAPI_Senv(hndl, TIMEOUT, value);
/* LAPI will time out on a communication if no response is received */
/* within timeout seconds. Valid range is (10 <= timeout <= 86400). */
/* 86400 seconds = 24 hours. Default value is 900 (15 minutes). */

Return Values
LAPI_SUCCESS  Indicates that the function call completed successfully.
LAPI_ERR_HNDL_INVALID  Indicates that the *hndl* passed in is not valid (not initialized or in terminated state).
LAPI_ERR_QUERY_TYPE  Indicates the query passed in is not valid.
LAPI_ERR_SET_VAL  Indicates the set_val pointer is not in valid range.

Location
/usr/lib/liblapi_r.a

Related Information
Subroutines: LAPI_Qenv
LAPI_Setcntr Subroutine

Purpose
Used to set a counter to a specified value.

Library
Availability Library (liblapi_r.a)

C Syntax
#include <lapi.h>

int LAPI_Setcntr(hndl, cntr, val)
lacli_handle_t hndl;
lacli_cntr_t *cntr;
int val;

FORTRAN Syntax
include 'lapif.h'

LAPI_SETCNTR(hndl, cntr, val, ierror)
INTEGER hndl
INTEGER (LAPI_CNTR_T) :: cntr
INTEGER val
INTEGER ierror

Description
Type of call: Local counter manipulation

This subroutine sets cntr to the value specified by val. Because the LAPI_Getcntr/LAPI_Setcntr sequence cannot be made atomic, you should only use LAPI_Setcntr when you know there will not be any competing operations.

Parameters

INPUT
hndl Specifies the LAPI handle.
val Specifies the value to which the counter needs to be set.

INPUT/OUTPUT
cntr Specifies the address of the counter to be set (in C) or the counter structure (in FORTRAN). The value of this parameter cannot be NULL (in C) or LAPI_ADDR_NULL (in FORTRAN).

OUTPUT
ierror Specifies a FORTRAN return code. This is always the last parameter.

Restrictions
LAPI statistics are not reported for shared memory communication and data transfer, or for messages that a task sends to itself.

C Examples
To initialize a counter for use in a communication API call:
{  
lapi_cntr_t my_tgt_cntr, *tgt_cntr_array;
  int initial_value, expected_value, current_value;
  lapi_handle_t hndl;
  ...

  /* Note: the code below is executed on all tasks */
  /* initialize, allocate and create structures */
  initial_value = 0;
  expected_value = 1;

  /* set the cntr to zero */
  LAPI_Setcntr(hndl, &my_tgt_cntr, initial_value);
  /* set other counters */
  ...

  /* exchange counter addresses, LAPI_Address_init synchronizes */
  LAPI_Address_init(hndl, &my_tgt_cntr, tgt_cntr_array);
  /* more address exchanges */
  ...

  /* Communication calls using my_tgt_cntr */
  LAPI_Put(..., tgt_cntr_array[tgt], ....);
  ...

  /* Wait for counter to reach value */
  for (;;) {
      LAPI_Getcntr(hndl, &my_tgt_cntr, &current_value);
      if (current_value >= expected_value) {
          break; /* out of infinite loop */
      } else {
          LAPI_Probe(hndl);
      }
  }
  ...

  /* Quiesce/synchronize to ensure communication using our counter is done */
  LAPI_Gfence(hndl);
  /* Reset the counter */
  LAPI_Setcntr(hndl, &my_tgt_cntr, initial_value);
  /* Synchronize again so that no other communication using the counter can */
  /* begin from any other task until we're all finished resetting the counter. */
  LAPI_Gfence(hndl);
  /* More communication calls */
  ...
}

Return Values

LAPI_SUCCESS Indicates that the function call completed successfully.
LAPI_ERR_CNTR_NULL Indicates that the cntr value passed in is NULL (in C) or
LAPI_ADDR_NULL (in FORTRAN).
LAPI_ERR_HNDL_INVALID  Indicates that the *hndl* passed in is not valid (not initialized or in terminated state).

Location
/usr/lib/liblapi_r.a

Related Information
Subroutines: LAPI_Getcntr, LAPI_Waitcntr

LAPI_Setcntr_wstatus  Subroutine

Purpose
Used to set a counter to a specified value and to set the associated destination list array and destination status array to the counter.

Library
Availability Library (liblapi_r.a)

C Syntax

```
#include <lapi.h>

int LAPI_Setcntr_wstatus(hndl, cntr, num_dest, dest_list, dest_status)
```

typedef lapi_handle_t  hndl;
typedef lapi_cntr_t   *cntr;
typedef int num_dest;
typedef uint *dest_list;
typedef int *dest_status;

FORTRAN Syntax

```
include 'lapif.h'

LAPI_SETCNTR_WSTATUS(hndl, cntr, num_dest, dest_list, dest_status, ierror)
```

 INTEGER hndl
 TYPE (LAPI_CNTR_T) :: cntr
 INTEGER num_dest
 INTEGER dest_list(*)
 INTEGER dest_status
 INTEGER ierror

Description

Type of call: recovery

This subroutine sets *cntr* to 0. Use LAPI_Setcntr_wstatus to set the associated destination list array (*dest_list*) and destination status array (*dest_status*) to the counter. Use a corresponding LAPI_Nopoll_wait call to access these arrays. These arrays record the status of a task from where the thread calling LAPI_Nopoll_wait() is waiting for a response.

The return values for *dest_status* are:

- LAPI_MSG_INITIAL  The task is purged or is not received.
- LAPI_MSG_RECVD  The task is received.
- LAPI_MSG_PURGED  The task is purged, but not received.
LAPI_MSG_PURGED_RCVD  The task is received and then purged.
LAPI_MSG_INVALID      Not valid; the task is already purged.

Note: To use this subroutine, the lib_vers field in the lapi_info_t structure must be set to L2_LIB or LAST_LIB.

Params

**INPUT**

`hndl` Specifies the LAPI handle.

`num_dest` Specifies the number of tasks in the destination list.

`dest_list` Specifies an array of destinations waiting for this counter update. If the value of this parameter is NULL (in C) or `LAPI_ADDR_NULL` (in FORTRAN), no status is returned to the user.

**INPUT/OUTPUT**

`cntr` Specifies the address of the counter to be set (in C) or the counter structure (in FORTRAN). The value of this parameter cannot be NULL (in C) or `LAPI_ADDR_NULL` (in FORTRAN).

**OUTPUT**

`dest_status` Specifies an array of status that corresponds to `dest_list`. The value of this parameter can be NULL (in C) or `LAPI_ADDR_NULL` (in FORTRAN).

`ierror` Specifies a FORTRAN return code. This is always the last parameter.

**Restrictions**

Use of this subroutine is *not* recommended on a system that is running Parallel Environment (PE).

**Return Values**

<table>
<thead>
<tr>
<th>Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>LAPI_SUCCESS</strong></td>
<td>Indicates that the function call completed successfully.</td>
</tr>
<tr>
<td><strong>LAPI_ERR_CNTR_NULL</strong></td>
<td>Indicates that the <code>cntr</code> value passed in is NULL (in C) or <code>LAPI_ADDR_NULL</code> (in FORTRAN).</td>
</tr>
<tr>
<td><strong>LAPI_ERR_HNDL_INVALID</strong></td>
<td>Indicates that the <code>hndl</code> passed in is not valid (not initialized or in terminated state).</td>
</tr>
<tr>
<td><strong>LAPI_ERR_RET_PTR_NULL</strong></td>
<td>Indicates that the value of <code>dest_status</code> is NULL in C (or <code>LAPI_ADDR_NULL</code> in FORTRAN), but the value of <code>dest_list</code> is not NULL in C (or <code>LAPI_ADDR_NULL</code> in FORTRAN).</td>
</tr>
</tbody>
</table>

**Location**

/usr/lib/liblapi_r.a

**Related Information**

Subroutines: `LAPI_Getcntr`, `LAPI_Nopoll_wait`, `LAPI_Purge_totask`, `LAPI_Setcntr`

**LAPI_Term Subroutine**

**Purpose**

Terminates and cleans up a LAPI context.
Library
Availability Library (liblapi_r.a)

C Syntax
#include <lapi.h>

int LAPI_Term(hndl)
    lapi_handle_t hndl;

FORTRAN Syntax
include 'lapif.h'

LAPI_TERM(hndl, ierror)
INTEGER hndl
INTEGER ierror

Description
Type of call: local termination

Use this subroutine to terminate the LAPI context that is specified by hndl. Any LAPI notification threads that are associated with this context are terminated. An error occurs when any LAPI calls are made using hndl after LAPI_Term is called.

A DGSP that is registered under that LAPI handle remains valid even after LAPI_Term is called on hndl.

Parameters

INPUT

hndl Specifies the LAPI handle.

OUTPUT

ierror Specifies a FORTRAN return code. This is always the last parameter.

Restrictions
LAPI statistics are not reported for shared memory communication and data transfer, or for messages that a task sends to itself.

C Examples
To terminate a LAPI context (represented by hndl):

LAPI_Term(hndl);

Return Values

LAPI_SUCCESS Indicates that the function call completed successfully.

LAPI_ERR_HNDL_INVALID Indicates that the hndl passed in is not valid (not initialized or in terminated state).

Location
/usr/lib/liblapi_r.a

Related Information
Subroutines: LAPI_Init, LAPI_Purge_totask, LAPI_Resume_totask
**LAPI_Util Subroutine**

**Purpose**
Serves as a wrapper function for such data gather/scatter operations as registration and reservation, for updating UDP port information, and for obtaining pointers to locking and signaling functions that are associated with a shared LAPI lock.

**Library**
Availability Library (liblapi_r.a)

**C Syntax**
```c
#include <lapi.h>

int LAPI_Util(hndl, util_cmd)
    lapi_handle_t hndl;
    lapi_util_t *util_cmd;
```

**FORTRAN Syntax**
```fortran
include 'lapif.h'

LAPI_UTIL(hndl, util_cmd, ierror)
INTEGER hndl
TYPE (LAPI_UTIL_T) :: util_cmd
INTEGER ierror
```

**Description**
**Type of call:** Data gather/scatter program (DGSP), UDP port information, and lock sharing utilities

This subroutine is used for several different operations, which are indicated by the command type value in the beginning of the command structure. The `lapi_util_t` structure is defined as:

```c
typedef union {
    lapi_util_type_t Util_type;
    lapi_reg_dgsp_t RegDgsp;
    lapi_dref_dgsp_t DrefDgsp;
    lapi_resv_dgsp_t ResvDgsp;
    lapi_reg_ddm_t DdmFunc;
    lapi_add_udp_port_t Udp;
    lapi_pack_dgsp_t PackDgsp;
    lapi_unpack_dgsp_t UnpackDgsp;
    lapi_thread_func_t ThreadFunc;
} lapi_util_t;
```

The enumerated type `lapi_util_type_t` has these values:

<table>
<thead>
<tr>
<th>Value of <code>Util_type</code></th>
<th>Union member as interpreted by LAPI_Util</th>
</tr>
</thead>
<tbody>
<tr>
<td>LAPI_REGISTER_DGSP</td>
<td>lapi_reg_dgsp_t</td>
</tr>
<tr>
<td>LAPI_UNRESERVE_DGSP</td>
<td>lapi_dref_dgsp_t</td>
</tr>
<tr>
<td>LAPI_RESERVE_DGSP</td>
<td>lapi_resv_dgsp_t</td>
</tr>
<tr>
<td>LAPI_REG_DDM_FUNC</td>
<td>lapi_reg_ddm_t</td>
</tr>
<tr>
<td>LAPI_ADD_UDP_DEST_PORT</td>
<td>lapi_add_udp_port_t</td>
</tr>
<tr>
<td>LAPI_DGSP_PACK</td>
<td>lapi_pack_dgsp_t</td>
</tr>
<tr>
<td>LAPI_DGSP_UNPACK</td>
<td>lapi_unpack_dgsp_t</td>
</tr>
</tbody>
</table>
Table 1. lapi_util_type_t types (continued)

<table>
<thead>
<tr>
<th>Value of Util_type</th>
<th>Union member as interpreted by LAPI_Util</th>
</tr>
</thead>
<tbody>
<tr>
<td>LAPI_GET_THREAD_FUNC</td>
<td>lapi_thread_func_t</td>
</tr>
</tbody>
</table>

*hndl* is not checked for command type LAPI_REGISTER_DGSP, LAPI_RESERVE_DGSP, or LAPI_UNRESERVE_DGSP.

**LAPI_REGISTER_DGSP**

You can use this operation to register a LAPI DGSP that you have created. To register a LAPI DGSP, lapi_dgsp_descr_t idgsp must be passed in. LAPI returns a handle (lapi_dg_handle_t dgsp_handle) to use for all future LAPI calls. The dgsp_handle that is returned by a register operation is identified as a lapi_dg_handle_t type, which is the appropriate type for LAPI_Xfer and LAPI_Util calls that take a DGSP. This returned dgsp_handle is also defined to be castable to a pointer to a lapi_dgsp_descr_t for those situations where the LAPI user requires read-only access to information that is contained in the cached DGSP. The register operation delivers a DGSP to LAPI for use in future message send, receive, pack, and unpack operations. LAPI creates its own copy of the DGSP and protects it by reference count. All internal LAPI operations that depend on a DGSP cached in LAPI ensure the preservation of the DGSP by incrementing the reference count when they begin a dependency on the DGSP and decrementing the count when that dependency ends. A DGSP, once registered, can be used from any LAPI instance. LAPI_Term does not discard any DGSPs.

You can register a DGSP, start one or more LAPI operations using the DGSP, and then unreserve it with no concern about when the LAPI operations that depend on the DGSP will be done using it. See LAPI_RESERVE_DGSP and LAPI_UNRESERVE_DGSP for more information.

In general, the DGSP you create and pass in to the LAPI_REGISTER_DGSP call using the dgsp parameter is discarded after LAPI makes and caches its own copy. Because DGSP creation is complex, user errors may occur, but extensive error checking at data transfer time would hurt performance. When developing code that creates DGSPs, you can invoke extra validation at the point of registration by setting the LAPI_VERIFY_DGSP environment variable. LAPI_Util will return any detected errors. Any errors that exist and are not detected at registration time will cause problems during data transfer. Any errors detected during data transfer will be reported by an asynchronous error handler. A segmentation fault is one common symptom of a faulty DGSP. If multiple DGSPs are in use, the asynchronous error handler will not be able to identify which DGSP caused the error. For more information about asynchronous error handling, see LAPI_Init.

**LAPI_REGISTER_DGSP** uses the lapi_reg_dgsp_t command structure.

Table 2. The lapi_reg_dgsp_t fields

<table>
<thead>
<tr>
<th>lapi_reg_dgsp_t field</th>
<th>lapi_util_type_t field</th>
<th>lapi_reg_dgsp_t usage</th>
</tr>
</thead>
<tbody>
<tr>
<td>Util_type</td>
<td>lapi_dgsp_descr_t</td>
<td>LAPI_REGISTER_DGSP</td>
</tr>
<tr>
<td>idgsp</td>
<td>lapi_dgsp_descr_t</td>
<td>IN - pointer to DGSP program</td>
</tr>
<tr>
<td>dgsp_handle</td>
<td>lapi_dg_handle_t</td>
<td>OUT - handle for a registered DGSP program</td>
</tr>
<tr>
<td>in usr_func</td>
<td>lapi_usr_fcall_t</td>
<td>For debugging only</td>
</tr>
<tr>
<td>status</td>
<td>lapi_status_t</td>
<td>OUT - future support</td>
</tr>
</tbody>
</table>

**LAPI_RESERVE_DGSP**

You can use this operation to reserve a DGSP. This operation is provided because a LAPI client might cache a LAPI DGSP handle for later use. The client needs to ensure the DGSP will not be discarded before the cached handle is used. A DGSP handle, which is defined to be a pointer to a DGSP description that is already cached inside LAPI, is passed to this operation. The DGSP handle is also defined to be a structure pointer, so that client programs can get direct access to information in the DGSP. Unless the client can be certain that the DGSP will not be "unreserved" by another thread while it is being accessed,
the client should bracket the access window with its own reserve/unreserve operation. The client is not to modify the cached DGSP, but LAPI has no way to enforce this. The reserve operation increments the user reference count, thus protecting the DGSP until an unreserve operation occurs. This is needed because the thread that placed the reservation will expect to be able to use or examine the cached DGSP until it makes an unreserve call (which decrements the user reference count), even if the unreserve operation that matches the original register operation occurs within this window on some other thread.

**LAPI_RESERVE_DGSP** uses the `lapi_resv_dgsp_t` command structure.

<table>
<thead>
<tr>
<th>Field</th>
<th>Type</th>
<th>Usage</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>Util_type</code></td>
<td><code>lapi_util_type_t</code></td>
<td>LAPI_RESERVE_DGSP</td>
</tr>
<tr>
<td><code>dgsp_handle</code></td>
<td><code>lapi_dg_handle_t</code></td>
<td>OUT - handle for a registered DGSP program</td>
</tr>
<tr>
<td><code>in_usr_func</code></td>
<td><code>lapi_usr_fcall_t</code></td>
<td>For debugging only</td>
</tr>
<tr>
<td><code>status</code></td>
<td><code>lapi_status_t</code></td>
<td>OUT - future support</td>
</tr>
</tbody>
</table>

**LAPI_UNRESERVE_DGSP**
You can use this operation to unregister or unreserve a DGSP. This operation decrements the user reference count. If external and internal reference counts are zero, this operation lets LAPI free the DGSP. All operations that decrement a reference count cause LAPI to check to see if the counts have both become 0 and if they have, dispose of the DGSP. Several internal LAPI activities increment and decrement a second reference count. The cached DGSP is disposable only when all activities (internal and external) that depend on it and use reference counting to preserve it have discharged their reference. The DGSP handle is passed to LAPI as a value parameter and LAPI does not nullify the caller’s handle. It is your responsibility to not use this handle again because in doing an unreserve operation, you have indicated that you no longer count on the handle remaining valid.

**LAPI_UNRESERVE_DGSP** uses the `lapi_dref_dgsp_t` command structure.

<table>
<thead>
<tr>
<th>Field</th>
<th>Type</th>
<th>Usage</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>Util_type</code></td>
<td><code>lapi_util_type_t</code></td>
<td>LAPI_UNRESERVE_DGSP</td>
</tr>
<tr>
<td><code>dgsp_handle</code></td>
<td><code>lapi_dg_handle_t</code></td>
<td>OUT - handle for a registered DGSP program</td>
</tr>
<tr>
<td><code>in_usr_func</code></td>
<td><code>lapi_usr_fcall_t</code></td>
<td>For debugging only</td>
</tr>
<tr>
<td><code>status</code></td>
<td><code>lapi_status_t</code></td>
<td>OUT - future support</td>
</tr>
</tbody>
</table>

**LAPI_REG_DDM_FUNC**
You can use this operation to register data distribution manager (DDM) functions. It works in conjunction with the DGSM CONTROL instruction. Primarily, it is used for **MPI_Accumulate**, but LAPI clients can provide any DDM function. It is also used to establish a callback function for processing data that is being scattered into a user buffer on the destination side.

The native LAPI user can install a callback without affecting the one MPI has registered for **MPI_Accumulate**. The function prototype for the callback function is:

```c
typedef long ddm_func_t ( /* return number of bytes processed */
    void *in,     /* pointer to inbound data */
    void *inout,  /* pointer to destination space */
    long bytes,   /* number of bytes inbound */
    int operand,  /* CONTROL operand value */
    int operation /* CONTROL operation value */
);``
A DDM function acts between the arrival of message data and the target buffer. The most common usage is to combine inbound data with data already in the target buffer. For example, if the target buffer is an array of integers and the incoming message consists of integers, the DDM function can be written to add each incoming integer to the value that is already in the buffer. The operand and operation fields of the DDM function allow one DDM function to support a range of operations with the CONTROL instruction by providing the appropriate values for these fields.

See RSCT for AIX 5L: LAPI Programming Guide for more information about DGSP programming.

**LAPI_REG_DDM_FUNC** uses the *lapi_reg_ddm_t* command structure. Each call replaces the previous function pointer, if there was one.

<table>
<thead>
<tr>
<th>lapi_reg_ddm_t field</th>
<th>lapi_reg_ddm_t type</th>
<th>lapi_reg_ddm_t usage</th>
</tr>
</thead>
<tbody>
<tr>
<td>Util_type</td>
<td>lapi_util_type_t</td>
<td>LAPI_REG_DDM_FUNC</td>
</tr>
<tr>
<td>ddm_func</td>
<td>ddm_func_t *</td>
<td>IN - DDM function pointer</td>
</tr>
<tr>
<td>in_usr_func</td>
<td>lapi_usr_fcall_t</td>
<td>For debugging only</td>
</tr>
<tr>
<td>status</td>
<td>lapi_status_t</td>
<td>OUT - future support</td>
</tr>
</tbody>
</table>

**LAPI_DGSP_PACK**

You can use this operation to gather data to a pack buffer from a user buffer under control of a DGSP. A single buffer may be packed by a series of calls. The caller provides a position value that is initialized to the starting offset within the buffer. Each pack operation adjusts position, so the next pack operation can begin where the previous pack operation ended. In general, a series of pack operations begins with position initialized to 0, but any offset is valid. There is no state carried from one pack operation to the next. Each pack operation starts at the beginning of the DGSP it is passed.

**LAPI_DGSP_PACK** uses the *lapi_pack_dgsp_t* command structure.

<table>
<thead>
<tr>
<th>lapi_pack_dgsp_t field</th>
<th>lapi_pack_dgsp_t field type</th>
<th>lapi_pack_dgsp_t usage</th>
</tr>
</thead>
<tbody>
<tr>
<td>Util_type</td>
<td>lapi_util_type_t</td>
<td>LAPI_DGSP_PACK</td>
</tr>
<tr>
<td>dgsp_handle</td>
<td>lapi_dg_handle_t</td>
<td>OUT - handle for a registered DGSP program</td>
</tr>
<tr>
<td>in_buf</td>
<td>void *</td>
<td>IN - source buffer to pack</td>
</tr>
<tr>
<td>bytes</td>
<td>ulong</td>
<td>IN - number of bytes to pack</td>
</tr>
<tr>
<td>out_buf</td>
<td>void *</td>
<td>OUT - output buffer for pack</td>
</tr>
<tr>
<td>out_size</td>
<td>ulong</td>
<td>IN - output buffer size in bytes</td>
</tr>
<tr>
<td>position</td>
<td>ulong</td>
<td>IN/OUT - current buffer offset</td>
</tr>
<tr>
<td>in_usr_func</td>
<td>lapi_usr_fcall_t</td>
<td>For debugging only</td>
</tr>
<tr>
<td>status</td>
<td>lapi_status_t</td>
<td>OUT - future support</td>
</tr>
</tbody>
</table>

**LAPI_DGSP_UNPACK**

You can use this operation to scatter data from a packed buffer to a user buffer under control of a DGSP. A single buffer may be unpacked by a series of calls. The caller provides a position value that is initialized to the starting offset within the packed buffer. Each unpack operation adjusts position, so the next unpack operation can begin where the previous unpack operation ended. In general, a series of unpack operations begins with position initialized to 0, but any offset is valid. There is no state carried from one unpack operation to the next. Each unpack operation starts at the beginning of the DGSP it is passed.
**LAPI_DGSP_UNPACK** uses the `lapi_unpack_dgsp_t` command structure.

<table>
<thead>
<tr>
<th><code>lapi_unpack_dgsp_t</code> field</th>
<th><code>lapi_unpack_dgsp_t</code> field type</th>
<th><code>lapi_unpack_dgsp_t</code> usage</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>Util_type</code></td>
<td><code>lapi_util_type_t</code></td>
<td><code>LAPI_DGSP_UNPACK</code></td>
</tr>
<tr>
<td><code>dgsp_handle</code></td>
<td><code>lapi_dg_handle_t</code></td>
<td>OUT - handle for a registered DGSP program</td>
</tr>
<tr>
<td><code>buf</code></td>
<td><code>void *</code></td>
<td>IN - source buffer for unpack</td>
</tr>
<tr>
<td><code>in_size</code></td>
<td><code>ulong</code></td>
<td>IN - source buffer size in bytes</td>
</tr>
<tr>
<td><code>out_buf</code></td>
<td><code>void *</code></td>
<td>OUT - output buffer for unpack</td>
</tr>
<tr>
<td><code>bytes</code></td>
<td><code>ulong</code></td>
<td>IN - number of bytes to unpack</td>
</tr>
<tr>
<td><code>out_size</code></td>
<td><code>ulong</code></td>
<td>IN - output buffer size in bytes</td>
</tr>
<tr>
<td><code>position</code></td>
<td><code>ulong</code></td>
<td>IN/OUT - current buffer offset</td>
</tr>
<tr>
<td><code>in_usr_func</code></td>
<td><code>lapi_usr_fcall_t</code></td>
<td>For debugging only</td>
</tr>
<tr>
<td><code>status</code></td>
<td><code>lapi_status_t</code></td>
<td>OUT - future support</td>
</tr>
</tbody>
</table>

**LAPI_ADD_UDP_DEST_PORT**

You can use this operation to update UDP port information about the destination task. This operation can be used when you have written your own UDP handler (`udp_hndlr`) and you need to support recovery of failed tasks. You cannot use this operation under the POE runtime environment.

**LAPI_ADD_UDP_DEST_PORT** uses the `lapi_add_udp_port_t` command structure.

<table>
<thead>
<tr>
<th><code>lapi_add_udp_port_t</code> field</th>
<th><code>lapi_add_udp_port_t</code> field type</th>
<th><code>lapi_add_udp_port_t</code> usage</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>Util_type</code></td>
<td><code>lapi_util_type_t</code></td>
<td><code>LAPI_ADD_UDP_DEST_PORT</code></td>
</tr>
<tr>
<td><code>tgt</code></td>
<td><code>uint</code></td>
<td>IN - destination task ID</td>
</tr>
<tr>
<td><code>udp_port</code></td>
<td><code>lapi_udp_t *</code></td>
<td>IN - UDP port information for the target</td>
</tr>
<tr>
<td><code>instance_no</code></td>
<td><code>uint</code></td>
<td>IN - Instance number of UDP</td>
</tr>
<tr>
<td><code>in_usr_func</code></td>
<td><code>lapi_usr_fcall_t</code></td>
<td>For debugging only</td>
</tr>
<tr>
<td><code>status</code></td>
<td><code>lapi_status_t</code></td>
<td>OUT - future support</td>
</tr>
</tbody>
</table>

**LAPI_GET_THREAD_FUNC**

You can use this operation to retrieve various shared locking and signalling functions. Retrieval of these functions is valid only after LAPI is initialized and before LAPI is terminated. You should not call any of these functions after LAPI is terminated.

**LAPI_GET_THREAD_FUNC** uses the `lapi_thread_func_t` command structure.

<table>
<thead>
<tr>
<th><code>lapi_thread_func_t</code> field</th>
<th><code>lapi_thread_func_t</code> field type</th>
<th><code>lapi_thread_func_t</code> usage</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>Util_type</code></td>
<td><code>lapi_util_type_t</code></td>
<td><code>LAPI_GET_THREAD_FUNC</code></td>
</tr>
<tr>
<td><code>mutex_lock</code></td>
<td><code>lapi_mutex_lock_t</code></td>
<td>OUT - mutex lock function pointer</td>
</tr>
<tr>
<td><code>mutex_unlock</code></td>
<td><code>lapi_mutex_unlock_t</code></td>
<td>OUT - mutex unlock function pointer</td>
</tr>
<tr>
<td><code>mutex_trylock</code></td>
<td><code>lapi_mutex_trylock_t</code></td>
<td>OUT - mutex try lock function pointer</td>
</tr>
<tr>
<td><code>mutex_getowner</code></td>
<td><code>lapi_mutex_getowner_t</code></td>
<td>OUT - mutex get owner function pointer</td>
</tr>
<tr>
<td><code>cond_wait</code></td>
<td><code>lapi_cond_wait_t</code></td>
<td>OUT - condition wait function pointer</td>
</tr>
<tr>
<td><code>cond_timedwait</code></td>
<td><code>lapi_cond_timedwait_t</code></td>
<td>OUT - condition timed wait function pointer</td>
</tr>
</tbody>
</table>
Table 9. The lapi_thread_func_t fields (continued)

<table>
<thead>
<tr>
<th>lapi_thread_func_t field</th>
<th>lapi_thread_func_t field type</th>
<th>lapi_thread_func_t usage</th>
</tr>
</thead>
<tbody>
<tr>
<td>cond_signal</td>
<td>lapi_cond_signal_t</td>
<td>OUT - condition signal function pointer</td>
</tr>
<tr>
<td>cond_init</td>
<td>lapi_cond_init_t</td>
<td>OUT - initialize condition function pointer</td>
</tr>
<tr>
<td>cond_destroy</td>
<td>lapi_cond_destroy_t</td>
<td>OUT - destroy condition function pointer</td>
</tr>
</tbody>
</table>

LAPI uses the pthread library for thread ID management. You can therefore use `pthread_self()` to get the running thread ID and `lapi_mutex_getowner_t` to get the thread ID that owns the shared lock. Then, you can use `pthread_equal()` to see if the two are the same.

**Mutex thread functions:** LAPI_GET_THREAD_FUNC includes the following mutex thread functions: mutex lock, mutex unlock, mutex try lock, and mutex get owner.

**Mutex lock function pointer**

```c
int (*lapi_mutex_lock_t)(lapi_handle_t hndl);
```

This function acquires the lock that is associated with the specified LAPI handle. The call blocks if the lock is already held by another thread. Deadlock can occur if the calling thread is already holding the lock. You are responsible for preventing and detecting deadlocks.

**Parameters**

**INPUT**

*hndl* Specifies the LAPI handle.

**Return values**

0 Indicates that the lock was acquired successfully.

EINVAL Is returned if the lock is not valid because of an incorrect *hndl* value.

**Mutex unlock function pointer**

```c
int (*lapi_mutex_unlock_t)(lapi_handle_t hndl);
```

This function releases the lock that is associated with the specified LAPI handle. A thread should only unlock its own locks.

**Parameters**

**INPUT**

*hndl* Specifies the LAPI handle.

**Return values**

0 Indicates that the lock was released successfully.

EINVAL Is returned if the lock is not valid because of an incorrect *hndl* value.

**Mutex try lock function pointer**

```c
int (*lapi_mutex_trylock_t)(lapi_handle_t hndl);
```

This function tries to acquire the lock that is associated with the specified LAPI handle, but returns immediately if the lock is already held.

**Parameters**
INPUT

\texttt{hndl} \quad \text{Specifies the LAPI handle.}

Return values

0 \quad \text{Indicates that the lock was acquired successfully.}

EBUSY \quad \text{Indicates that the lock is being held.}

EINVAL \quad \text{Is returned if the lock is not valid because of an incorrect } \texttt{hndl} \text{ value.}

Mutex get owner function pointer

\begin{verbatim}
int (*lapi_mutex_getowner_t)(lapi_handle_t \texttt{hndl}, pthread_t *tid);
\end{verbatim}

This function gets the pthread ID of the thread that is currently holding the lock associated with the specified LAPI handle. \texttt{LAPI_NULL_THREAD_ID} indicates that the lock is not held at the time the function is called.

Parameters

INPUT

\texttt{hndl} \quad \text{Specifies the LAPI handle.}

OUTPUT

\texttt{tid} \quad \text{Is a pointer to hold the pthread ID to be retrieved.}

Return values

0 \quad \text{Indicates that the lock owner was retrieved successfully.}

EINVAL \quad \text{Is returned if the lock is not valid because of an incorrect } \texttt{hndl} \text{ value.}

\textbf{Condition functions:} \texttt{LAPI\_GET\_THREAD\_FUNC} includes the following condition functions: condition wait, condition timed wait, condition signal, initialize condition, and destroy condition.

Condition wait function pointer

\begin{verbatim}
int (*lapi_cond_wait_t)(lapi_handle_t \texttt{hndl}, lapi_cond_t *\texttt{cond});
\end{verbatim}

This function waits on a condition variable (\texttt{cond}). The user must hold the lock associated with the LAPI handle (\texttt{hndl}) before making the call. Upon the return of the call, LAPI guarantees that the lock is still being held. The same LAPI handle must be supplied to concurrent \texttt{lapi\_cond\_wait\_t} operations on the same condition variable.

Parameters

INPUT

\texttt{hndl} \quad \text{Specifies the LAPI handle.}

\texttt{cond} \quad \text{Is a pointer to the condition variable to be waited on.}

Return values

0 \quad \text{Indicates that the condition variable has been signaled.}

EINVAL \quad \text{Indicates that the value specified by } \texttt{hndl} \text{ or } \texttt{cond} \text{ is not valid.}

Condition timed wait function pointer
int (*lapi_cond_timedwait_t)(lapi_handle_t hndl, lapi_cond_t *cond, struct timespec *timeout);

This function waits on a condition variable (cond). The user must hold the lock associated with the LAPI handle (hndl) before making the call. Upon the return of the call, LAPI guarantees that the lock is still being held. The same LAPI handle must be supplied to concurrent lapi_cond_timedwait_t operations on the same condition variable.

Parameters

INPUT

hndl Specifies the LAPI handle.
cond Is a pointer to the condition variable to be waited on.
timeout Is a pointer to the absolute time structure specifying the timeout.

Return values

0 Indicates that the condition variable has been signaled.
ETIMEDOUT Indicates that time specified by timeout has passed.
EINVAL Indicates that the value specified by hndl, cond, or timeout is not valid.

Condition signal function pointer

int (*lapi_cond_wait_t)(lapi_handle_t hndl, lapi_cond_t *cond);
typedef int (*lapi_cond_signal_t)(lapi_handle_t hndl, lapi_cond_t *cond);

This function signals a condition variable (cond) to wake up a thread that is blocked on the condition. If there are multiple threads waiting on the condition variable, which thread to wake up is decided randomly.

Parameters

INPUT

hndl Specifies the LAPI handle.
cond Is a pointer to the condition variable to be signaled.

Return values

0 Indicates that the condition variable has been signaled.
EINVAL Indicates that the value specified by hndl or cond is not valid.

Initialize condition function pointer

int (*lapi_cond_init_t)(lapi_handle_t hndl, lapi_cond_t *cond);

This function initializes a condition variable.

Parameters

INPUT

hndl Specifies the LAPI handle.
cond Is a pointer to the condition variable to be initialized.

Return values

0 Indicates that the condition variable was initialized successfully.
EAGAIN Indicates that the system lacked the necessary resources (other than memory) to initialize another condition variable.

ENOMEM Indicates that there is not enough memory to initialize the condition variable.

EINVAL Is returned if the hndl value is not valid.

Destroy condition function pointer
int (*lapi_cond_destroy_t)(lapi_handle_t hndl, lapi_cond_t *cond);

This function destroys a condition variable.

Parameters
INPUT
hndl Specifies the LAPI handle.
cond Is a pointer to the condition variable to be destroyed.

Return values
0 Indicates that the condition variable was destroyed successfully.
EBUSY Indicates that the implementation has detected an attempt to destroy the object referenced by cond while it is referenced (while being used in a lapi_cond_wait_t or lapi_cond_timedwait_t by another thread, for example).
EINVAL Indicates that the value specified by hndl or cond is not valid.

Parameters
INPUT
hndl Specifies the LAPI handle.

INPUT/OUTPUT
util_cmd Specifies the command type of the utility function.

OUTPUT
ierror Specifies a FORTRAN return code. This is always the last parameter.

Return Values
LAPI_SUCCESS Indicates that the function call completed successfully.
LAPI_ERR_DGSP Indicates that the DGSP that was passed in is NULL (in C) or LAPI_ADDR_NULL (in FORTRAN) or is not a registered DGSP.
LAPI_ERR_DGSP_ATOM Indicates that the DGSP has an atom_size that is less than 0 or greater than MAX_ATOM_SIZE.
LAPI_ERR_DGSP_BRANCH Indicates that the DGSP attempted a branch that fell outside of the code array. This is returned only in validation mode.
LAPI_ERR_DGSP_COPY_SZ Is returned with DGSP validation turned on when MCOPY block < 0 or COPY instruction with bytes < 0. This is returned only in validation mode.
LAPI_ERR_DGSP_FREE Indicates that LAPI tried to free a DGSP that is not valid or is no longer registered. There should be one LAPI_UNRESERVE_DGSP operation to
close the **LAPI_REGISTER_DGSP** operation and one **LAPI_UNRESERVE_DGSP** operation for each **LAPI_RESERVE_DGSP** operation.

**LAPI_ERR_DGSP_OP**

Indicates that the DGSP *opcode* is not valid. This is returned only in validation mode.

**LAPI_ERR_DGSP_STACK**

Indicates that the DGSP has a greater GOSUB depth than the allocated stack supports. Stack allocation is specified by the `dgsp->depth` member. This is returned only in validation mode.

**LAPI_ERR_HNDL_INVALID**

Indicates that the *hndl* passed in is not valid (not initialized or in terminated state).

**LAPI_ERR_MEMORY_EXHAUSTED**

Indicates that LAPI is unable to obtain memory from the system.

**LAPI_ERR_UDP_PORT_INFO**

Indicates that the *udp_port* information pointer is NULL (in C) or that the value of *udp_port* is **LAPI_ADDR_NULL** (in FORTRAN).

**LAPI_ERR_UTIL_CMD**

Indicates that the command type is not valid.

### C Examples

1. To create and register a DGSP:

```c
{
    /*
    ** DGSP code array. DGSP instructions are stored
    ** as ints (with constants defined in lapi.h for
    ** the number of integers needed to store each
    ** instruction). We will have one COPY and one ITERATE
    ** instruction in our DGSP. We use LAPI's constants
    ** to allocate the appropriate storage.
    */
    int code[LAPI_DGSM_COPY_SIZE+LAPI_DGSM_ITERATE_SIZE];

    /* DGSP description */
    lapi_dgsp_descr_t dgsp_d;

    /*
    ** Data structure for the xfer call.
    */
    lapi_xfer_t xfer_struct;

    /*
    */
    lapi_dgsm_copy_t *copy_p; /* copy instruction */
    lapi_dgsm_iterate_t *iter_p; /* iterate instruction */
    int *code_ptr; /* code pointer */

    /* constant for holding code array info */
    int code_less_iterate_size;

    /* used for DGSP registration */
    lapi_reg_dgsp_t reg_util;

    /*
    ** Set up dgsp description
    */
    /* set pointer to code array */
    dgsp_d.code = &code[0];
}
```
/** set size of code array */
dgsp_d.code_size = LAPI_DGSM_COPY_SIZE + LAPI_DGSM_ITERATE_SIZE;

/** not using DGSP gosub instruction */
dgsp_d.depth = 1;

/ *
  ** set density to show internal gaps in the
  ** DGSP data layout
  */
dgsp_d.density = LAPI_DGSM_SPARSE;

/ *
  ** transfer 4 bytes at a time */
dgsp_d.size = 4;

/ *
  ** advance the template by 8 for each iteration */
dgsp_d.extent = 8;

/ *
  ** ext specifies the memory 'footprint' of
  ** data to be transferred. The lext specifies
  ** the offset from the base address to begin
  ** viewing the data. The rext specifies the
  ** length from the base address to use.
  */
dgsp_d.lext = 0;
dgsp_d.rext = 4;

/ *
  ** atom size of 0 lets LAPI choose the packet size */
dgsp_d.atom_size = 0;

/ *
  ** set up the copy instruction
  */
copy_p = (lapi_dgsm_copy_t*)(dgsp_d.code);
copy_p->opcode = LAPI_DGSM_COPY;

/ *
  ** copy 4 bytes at a time */
copy_p->bytes = (long)4;

/ *
  ** start at offset 0 */
copy_p->offset = (long)0;

/ *
  ** set code pointer to address of iterate instruction */
code_less_iterate_size = dgsp_d.code_size - LAPI_DGSM_ITERATE_SIZE;

code_ptr = ((int*)(code))+code_less_iterate_size;

/ *
  ** Set up iterate instruction
  */
iter_p = (lapi_dgsm_iterate_t*)code_ptr;
iter_p->opcode = LAPI_DGSM_ITERATE;
iter_p->iter_loc = (-code_less_iterate_size);

/ *
  ** Set up and do DGSP registration */
reg_util.Util_type = LAPI_REGISTER_DGSP;
reg_util.idgsp = &dgsp_d;
LAPI_Util(hndl, reg_util);

/ *
  ** LAPI returns a usable DGSP handle in
  ** reg_util.dgsp_handle
  ** Use this handle for subsequent reserve/unreserve
  ** and Xfer calls. On the receive side, this
  ** handle can be returned by the header handler using the
  ** return_info_t mechanism. The DGSP will then be used for
2. To reserve a DGSP handle:

```c
reg_util.dgsp_handle = dgsp_handle;
/*
** dgsp_handle has already been created and
** registered as in the above example
*/
reg_util.Util_type = LAPI_RESERVE_DGSP;
LAPI_Util(hndl, (lapi_util_t *)&reg_util);
/*
** LAPI's internal reference count to dgsp_handle
** will be incremented. DGSP will
** remain available until an unreserve is
** done for each reserve, plus one more for
** the original registration.
*/
```

3. To unreserve a DGSP handle:

```c
reg_util.dgsp_handle = dgsp_handle;
/*
** dgsp_handle has already created and
** registered as in the above example, and
** this thread no longer needs it.
*/
reg_util.Util_type = LAPI_UNRESERVE_DGSP;
LAPI_Util(hndl, (lapi_util_t *)&reg_util);
/*
** An unreserve is required for each reserve,
** plus one more for the original registration.
*/
```

** Location**

/usr/lib/liblapi_r.a

**Related Information**

Subroutines: LAPI_Init, LAPI_Xfer

**LAPI_Waitcntr Subroutine**

**Purpose**

Waits until a specified counter reaches the value specified.
Library
Availability Library (liblapi_r.a)

C Syntax
#include <lapi.h>

int LAPI_Waitcntr(hndl, cntr, val, cur_cntr_val)
laip_handle_t  hndl;
lapi_cntr_t*  *cntr;
int           val;
int           *cur_cntr_val;

FORTRAN Syntax
include 'lapif.h'

LAPI_WAITCNTR(hndl, cntr, val, cur_cntr_val, ierror)
INTEGER      hndl
TYPE (LAPI_CNTR_T) :: cntr
INTEGER      val
INTEGER      cur_cntr_val
INTEGER      ierror

Description
Type of call: local progress monitor (blocking)

This subroutine waits until cntr reaches or exceeds the specified val. Once cntr reaches val, cntr is decremented by the value of val. In this case, "decremented" is used (as opposed to "set to zero") because cntr could have contained a value that was greater than the specified val when the call was made. This call may or may not check for message arrivals over the LAPI context hndl. The cur_cntr_val variable is set to the current counter value.

Parameters

INPUT
hndl          Specifies the LAPI handle.
val           Specifies the value the counter needs to reach.

INPUT/OUTPUT
cntr          Specifies the counter structure (in FORTRAN) to be waited on or its address (in C). The value of this parameter cannot be NULL (in C) or LAPI_ADDR_NULL (in FORTRAN).

OUTPUT
cur_cntr_val  Specifies the integer value of the current counter. This value can be NULL (in C) or LAPI_ADDR_NULL (in FORTRAN).
ierror        Specifies a FORTRAN return code. This is always the last parameter.

Restrictions
LAPI statistics are not reported for shared memory communication and data transfer, or for messages that a task sends to itself.

C Examples
To wait on a counter to reach a specified value:
Return Values

**LAPI_SUCCESS** Indicates that the function call completed successfully.

**LAPI_ERR_CNTR_NULL** Indicates that the `cntr` pointer is NULL (in C) or that the value of `cntr` is `LAPI_ADDR_NULL` (in FORTRAN).

**LAPI_ERR_HNDL_INVALID** Indicates that the `hndl` passed in is not valid (not initialized or in terminated state).

Location

/usr/lib/liblapi_r.a

Related Information

Subroutines: LAPI_Amsend, LAPI_Amsendv, LAPI_Get, LAPI_Getcntr, LAPI_Getv, LAPI_Put, LAPI_Putv, LAPI_Rmw, LAPI_Rmw64, LAPI_Setcntr, LAPI_Xfer

---

**LAPI_Xfer Subroutine**

**Purpose**

Serves as a wrapper function for LAPI data transfer functions.

**Library**

Availability Library (liblapi_r.a)

**C Syntax**

```c
#include <lapi.h>

int LAPI_Xfer(hndl, xfer_cmd)
    lapl_handle_t hndl;
    lapi_xfer_t *xfer_cmd;

typedef struct {
    uint src;    /* Target task ID */
    uint reason; /* LAPI return codes */
    ulong reserve[6]; /* Reserved */
} lapi_sh_info_t;

typedef void (scompl_hndlr_t)(lapl_handle_t *hndl, void *completion_param,
    lapi_sh_info_t *info);
```
FORTRAN Syntax

include 'lapif.h'

LAPI_XFER(hndl, xfer_cmd, ierror)
INTEGER hndl
TYPE (fortran_xfer_type) :: xfer_cmd
INTEGER ierror

Description

Type of call: point-to-point communication (non-blocking)

The LAPI_Xfer subroutine provides a superset of the functionality of these subroutines: LAPI_Amsend, LAPI_Amsendv, LAPI_Put, LAPI_Putv, LAPI_Get, LAPI_Getv, and LAPI_Rmw. In addition, LAPI_Xfer provides data gather/scatter program (DGSP) messages transfer.

In C, the LAPI_Xfer command is passed a pointer to a union. It examines the first member of the union, Xfer_type, to determine the transfer type, and to determine which union member was passed. LAPI_Xfer expects every field of the identified union member to be set. It does not examine or modify any memory outside of the identified union member. LAPI_Xfer treats all union members (except status) as read-only data.

This subroutine provides the following functions:

- The remote address fields are expanded to be of type lapi_long_t, which is long enough for a 64-bit address. This allows a 32-bit task to send data to 64-bit addresses, which may be important in client/server programs.
- LAPI_Xfer allows the origin counter to be replaced with a send completion callback.
- LAPI_Xfer is used to transfer data using LAPI's data gather/scatter program (DGSP) interface.

The lapi_xfer_t structure is defined as:

type definition union {
    lapi_xfer_type_t Xfer_type;
    lapi_get_t Get;
    lapi_am_t Am;
    lapi_rmw_t Rmw;
    lapi_put_t Put;
    lapi_getv_t Getv;
    lapi_putv_t Putv;
    lapi_amv_t Amv;
    lapi_amdgsp_t Dgsp;
} lapi_xfer_t;

Though the lapi_xfer_t structure applies only to the C version of LAPI_Xfer, the following tables include the FORTRAN equivalents of the C datatypes.

Table 10 list the values of the lapi_xfer_type_t structure for C and the explicit Xfer_type values for FORTRAN.

<table>
<thead>
<tr>
<th>Value of Xfer_type (C or FORTRAN)</th>
<th>Union member as interpreted by LAPI_Xfer (C)</th>
<th>Value of fortran_xfer_type (FORTRAN)</th>
</tr>
</thead>
<tbody>
<tr>
<td>LAPI_AM_XFER</td>
<td>lapi_am_t</td>
<td>LAPI_AM_T</td>
</tr>
<tr>
<td>LAPI_AMV_XFER</td>
<td>lapi_amv_t</td>
<td>LAPI_AMV_T</td>
</tr>
<tr>
<td>LAPI_DGSP_XFER</td>
<td>lapi_amdgsp_t</td>
<td>LAPI_AMDGSP_T</td>
</tr>
<tr>
<td>LAPI_GET_XFER</td>
<td>lapi_get_t</td>
<td>LAPI_GET_T</td>
</tr>
<tr>
<td>LAPI_GETV_XFER</td>
<td>lapi_getv_t</td>
<td>LAPI_GETV_T</td>
</tr>
</tbody>
</table>
Table 10. LAPI_Xfer structure types (continued)

<table>
<thead>
<tr>
<th>Value of Xfer_type (C or FORTRAN)</th>
<th>Union member as interpreted by LAPI_Xfer (C)</th>
<th>Value of fortran_xfer_type (FORTRAN)</th>
</tr>
</thead>
<tbody>
<tr>
<td>LAPI_PUT_XFER</td>
<td>lapi_put_t</td>
<td>LAPI_PUT_T</td>
</tr>
<tr>
<td>LAPI_PUTV_XFER</td>
<td>lapi_putv_t</td>
<td>LAPI_PUTV_T</td>
</tr>
<tr>
<td>LAPI_RMW_XFER</td>
<td>lapi_rmw_t</td>
<td>LAPI_RMW_T</td>
</tr>
</tbody>
</table>

**lapi_am_t details**

Table 11 shows the correspondence among the parameters of the LAPI_Amsend subroutine, the fields of the C lapi_am_t structure and their datatypes, and the equivalent FORTRAN datatypes. The lapi_am_t fields are listed in Table 11 in the order that they occur in the lapi_xfer_t structure.

Table 11. LAPI_Amsend and lapi_am_t equivalents

<table>
<thead>
<tr>
<th>lapi_am_t field name (C)</th>
<th>lapi_am_t field type (C)</th>
<th>Equivalent FORTRAN datatype</th>
<th>Equivalent LAPI_Amsend parameter</th>
</tr>
</thead>
<tbody>
<tr>
<td>Xfer_type</td>
<td>lapi_xfer_type_t</td>
<td>INTEGER(KIND = 4)</td>
<td>implicit in C LAPI_Xfer value in FORTRAN: LAPI_AM_XFER</td>
</tr>
<tr>
<td>flags</td>
<td>int</td>
<td>INTEGER(KIND = 4)</td>
<td>none LAPI_Xfer parameter in FORTRAN: flags</td>
</tr>
<tr>
<td>tgt</td>
<td>uint</td>
<td>INTEGER(KIND = 4)</td>
<td>tgt LAPI_Xfer parameter in FORTRAN: pad</td>
</tr>
<tr>
<td>none</td>
<td>none</td>
<td>INTEGER(KIND = 4)</td>
<td>LAPI_Xfer parameter in FORTRAN: hdr_hdl</td>
</tr>
<tr>
<td>hdr_hdl</td>
<td>lapi_long_t</td>
<td>INTEGER(KIND = 8)</td>
<td>hdr_hdl LAPI_Xfer parameter in FORTRAN (64-bit): pad2</td>
</tr>
<tr>
<td>uhdr_len</td>
<td>uint</td>
<td>INTEGER(KIND = 4)</td>
<td>uhdr_len LAPI_Xfer parameter in FORTRAN (64-bit): pad2</td>
</tr>
<tr>
<td>uhdr</td>
<td>void (*)</td>
<td>INTEGER(KIND = 4)</td>
<td>uhdr</td>
</tr>
<tr>
<td>udata</td>
<td>void (*)</td>
<td>INTEGER(KIND = 4)</td>
<td>udata</td>
</tr>
<tr>
<td>udata_len</td>
<td>ulong</td>
<td>INTEGER(KIND = 4)</td>
<td>udata_len</td>
</tr>
<tr>
<td>shdlr</td>
<td>scompl_hndlr_t *</td>
<td>INTEGER(KIND = 4)</td>
<td>none LAPI_Xfer parameter in FORTRAN: shdlr</td>
</tr>
<tr>
<td>sinfo</td>
<td>void (*)</td>
<td>INTEGER(KIND = 4)</td>
<td>none LAPI_Xfer parameter in FORTRAN: sinfo</td>
</tr>
<tr>
<td>tgt_cntr</td>
<td>lapi_long_t</td>
<td>INTEGER(KIND = 8)</td>
<td>tgt_cntr</td>
</tr>
<tr>
<td>org_cntr</td>
<td>lapi_cntr_t *</td>
<td>INTEGER(KIND = 4)</td>
<td>org_cntr</td>
</tr>
<tr>
<td>cmpl_cntr</td>
<td>lapi_cntr_t *</td>
<td>INTEGER(KIND = 4)</td>
<td>cmpl_cntr</td>
</tr>
</tbody>
</table>

When the origin data buffer is free to be used, the pointer to the send completion handler (shdlr) is called
with the send completion data \((sinfo)\) if \(shdlr\) is not a NULL pointer (in C) or \(\text{LAPI\_ADDR\_NULL}\) (in FORTRAN). Otherwise, the behavior is identical to that of \(\text{LAPI\_Amsend}\).

**lapi_amv_t details**

Table 12 shows the correspondence among the parameters of the \(\text{LAPI\_Amsendv}\) subroutine, the fields of the C \(\text{lapi\_amv\_t}\) structure and their datatypes, and the equivalent FORTRAN datatypes. The \(\text{lapi\_amv\_t}\) fields are listed in Table 12 in the order that they occur in the \(\text{lapi\_xfer\_t}\) structure.

<table>
<thead>
<tr>
<th>lapi_amv_t field name (C)</th>
<th>lapi_amv_t field type (C)</th>
<th>Equivalent FORTRAN datatype</th>
<th>Equivalent LAPI_Amsendv parameter</th>
</tr>
</thead>
<tbody>
<tr>
<td>Xfer_type</td>
<td>lapi_xfer_type_t</td>
<td>INTEGER(KIND = 4)</td>
<td>implicit in C</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>(\text{LAPI_Xfer}) value in FORTRAN: (\text{LAPI_AMV_XFER})</td>
</tr>
<tr>
<td>flags</td>
<td>int</td>
<td>INTEGER(KIND = 4)</td>
<td>none</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>(\text{LAPI_Xfer}) parameter in FORTRAN: (\text{flags})</td>
</tr>
<tr>
<td>tgt</td>
<td>uint</td>
<td>INTEGER(KIND = 4)</td>
<td>tgt</td>
</tr>
<tr>
<td>none</td>
<td>none</td>
<td>INTEGER(KIND = 4)</td>
<td>(\text{LAPI_Xfer}) parameter in FORTRAN: (\text{pad})</td>
</tr>
<tr>
<td>hdr_hdl</td>
<td>lapi_long_t</td>
<td>INTEGER(KIND = 8)</td>
<td>hdr_hdl</td>
</tr>
<tr>
<td>uhdr_len</td>
<td>uint</td>
<td>INTEGER(KIND = 4)</td>
<td>uhdr_len</td>
</tr>
<tr>
<td>none</td>
<td>none</td>
<td>INTEGER(KIND = 4)</td>
<td>(\text{LAPI_Xfer}) parameter in FORTRAN: (\text{pad})</td>
</tr>
<tr>
<td>uhdr</td>
<td>void *</td>
<td>INTEGER(KIND = 4)</td>
<td>uhdr</td>
</tr>
<tr>
<td></td>
<td></td>
<td>(32-bit)</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>(64-bit)</td>
<td></td>
</tr>
<tr>
<td>shdlr</td>
<td>scompl_hndlr_t *</td>
<td>INTEGER(KIND = 4)</td>
<td>none</td>
</tr>
<tr>
<td></td>
<td></td>
<td>(32-bit)</td>
<td>(\text{LAPI_Xfer}) parameter in FORTRAN: (\text{shdlr})</td>
</tr>
<tr>
<td></td>
<td></td>
<td>(64-bit)</td>
<td></td>
</tr>
<tr>
<td>sinfo</td>
<td>void *</td>
<td>INTEGER(KIND = 4)</td>
<td>none</td>
</tr>
<tr>
<td></td>
<td></td>
<td>(32-bit)</td>
<td>(\text{LAPI_Xfer}) parameter in FORTRAN: (\text{sinfo})</td>
</tr>
<tr>
<td></td>
<td></td>
<td>(64-bit)</td>
<td></td>
</tr>
<tr>
<td>org_vec</td>
<td>lapi_vec_t *</td>
<td>INTEGER(KIND = 4)</td>
<td>org_vec</td>
</tr>
<tr>
<td></td>
<td></td>
<td>(32-bit)</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>(64-bit)</td>
<td></td>
</tr>
<tr>
<td>none</td>
<td>none</td>
<td>INTEGER(KIND = 4)</td>
<td>(\text{LAPI_Xfer}) parameter in FORTRAN: (\text{pad})</td>
</tr>
<tr>
<td>tgt_cntr</td>
<td>lapi_long_t</td>
<td>INTEGER(KIND = 8)</td>
<td>tgt_cntr</td>
</tr>
<tr>
<td>org_cntr</td>
<td>lapi_cntr_t *</td>
<td>INTEGER(KIND = 4)</td>
<td>org_cntr</td>
</tr>
<tr>
<td></td>
<td></td>
<td>(32-bit)</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>(64-bit)</td>
<td></td>
</tr>
<tr>
<td>cmpl_cntr</td>
<td>lapi_cntr_t *</td>
<td>INTEGER(KIND = 4)</td>
<td>cmpl_cntr</td>
</tr>
<tr>
<td></td>
<td></td>
<td>(32-bit)</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>(64-bit)</td>
<td></td>
</tr>
</tbody>
</table>

**lapi_amdgsp_t details**

Table 13 on page 669 shows the correspondence among the fields of the C \(\text{lapi\_amdgsp\_t}\) structure and their datatypes, how they are used in \(\text{LAPI\_Xfer}\), and the equivalent FORTRAN datatypes. The \(\text{lapi\_amdgsp\_t}\) fields are listed in Table 13 on page 669 in the order that they occur in the \(\text{lapi\_xfer\_t}\) structure.
<table>
<thead>
<tr>
<th>lapi_amdgsp_t field name (C)</th>
<th>lapi_amdgsp_t field type (C)</th>
<th>Equivalent FORTRAN datatype</th>
<th>LAPI_Xfer usage</th>
</tr>
</thead>
<tbody>
<tr>
<td>Xfer_type</td>
<td>lapi_xfer_type_t</td>
<td>INTEGER(KIND = 4)</td>
<td>LAPI_DGSP_XFER</td>
</tr>
<tr>
<td>flags</td>
<td>int</td>
<td>INTEGER(KIND = 4)</td>
<td>This field allows users to specify directives or hints to LAPI. If you do not want to use any directives or hints, you must set this field to 0. See “The lapi_amdgsp_t flags field” for more information.</td>
</tr>
<tr>
<td>tgt</td>
<td>uint</td>
<td>INTEGER(KIND = 4)</td>
<td>target task</td>
</tr>
<tr>
<td>none</td>
<td>none</td>
<td>INTEGER(KIND = 4)</td>
<td>pad (padding alignment for FORTRAN only)</td>
</tr>
<tr>
<td>hdr_hdl</td>
<td>lapi_long_t</td>
<td>INTEGER(KIND = 8)</td>
<td>header handler to invoke at target</td>
</tr>
<tr>
<td>uhdr_len</td>
<td>uint</td>
<td>INTEGER(KIND = 4)</td>
<td>user header length (multiple of processor’s doubleword size)</td>
</tr>
<tr>
<td>none</td>
<td>none</td>
<td>INTEGER(KIND = 4)</td>
<td>pad2 (padding alignment for 64-bit FORTRAN only)</td>
</tr>
<tr>
<td>uhdr</td>
<td>void *</td>
<td>INTEGER(KIND = 4)</td>
<td>(32-bit) pointer to user header</td>
</tr>
<tr>
<td></td>
<td></td>
<td>INTEGER(KIND = 8)</td>
<td>(64-bit)</td>
</tr>
<tr>
<td>udata</td>
<td>void *</td>
<td>INTEGER(KIND = 4)</td>
<td>(32-bit) pointer to user data</td>
</tr>
<tr>
<td></td>
<td></td>
<td>INTEGER(KIND = 8)</td>
<td>(64-bit)</td>
</tr>
<tr>
<td>udata_len</td>
<td>ulong</td>
<td>INTEGER(KIND = 4)</td>
<td>user data length</td>
</tr>
<tr>
<td></td>
<td></td>
<td>INTEGER(KIND = 8)</td>
<td>(32-bit)</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>(64-bit)</td>
</tr>
<tr>
<td>shdlr</td>
<td>scompl_hndlr_t *</td>
<td>INTEGER(KIND = 4)</td>
<td>(32-bit) send completion handler (optional)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>INTEGER(KIND = 8)</td>
<td>(64-bit)</td>
</tr>
<tr>
<td>sinfo</td>
<td>void *</td>
<td>INTEGER(KIND = 4)</td>
<td>data pointer to pass to send completion handler (optional)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>INTEGER(KIND = 8)</td>
<td>(32-bit)</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>(64-bit)</td>
</tr>
<tr>
<td>tgt_cntr</td>
<td>lapi_long_t</td>
<td>INTEGER(KIND = 8)</td>
<td>target counter (optional)</td>
</tr>
<tr>
<td>org_cntr</td>
<td>lapi_cntr_t *</td>
<td>INTEGER(KIND = 4)</td>
<td>origin counter (optional)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>INTEGER(KIND = 8)</td>
<td>(32-bit)</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>(64-bit)</td>
</tr>
<tr>
<td>compl_cntr</td>
<td>lapi_cntr_t *</td>
<td>INTEGER(KIND = 4)</td>
<td>completion counter (optional)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>INTEGER(KIND = 8)</td>
<td>(32-bit)</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>(64-bit)</td>
</tr>
<tr>
<td>dgsp</td>
<td>lapi_dg_handle_t</td>
<td>INTEGER(KIND = 4)</td>
<td>Handle of a registered DGSP</td>
</tr>
<tr>
<td></td>
<td></td>
<td>INTEGER(KIND = 8)</td>
<td>(32-bit)</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>(64-bit)</td>
</tr>
<tr>
<td>status</td>
<td>lapi_status_t</td>
<td>INTEGER(KIND = 4)</td>
<td>Status to return (future use)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>INTEGER(KIND = 8)</td>
<td>(32-bit)</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>(64-bit)</td>
</tr>
<tr>
<td>none</td>
<td>none</td>
<td>INTEGER(KIND = 4)</td>
<td>pad3 (padding alignment for 64-bit FORTRAN only)</td>
</tr>
</tbody>
</table>

When the origin data buffer is free to be modified, the send completion handler (shdlr) is called with the send completion data (sinfo), if shdlr is not a NULL pointer (in C) or LAPI_ADDR_NULL (in FORTRAN).

See "Using lapi_am_dgsp_t for scatter-side DGSP processing" on page 670 for more information.

**The lapi_amdgsp_t flags field:** One or more flags can be set using the | (bitwise or) operator. User directives are always followed and could result in incorrect results if used improperly. Appropriate hints might improve performance, but they may be ignored by LAPI. Inappropriate hints might degrade performance, but they will not cause incorrect results.
The following directive flags are defined:

**USE_TGT_VEC_TYPE**
Instructs LAPI to use the vector type of the target vector (\( tgt\_vec \)). In other words, \( tgt\_vec \) is to be interpreted as type \textit{lapi_vec_t}; otherwise, it is interpreted as type \textit{lapi_lvec_t}. The \textit{lapi_lvec_t} type uses \textit{lapi_long_t}. The \textit{lapi_vec_t} type uses \textit{void *} or \textit{long}. Incorrect results will occur if one type is used in place of the other.

**BUFFER_BOTH_CONTIGUOUS**
Instructs LAPI to treat all data to be transferred as contiguous, which can improve performance. If this flag is set when non-contiguous data is sent, data will likely be corrupted.

The following hint flags are defined:

**LAPI_NOT_USE_BULK_XFER**
Instructs LAPI not to use bulk transfer, independent of the current setting for the task.

**LAPI_USE_BULK_XFER**
Instructs LAPI to use bulk transfer, independent of the current setting for the task.

If neither of these hint flags is set, LAPI will use the behavior defined for the task. If both of these hint flags are set, \texttt{LAPI\_NOT\_USE\_BULK\_XFER} will take precedence.

These hints may or may not be honored by the communication library.

**Using lapi_am_dgps_t for scatter-side DGSP processing:** Beginning with AIX 5.2, LAPI allows additional information to be returned from the header handler through the use of the \textit{lapi_return_info_t} datatype. See \textit{RSCT for AIX 5L: LAPI Programming Guide} for more information about \textit{lapi_return_info_t}. In the case of transfer type \textit{lapi_am_dgps_t}, this mechanism can be used to instruct LAPI to run a user DGSP to scatter data on the receive side.

To use this mechanism, pass a \textit{lapi_return_info_t *} pointer back to LAPI through the \textit{msg_len} member of the user header handler. The \textit{dgsp\_handle} member of the passed structure must point to a DGSP description that has been registered on the receive side. See \textit{LAPI\_Util} and \textit{RSCT for AIX 5L: LAPI Programming Guide} for details on building and registering DGSPs.

**lapi_get_t details**
Table 14 shows the correspondence among the parameters of the \texttt{LAPI\_Get} subroutine, the fields of the C \texttt{lapi\_get\_t} structure and their datatypes, and the equivalent FORTRAN datatypes. The \texttt{lapi\_get\_t} fields are listed in Table 14 in the order that they occur in the \texttt{lapi\_xfer\_t} structure.

<table>
<thead>
<tr>
<th>\texttt{lapi_get_t} field name (C)</th>
<th>\texttt{lapi_get_t} field type (C)</th>
<th>Equivalent FORTRAN datatype</th>
<th>Equivalent \texttt{LAPI_Get} parameter</th>
</tr>
</thead>
<tbody>
<tr>
<td>Xfer_type</td>
<td>\textit{lapi_xfer_type_t}</td>
<td>\textit{INTEGER(KIND = 4)}</td>
<td>implicit in C \texttt{LAPI_Xfer} value in FORTRAN: \texttt{LAPI_GET_XFER}</td>
</tr>
<tr>
<td>flags</td>
<td>\textit{int}</td>
<td>\textit{INTEGER(KIND = 4)}</td>
<td>none \texttt{LAPI_Xfer} parameter in FORTRAN: \texttt{flags}</td>
</tr>
<tr>
<td>tgt</td>
<td>\textit{uint}</td>
<td>\textit{INTEGER(KIND = 4)}</td>
<td>( tgt ) \texttt{LAPI_Xfer} parameter in FORTRAN: \texttt{pad}</td>
</tr>
<tr>
<td>none</td>
<td>none</td>
<td>\textit{INTEGER(KIND = 4)}</td>
<td>none \texttt{LAPI_Xfer} parameter in FORTRAN: \texttt{pad}</td>
</tr>
</tbody>
</table>
Table 14. LAPI_Get and lapi_get_t equivalents (continued)

<table>
<thead>
<tr>
<th>lapi_get_t field name (C)</th>
<th>lapi_get_t field type (C)</th>
<th>Equivalent FORTRAN datatype</th>
<th>Equivalent LAPI_Get parameter</th>
</tr>
</thead>
<tbody>
<tr>
<td>tgt_addr</td>
<td>lapi_long_t</td>
<td>INTEGER(KIND = 8)</td>
<td>tgt_addr</td>
</tr>
</tbody>
</table>
| org_addr                  | void *                    | INTEGER(KIND = 4) (32-bit)
|                           |                           | INTEGER(KIND = 8) (64-bit)  | org_addr                      |
| tgt_vec                   | void *                    | INTEGER(KIND = 4) (32-bit)
|                           |                           | INTEGER(KIND = 8) (64-bit)  | tgt_vec                       |
| tgt_cntr                  | lapi_long_t               | INTEGER(KIND = 8)           | tgt_cntr                      |
| org_cntr                  | lapi_cntr_t *             | INTEGER(KIND = 4) (32-bit)
|                           |                           | INTEGER(KIND = 8) (64-bit)  | org_cntr                      |
| chndlr                    | compl_hndlr_t *           | INTEGER(KIND = 4) (32-bit)
|                           |                           | INTEGER(KIND = 8) (64-bit)  | none                          |
|                           |                           |                             | LAPI_Xfer parameter in FORTRAN: chndlr |
| cinfo                     | void *                    | INTEGER(KIND = 4) (32-bit)
|                           |                           | INTEGER(KIND = 8) (64-bit)  | none                          |
|                           |                           |                             | LAPI_Xfer parameter in FORTRAN: cinfo |

When the origin data buffer has completely arrived, the pointer to the completion handler (chndlr) is called with the completion data (cinfo), if chndlr is not a NULL pointer (in C) or LAPI_ADDR_NULL (in FORTRAN). Otherwise, the behavior is identical to that of LAPI_Get.

lapi_getv_t details

Table 15 shows the correspondence among the parameters of the LAPI_Getv subroutine, the fields of the C lapi_getv_t structure and their datatypes, and the equivalent FORTRAN datatypes. The lapi_getv_t fields are listed in Table 14 on page 670 in the order that they occur in the lapi_xfer_t structure.

Table 15. LAPI_Getv and lapi_getv_t equivalents

<table>
<thead>
<tr>
<th>lapi_getv_t field name (C)</th>
<th>lapi_getv_t field type (C)</th>
<th>Equivalent FORTRAN datatype</th>
<th>Equivalent LAPI_Getv parameter</th>
</tr>
</thead>
<tbody>
<tr>
<td>Xfer_type</td>
<td>lapi_xfer_type_t</td>
<td>INTEGER(KIND = 4)</td>
<td>implicit in C</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>LAPI_Xfer value in FORTRAN:</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>LAPI_GETV_XFER</td>
</tr>
<tr>
<td>flags</td>
<td>int</td>
<td>INTEGER(KIND = 4)</td>
<td>none</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>LAPI_Xfer parameter in FORTRAN: flags</td>
</tr>
<tr>
<td>tgt</td>
<td>uint</td>
<td>INTEGER(KIND = 4)</td>
<td>tgt</td>
</tr>
<tr>
<td>none</td>
<td>none</td>
<td>INTEGER(KIND = 4)</td>
<td>LAPI_Xfer parameter in FORTRAN (64-bit): pad</td>
</tr>
</tbody>
</table>
| org_vec                   | lapi_vec_t *              | INTEGER(KIND = 4) (32-bit)
|                           |                           | INTEGER(KIND = 8) (64-bit)  | org_vec                       |
| tgt_vec                   | void *                    | INTEGER(KIND = 4) (32-bit)
|                           |                           | INTEGER(KIND = 8) (64-bit)  | tgt_vec                       |
| none                      | none                      | INTEGER(KIND = 4)           | LAPI_Xfer parameter in FORTRAN (32-bit): pad |
| tgt_cntr                  | lapi_long_t               | INTEGER(KIND = 8)           | tgt_cntr                      |
| org_cntr                  | lapi_cntr_t *             | INTEGER(KIND = 4) (32-bit)
|                           |                           | INTEGER(KIND = 8) (64-bit)  | org_cntr                      |
Table 15. LAPI_Getv and lapi_getv_t equivalents (continued)

<table>
<thead>
<tr>
<th>lapi_getv_t field name (C)</th>
<th>lapi_getv_t field type (C)</th>
<th>Equivalent FORTRAN datatype</th>
<th>Equivalent LAPI_Getv parameter</th>
</tr>
</thead>
<tbody>
<tr>
<td>chndlr</td>
<td>compl_hndlr_t *</td>
<td>INTEGER(KIND = 4) (32-bit)</td>
<td>none</td>
</tr>
<tr>
<td></td>
<td></td>
<td>INTEGER(KIND = 8) (64-bit)</td>
<td>LAPI_Xfer parameter in FORTRAN: chndlr</td>
</tr>
<tr>
<td>cinfo</td>
<td>void *</td>
<td>INTEGER(KIND = 4) (32-bit)</td>
<td>none</td>
</tr>
<tr>
<td></td>
<td></td>
<td>INTEGER(KIND = 8) (64-bit)</td>
<td>LAPI_Xfer parameter in FORTRAN: cinfo</td>
</tr>
<tr>
<td>none</td>
<td>none</td>
<td>INTEGER(KIND = 4)</td>
<td>LAPI_Xfer parameter in FORTRAN (32-bit): pad2</td>
</tr>
</tbody>
</table>

The flags field accepts USE_TGT_VEC_TYPE (see “The lapi_amdgsp_t flags field” on page 669) to indicate that tgt_vec is to be interpreted as type lapi_vec_t; otherwise, it is interpreted as type lapi_lvec_t. Note the corresponding field is lapi_vec_t in the LAPI_Getv call.

When the origin data buffer has completely arrived, the pointer to the completion handler (chndlr) is called with the completion data (cinfo) if chndlr is not a NULL pointer (in C) or LAPI_ADDR_NULL (in FORTRAN). Otherwise, the behavior is identical to that of LAPI_Getv.

lapi_put_t details

Table 16 shows the correspondence among the parameters of the LAPI_Put subroutine, the fields of the C lapi_put_t structure and their datatypes, and the equivalent FORTRAN datatypes. The lapi_put_t fields are listed in Table 16 in the order that they occur in the lapi_xfer_t structure.

Table 16. LAPI_Put and lapi_put_t equivalents

<table>
<thead>
<tr>
<th>lapi_put_t field name (C)</th>
<th>lapi_put_t field type (C)</th>
<th>Equivalent FORTRAN datatype</th>
<th>Equivalent LAPI_Put parameter</th>
</tr>
</thead>
<tbody>
<tr>
<td>Xfer_type</td>
<td>lapi_xfer_type_t</td>
<td>INTEGER(KIND = 4)</td>
<td>implicit in C</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>LAPI_Xfer value in FORTRAN: LAPI_PUT_XFER</td>
</tr>
<tr>
<td>flags</td>
<td>int</td>
<td>INTEGER(KIND = 4)</td>
<td>none</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>LAPI_Xfer parameter in FORTRAN: flags</td>
</tr>
<tr>
<td>tgt</td>
<td>uint</td>
<td>INTEGER(KIND = 4)</td>
<td>tgt</td>
</tr>
<tr>
<td>none</td>
<td>none</td>
<td>INTEGER(KIND = 4)</td>
<td>LAPI_Xfer parameter in FORTRAN: pad</td>
</tr>
<tr>
<td>tgt_addr</td>
<td>lapi_long_t</td>
<td>INTEGER(KIND = 8)</td>
<td>tgt_addr</td>
</tr>
<tr>
<td>org_addr</td>
<td>void *</td>
<td>INTEGER(KIND = 4) (32-bit)</td>
<td>org_addr</td>
</tr>
<tr>
<td></td>
<td></td>
<td>INTEGER(KIND = 8) (64-bit)</td>
<td></td>
</tr>
<tr>
<td>len</td>
<td>ulong</td>
<td>INTEGER(KIND = 4) (32-bit)</td>
<td>len</td>
</tr>
<tr>
<td></td>
<td></td>
<td>INTEGER(KIND = 8) (64-bit)</td>
<td></td>
</tr>
<tr>
<td>shdlr</td>
<td>scompl_hndlr_t *</td>
<td>INTEGER(KIND = 4) (32-bit)</td>
<td>none</td>
</tr>
<tr>
<td></td>
<td></td>
<td>INTEGER(KIND = 8) (64-bit)</td>
<td>LAPI_Xfer parameter in FORTRAN: shdlr</td>
</tr>
<tr>
<td>sinfo</td>
<td>void *</td>
<td>INTEGER(KIND = 4) (32-bit)</td>
<td>none</td>
</tr>
<tr>
<td></td>
<td></td>
<td>INTEGER(KIND = 8) (64-bit)</td>
<td>LAPI_Xfer parameter in FORTRAN: sinfo</td>
</tr>
</tbody>
</table>
### Table 16. LAPI_Put and lapi_put_t equivalents (continued)

<table>
<thead>
<tr>
<th>lapi_put_t field name (C)</th>
<th>lapi_put_t field type (C)</th>
<th>Equivalent FORTRAN datatype</th>
<th>Equivalent LAPI_Put parameter</th>
</tr>
</thead>
<tbody>
<tr>
<td>tgt_cntr</td>
<td>lapi_long_t</td>
<td>INTEGER(KIND = 8)</td>
<td>tgt_cntr</td>
</tr>
<tr>
<td>org_cntr</td>
<td>lapi_cntr_t *</td>
<td>INTEGER(KIND = 4) (32-bit)</td>
<td>org_cntr</td>
</tr>
<tr>
<td></td>
<td></td>
<td>INTEGER(KIND = 8) (64-bit)</td>
<td></td>
</tr>
<tr>
<td>cmpl_cntr</td>
<td>lapi_cntr_t *</td>
<td>INTEGER(KIND = 4) (32-bit)</td>
<td>cmpl_cntr</td>
</tr>
<tr>
<td></td>
<td></td>
<td>INTEGER(KIND = 8) (64-bit)</td>
<td></td>
</tr>
</tbody>
</table>

When the origin data buffer is free to be used, the pointer to the send completion handler (shdlr) is called with the send completion data (sinfo), if shdlr is not a NULL pointer (in C) or LAPI_ADDR_NULL (in FORTRAN). Otherwise, the behavior is identical to that of LAPI_Put.

### lapi_putv_t details

Table 17 shows the correspondence among the parameters of the LAPI_Putv subroutine, the fields of the C lapi_putv_t structure and their datatypes, and the equivalent FORTRAN datatypes. The lapi_putv_t fields are listed in Table 16 on page 672 in the order that they occur in the lapi_xfer_t structure.

### Table 17. LAPI_Putv and lapi_putv_t equivalents

<table>
<thead>
<tr>
<th>lapi_putv_t field name (C)</th>
<th>lapi_putv_t field type (C)</th>
<th>Equivalent FORTRAN datatype</th>
<th>Equivalent LAPI_Putv parameter</th>
</tr>
</thead>
<tbody>
<tr>
<td>Xfer_type</td>
<td>lapi_xfer_type_t</td>
<td>INTEGER(KIND = 4)</td>
<td>implicit in C</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>LAPI_Xfer value in FORTRAN:</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>LAPI_PUT_XFER</td>
</tr>
<tr>
<td>flags</td>
<td>int</td>
<td>INTEGER(KIND = 4)</td>
<td>none</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>LAPI_Xfer parameter in</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>FORTRAN: flags</td>
</tr>
<tr>
<td>tgt</td>
<td>uint</td>
<td>INTEGER(KIND = 4)</td>
<td>tgt</td>
</tr>
<tr>
<td>none</td>
<td>none</td>
<td>INTEGER(KIND = 4)</td>
<td>LAPI_Xfer parameter in</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>FORTRAN: flags</td>
</tr>
<tr>
<td>shdlr</td>
<td>scompl_hndlr_t *</td>
<td>INTEGER(KIND = 4) (32-bit)</td>
<td>none</td>
</tr>
<tr>
<td></td>
<td></td>
<td>INTEGER(KIND = 8) (64-bit)</td>
<td>LAPI_Xfer parameter in</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>FORTRAN: shdlr</td>
</tr>
<tr>
<td>sinfo</td>
<td>void *</td>
<td>INTEGER(KIND = 4) (32-bit)</td>
<td>none</td>
</tr>
<tr>
<td></td>
<td></td>
<td>INTEGER(KIND = 8) (64-bit)</td>
<td>LAPI_Xfer parameter in</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>FORTRAN: sinfo</td>
</tr>
<tr>
<td>org_vec</td>
<td>lapi_vec_t *</td>
<td>INTEGER(KIND = 4) (32-bit)</td>
<td>org_vec</td>
</tr>
<tr>
<td></td>
<td></td>
<td>INTEGER(KIND = 8) (64-bit)</td>
<td></td>
</tr>
<tr>
<td>tgt_vec</td>
<td>void *</td>
<td>INTEGER(KIND = 4) (32-bit)</td>
<td>tgt_vec</td>
</tr>
<tr>
<td></td>
<td></td>
<td>INTEGER(KIND = 8) (64-bit)</td>
<td></td>
</tr>
<tr>
<td>none</td>
<td>none</td>
<td>INTEGER(KIND = 4)</td>
<td>LAPI_Xfer parameter in</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>FORTRAN: pad</td>
</tr>
<tr>
<td>tgt_cntr</td>
<td>lapi_long_t</td>
<td>INTEGER(KIND = 8)</td>
<td>tgt_cntr</td>
</tr>
<tr>
<td>org_cntr</td>
<td>lapi_cntr_t *</td>
<td>INTEGER(KIND = 4) (32-bit)</td>
<td>org_cntr</td>
</tr>
<tr>
<td></td>
<td></td>
<td>INTEGER(KIND = 8) (64-bit)</td>
<td></td>
</tr>
<tr>
<td>cmpl_cntr</td>
<td>lapi_cntr_t *</td>
<td>INTEGER(KIND = 4) (32-bit)</td>
<td>cmpl_cntr</td>
</tr>
<tr>
<td></td>
<td></td>
<td>INTEGER(KIND = 8) (64-bit)</td>
<td></td>
</tr>
</tbody>
</table>

The flags field accepts USE_TGT_VEC_TYPE (see "The lapi_amdgsp_t flags field" on page 669) to
indicate that \texttt{tgt_vec} is to be interpreted as \texttt{lapi_vec_t}; otherwise, it is interpreted as a \texttt{lapi_lvec_t}. Note that the corresponding field is \texttt{lapi_vec_t} in the \texttt{LAPI_Putv} call.

When the origin data buffer is free to be modified, the pointer to the send completion handler (\texttt{shdlr}) is called with the send completion data (\texttt{sinfo}), if \texttt{shdlr} is not a NULL pointer (in C) or \texttt{LAPI_ADDR_NULL} (in FORTRAN). Otherwise, the behavior is identical to that of \texttt{LAPI_Putv}.

\textbf{lapi_rmw_t details}

\textbf{Table 18} shows the correspondence among the parameters of the \texttt{LAPI_Rmw} subroutine, the fields of the C \texttt{lapi_rmw_t} structure and their datatypes, and the equivalent FORTRAN datatypes. The \texttt{lapi_rmw_t} fields are listed in \textbf{Table 16 on page 672} in the order that they occur in the \texttt{lapi_xfer_t} structure.

\begin{table}[h]
\centering
\begin{tabular}{|l|l|l|l|}
\hline
\textbf{lapi_rmw_t field name (C)} & \textbf{lapi_rmw_t field type (C)} & \textbf{Equivalent FORTRAN datatype} & \textbf{Equivalent \texttt{LAPI_Rmw} parameter} \\
\hline
\texttt{Xfer_type} & \texttt{lapi_xfer_type_t} & \texttt{INTEGER(KIND = 4)} & \texttt{LAPI_Xfer} value in FORTRAN: \texttt{LAPI_RMW_XFER} \\
\hline
\texttt{op} & \texttt{Rmw_ops_t} & \texttt{INTEGER(KIND = 4)} & \texttt{op} \\
\hline
\texttt{tgt} & \texttt{uint} & \texttt{INTEGER(KIND = 4)} & \texttt{tgt} \\
\hline
\texttt{size} & \texttt{uint} & \texttt{INTEGER(KIND = 4)} & implicit in C \texttt{LAPI_Xfer} parameter in FORTRAN: \texttt{size} (must be 32 or 64) \\
\hline
\texttt{tgt_var} & \texttt{lapi_long_t} & \texttt{INTEGER(KIND = 8)} & \texttt{tgt_var} \\
\hline
\texttt{in_val} & \texttt{void *} & \texttt{INTEGER(KIND = 4)} (32-bit) \texttt{INTEGER(KIND = 8)} (64-bit) & \texttt{in_val} \\
\hline
\texttt{prev_tgt_val} & \texttt{void *} & \texttt{INTEGER(KIND = 4)} (32-bit) \texttt{INTEGER(KIND = 8)} (64-bit) & \texttt{prev_tgt_val} \\
\hline
\texttt{org_cntr} & \texttt{lapi_cntr_t *} & \texttt{INTEGER(KIND = 4)} (32-bit) \texttt{INTEGER(KIND = 8)} (64-bit) & \texttt{org_cntr} \\
\hline
\texttt{shdlr} & \texttt{scompl_hndlr_t *} & \texttt{INTEGER(KIND = 4)} (32-bit) \texttt{INTEGER(KIND = 8)} (64-bit) & \texttt{none} \texttt{LAPI_Xfer} parameter in FORTRAN: \texttt{shdlr} \\
\hline
\texttt{sinfo} & \texttt{void *} & \texttt{INTEGER(KIND = 4)} (32-bit) \texttt{INTEGER(KIND = 8)} (64-bit) & \texttt{none} \texttt{LAPI_Xfer} parameter in FORTRAN: \texttt{shdlr} \\
\hline
\texttt{none} & \texttt{none} & \texttt{INTEGER(KIND = 4)} & \texttt{LAPI_Xfer} parameter in FORTRAN (32-bit): \texttt{pad} \\
\hline
\end{tabular}
\caption{\texttt{LAPI_Rmw} and \texttt{lapi_rmw_t} equivalents}
\end{table}

When the origin data buffer is free to be used, the pointer to the send completion handler (\texttt{shdlr}) is called with the send completion data (\texttt{sinfo}), if \texttt{shdlr} is not a NULL pointer (in C) or \texttt{LAPI_ADDR_NULL} (in FORTRAN). The \texttt{size} value must be either \texttt{32} or \texttt{64}, indicating whether you want the \texttt{in_val} and \texttt{prev_tgt_val} fields to point to a 32-bit or 64-bit quantity, respectively. Otherwise, the behavior is identical to that of \texttt{LAPI_Rmw}.

\textbf{Parameters}

\textbf{INPUT}

\texttt{hndl} \hspace{1cm} Specifies the LAPI handle.

\texttt{xfer_cmd} \hspace{1cm} Specifies the name and parameters of the data transfer function.
OUTPUT

$ierror$ Specifies a FORTRAN return code. This is always the last parameter.

Return Values

LAPI_SUCCESS Indicates that the function call completed successfully.

LAPI_ERR_DATA_LEN Indicates that the value of $udata\_len$ or $len$ is greater than the value of LAPI constant LAPI_MAX_MSG_SZ.

LAPI_ERR DGSP Indicates that the DGSP that was passed in is NULL (in C) or LAPI_ADDR_NULL (in FORTRAN) or is not a registered DGSP.

LAPI_ERR DGSP_ATOM Indicates that the DGSP has an atom_size that is less than 0 or greater than MAX_ATOM_SIZE.

LAPI_ERR DGSP_BRANCH Indicates that the DGSP attempted a branch that fell outside the code array.

LAPI_ERR DGSP_CTL Indicates that a DGSP control instruction was encountered in a non-valid context (such as a gather-side control or scatter-side control with an atom size of 0 at gather, for example).

LAPI_ERR DGSP_OPCODE Indicates that the DGSP op-code is not valid.

LAPI_ERR DGSP_STACK Indicates that the DGSP has greater GOSUB depth than the allocated stack supports. Stack allocation is specified by the dgsp->depth member.

LAPI_ERR_HDR_HNDLR_NULL Indicates that the $hdr\_hdl$ passed in is NULL (in C) or LAPI_ADDR_NULL (in FORTRAN).

LAPI_ERR HNDL_INVALID Indicates that the $hndl$ passed in is not valid (not initialized or in terminated state).

LAPI_ERR IN_VAL_NULL Indicates that the $in\_val$ pointer is NULL (in C) or LAPI_ADDR_NULL (in FORTRAN).

LAPI_ERR MEMORY EXHAUSTED LAPI is unable to obtain memory from the system.

LAPI_ERR OP_SZ Indicates that the $lapi\_rmw\_t$ size field is not set to 32 or 64.

LAPI_ERR ORG_ADDR NULL Indicates either that the $udata$ parameter passed in is NULL (in C) or LAPI_ADDR_NULL (in FORTRAN) and $udata\_len$ is greater than 0, or that the $org\_addr$ passed in is NULL (in C) or LAPI_ADDR_NULL (in FORTRAN) and $len$ is greater than 0.

Note: if $Xfer\_type = LAPI\_DGSP\_XFER$, the case in which $udata$ is NULL (in C) or LAPI_ADDR_NULL (in FORTRAN) and $udata\_len$ is greater than 0 is valid, so an error is not returned.

LAPI_ERR ORG EXTENT Indicates that the $org\_vec$'s extent (stride * num_vecs) is greater than the value of LAPI constant LAPI_MAX_MSG_SZ.

LAPI_ERR ORG STRIDE Indicates that the $org\_vec$ stride is less than block.

LAPI_ERR ORG VEC ADDR Indicates that the $org\_vec$->$info[j]$ is NULL (in C) or LAPI_ADDR_NULL (in FORTRAN), but its length ($org\_vec$->$len[j]$) is not 0.

LAPI_ERR ORG VEC LEN Indicates that the sum of $org\_vec$->$len$ is greater than the value of LAPI constant LAPI_MAX_MSG_SZ.
LAPI_ERR_ORG_VEC_NULL Indicates that the org_vec value is NULL (in C) or LAPI_ADDR_NULL (in FORTRAN).

LAPI_ERR_ORG_VEC_TYPE Indicates that the org_vec->vec_type is not valid.

LAPI_ERR_RMW_OP Indicates the op is not valid.

LAPI_ERR_STRIDE_ORG_VEC_ADDR_NULL Indicates that the strided vector address org_vec->info[0] is NULL (in C) or LAPI_ADDR_NULL (in FORTRAN).

LAPI_ERR_STRIDE_TGT_VEC_ADDR_NULL Indicates that the strided vector address tgt_vec->info[0] is NULL (in C) or LAPI_ADDR_NULL (in FORTRAN).

LAPI_ERR_TGT Indicates that the tgt passed in is outside the range of tasks defined in the job.

LAPI_ERR_TGT_ADDR_NULL Indicates that ret_addr is NULL (in C) or LAPI_ADDR_NULL (in FORTRAN).

LAPI_ERR_TGT_EXTENT Indicates that tgt_vec's extent (stride * num_vecs) is greater than the value of LAPI constant LAPI_MAX_MSG_SZ.

LAPI_ERR_TGT_PURGED Indicates that the subroutine returned early because LAPI_Purge_totask() was called.

LAPI_ERR_TGT_STRIDE Indicates that the tgt_vec stride is less than block.

LAPI_ERR_TGT_VAR_NULL Indicates that the tgt_var address is NULL (in C) or that the value of tgt_var is LAPI_ADDR_NULL (in FORTRAN).

LAPI_ERR_TGT_VEC_ADDR Indicates that the tgt_vec->info[i] is NULL (in C) or LAPI_ADDR_NULL (in FORTRAN), but its length (tgt_vec->len[i]) is not 0.

LAPI_ERR_TGT_VEC_LEN Indicates that the sum of tgt_vec->len is greater than the value of LAPI constant LAPI_MAX_MSG_SZ.

LAPI_ERR_TGT_VEC_NULL Indicates that tgt_vec is NULL (in C) or LAPI_ADDR_NULL (in FORTRAN).

LAPI_ERR_TGT_VEC_TYPE Indicates that the tgt_vec->vec_type is not valid.

LAPI_ERR_UHDR_LEN Indicates that the uhdr_len value passed in is greater than MAX_UHDR_SZ or is not a multiple of the processor's doubleword size.

LAPI_ERR_UHDR_NULL Indicates that the uhdr passed in is NULL (in C) or LAPI_ADDR_NULL (in FORTRAN), but uhdr_len is not 0.

LAPI_ERR_VEC_LEN_DIFF Indicates that org_vec and tgt_vec have different lengths (len[ ]).

LAPI_ERR_VEC_NUM_DIFF Indicates that org_vec and tgt_vec have different num_vecs.

LAPI_ERR_VEC_TYPE_DIFF Indicates that org_vec and tgt_vec have different vector types (vec_type).

LAPI_ERR_XFER_CMD Indicates that the Xfer_cmd is not valid.

C Examples
1. To run the sample code shown in LAPI_Get using the LAPI_Xfer interface:

   ```
   lapi_xfer_t xfer_struct;
   /* initialize the table buffer for the data addresses */
   ```
/* get remote data buffer addresses */
LAPI_Address_init(hndl,(void *)data_buffer,data_buffer_list);

东

/* retrieve data_len bytes from address data_buffer_list[tgt] on */
/* task tgt. write the data starting at address data_buffer. */
/* tgt_cntr and org_cntr can be NULL. */
xfer_struct.Get.Xfer_type = LAPI_GET_XFER;
xfer_struct.Get.flags = 0;
xfer_struct.Get.tgt = tgt;
xfer_struct.Get.tgt_addr = data_buffer_list[tgt];
xfer_struct.Get.org_addr = data_buffer;
xfer_struct.Get.len = data_len;
xfer_struct.Get.tgt_cntr = tgt_cntr;
xfer_struct.Get.org_cntr = org_cntr;
LAPI_Xfer(hndl, &xfer_struct);

2. To implement the LAPI_STRIDED_VECTOR example from LAPI_Amsendv using the LAPI_Xfer
interface:

{
lapi_xfer_t xfer_struct; /* info for LAPI_Xfer call */
lapi_vec_t vec; /* data for data transfer */

vec->num_vecs = NUM_VECS; /* NUM_VECS = number of vectors to transfer */
vec->vec_type = LAPI_GEN_STRIDED_XFER; /* same as target vector */
vec->info[0] = buffer_address; /* starting address for data copy */
vec->info[1] = block_size; /* bytes of data to copy */
vec->info[2] = stride; /* distance from copy block to copy block */
/* data will be copied as follows: */
/* block_size bytes will be copied from buffer_address */
/* block_size bytes will be copied from buffer_address+stride */
/* block_size bytes will be copied from buffer_address+(2*stride) */
/* block_size bytes will be copied from buffer_address+(3*stride) */

/* block_size bytes will be copied from buffer_address+((NUM_VECS-1)*stride) */

xfer_struct.Amv.Xfer_type = LAPI_AMV_XFER;
xfer_struct.Amv.flags = 0;
xfer_struct.Amv.tgt = tgt;
xfer_struct.Amv.hdr_hdl = hdr_hdl_list[tgt];
xfer_struct.Amv.uhdr_len = uhdr_len; /* user header length */
xfer_struct.Amv.uhdr = uhdr;

/* LAPI_AMV_XFER allows the use of a send completion handler */
/* If non-null, the shdlr function is invoked at the point */
/* the origin counter would increment. Note that both the */
/* org_cntr and shdlr can be used. */
/* The user's shdlr must be of type scompl_hndlr_t. */
/* scompl_hndlr_t is defined in /usr/include/lapi.h */
xfer_struct.shdlr = shdlr;
Use sinfo to pass user-defined data into the send completion handler, if desired.

```c
xfer_struct.sinfo = sinfo; /* send completion data */
```

```c
xfer_struct.org_vec = vec;
xfer_struct.tgt_cntr = tgt_cntr;
xfer_struct.org_cntr = org_cntr;
xfer_struct.cmpl_cntr = cmpl_cntr;
```

```c
LAPI_Xfer(hndl, &xfer_struct);
```

See the `LAPI_Amsendv` subroutine for more information about the header handler definition.

**Location**
/usr/lib/liblapi_r.a

**Related Information**
Books: RSCT for AIX 5L: LAPI Programming Guide for information about bulk message transfer

Subroutines: `LAPI_Amsend`, `LAPI_Amsendv`, `LAPI_Get`, `LAPI_Getv`, `LAPI_Put`, `LAPI_Putv`, `LAPI_Rmw`

---

**Layout Library Reference**

**layout_object_create Subroutine**

**Purpose**
Initializes a layout context.

**Library**
Layout Library (`libi18n.a`)

**Syntax**

```c
#include <sys/lc_layout.h>

int layout_object_create (locale_name, layout_object)
const char *locale_name;
LayoutObject *layout_object;
```

**Description**

The `layout_object_create` subroutine creates the `LayoutObject` structure associated with the locale specified by the `locale_name` parameter. The `LayoutObject` structure is a symbolic link containing all the data and methods necessary to perform the layout operations on context dependent and bidirectional characters of the locale specified.

When the `layout_object_create` subroutine completes without errors, the `layout_object` parameter points to a valid `LayoutObject` structure that can be used by other BIDI subroutines. The returned `LayoutObject` structure is initialized to an initial state that defines the behavior of the BIDI subroutines. This initial state is locale dependent and is described by the layout values returned by the `layout_object_getvalue` subroutine. You can change the layout values of the `LayoutObject` structure using the `layout_object_setvalue` subroutine. Any state maintained by the `LayoutObject` structure is independent of the current global locale set with the `setlocale` subroutine.
Note: If you are developing internationalized applications that may support multibyte locales, please see Use of the libcun Package in AIX 5L Version 5.3 General Programming Concepts: Writing and Debugging Programs

Parameters

locale_name
Specifies a locale. It is recommended that you use the LC_CTYPE category by calling the setlocale (LC_CTYPE, NULL) subroutine.

layout_object
Points to a valid LayoutObject structure that can be used by other layout subroutines. This parameter is used only when the layout_object_create subroutine completes without errors.

The layout_object parameter is not set and a non-zero value is returned if a valid LayoutObject structure cannot be created.

Return Values

Upon successful completion, the layout_object_create subroutine returns a value of 0. The layout_object parameter points to a valid handle.

Error Codes

If the layout_object_create subroutine fails, it returns the following error codes:

LAYOUT EINVAL
The locale specified by the locale_name parameter is not available.

LAYOUT EMFILE
The OPEN_MAX value of file descriptors are currently open in the calling process.

LAYOUT ENOMEM
Insufficient storage space is available.

Related Information

The layout_object_editshape or wcslayout_object_editshape Subroutine, layout_object_free Subroutine, layout_object_getvalue Subroutine, layout_object_setvalue Subroutine, layout_object_shapeboxchars Subroutine, layout_object_transform or wcslayout_object_transform Subroutine.


layout_object_editshape or wcslayout_object_editshape Subroutine

Purpose
Edits the shape of the context text.

Library
Layout library (libi18n.a)
Syntax

```c
#include <sys/lc_layout.h>

int layout_editshape (LayoutObject layout_object, EditType index, InpBuf, Inpsize, OutBuf, OutSize)

LayoutObject layout_object;
BooleanValue EditType;
size_t *index;
const char *InpBuf;
size_t *Inpsize;
void *OutBuf;
size_t *OutSize;

int wcslayout_object_editshape(layout_object, EditType, index, InpBuf, Inpsize, OutBuf, OutSize)

LayoutObject layout_object;
BooleanValue EditType;
size_t *index;
const wchar t *InpBuf;
size_t *Inpsize;
void *OutBuf;
size_t *OutSize;
```

Description
The `layout_object_editshape` and `wcslayout_object_editshape` subroutines provide the shapes of the context text. The shapes are defined by the code element specified by the `index` parameter and any surrounding code elements specified by the ShapeContextSize layout value of the `LayoutObject` structure. The `layout_object` parameter specifies this `LayoutObject` structure.

Use the `layout_object_editshape` subroutine when editing code elements of one byte. Use the `wcslayout_object_editshape` subroutine when editing single code elements of multibytes. These subroutines do not affect any state maintained by the `layout_object_transform` or `wcslayout_object_transform` subroutine.

Note: If you are developing internationalized applications that may support multibyte locales, please see Use of the libc package in AIX 5L Version 5.3 General Programming Concepts: Writing and Debugging Programs

Parameters

- `layout_object` Specifies the `LayoutObject` structure created by the `layout_object_create` subroutine.
**EditType**

Specifies the type of edit shaping. When the `EditType` parameter stipulates the `EditInput` field, the subroutine reads the current code element defined by the `index` parameter and any preceding code elements defined by `ShapeContextSize` layout value of the `LayoutObject` structure. When the `EditType` parameter stipulates the `EditReplace` field, the subroutine reads the current code element defined by the `index` parameter and any surrounding code elements defined by `ShapeContextSize` layout value of the `LayoutObject` structure.

**Note:** The editing direction defined by the `Orientation` and `TEXT_VISUAL` of the `TypeOfText` layout values of the `LayoutObject` structure determines which code elements are preceding and succeeding.

When the `ActiveShapeEditing` layout value of the `LayoutObject` structure is set to True, the `LayoutObject` structure maintains the state of the `EditInput` field that may affect subsequent calls to these subroutines with the `EditInput` field defined by the `EditType` parameter. The state of the `EditInput` field of `LayoutObject` structure is not affected when the `EditType` parameter is set to the `EditReplace` field. To reset the state of the `EditInput` field to its initial state, call these subroutines with the `InpBuf` parameter set to NULL. The state of the `EditInput` field is not affected if errors occur within the subroutines.

**index**

Specifies an offset (in bytes) to the start of a code element in the `InpBuf` parameter on input. The `InpBuf` parameter provides the base text to be edited. In addition, the context of the surrounding code elements is considered where the minimum set of code elements needed for the specific context dependent script(s) is identified by the `ShapeContextSize` layout value.

If the set of surrounding code elements defined by the `index`, `InpBuf`, and `InpSize` parameters is less than the size of front and back of the `ShapeContextSize` layout value, these subroutines assume there is no additional context available. The caller must provide the minimum context if it is available. The `index` parameter is in units associated with the type of the `InpBuf` parameter.

On return, the `index` parameter is modified to indicate the offset to the first code element of the `InpBuf` parameter that required shaping. The number of code elements that required shaping is indicated on return by the `InpSize` parameter.

**InpBuf**

Specifies the source to be processed. A Null value with the `EditInput` field in the `EditType` parameter indicates a request to reset the state of the `EditInput` field to its initial state.

Any portion of the `InpBuf` parameter indicates the necessity for redrawing or shaping.

**InpSize**

Specifies the number of code elements to be processed in units on input. These units are associated with the types for these subroutines. A value of -1 indicates that the input is delimited by a Null code element.

On return, the value is modified to the actual number of code elements needed by the `InpBuf` parameter. A value of 0 when the value of the `EditType` parameter is the `EditInput` field indicates that the state of the `EditInput` field is reset to its initial state. If the `OutBuf` parameter is not NULL, the respective shaped code elements are written into the `OutBuf` parameter.

**OutBuf**

Contains the shaped output text. You can specify this parameter as a Null pointer to indicate that no transformed text is required. If Null, the subroutines return the `index` and `InpSize` parameters, which specify the amount of text required, to be redrawn.

The encoding of the `OutBuf` parameter depends on the `ShapeCharset` layout value defined in `layout_object` parameter. If the `ActiveShapeEditing` layout value is set to False, the encoding of the `OutBuf` parameter is to be the same as the code set of the locale associated with the specified `LayoutObject` structure.
Specifies the size of the output buffer on input in number of bytes. Only the code elements required to be shaped are written into the OutBuf parameter.

The output buffer should be large enough to contain the shaped result; otherwise, only partial shaping is performed. If the ActiveShapeEditing layout value is set to True, the OutBuf parameter should be allocated to contain at least the number of code elements in the InpBuf parameter multiplied by the value of the ShapeCharsetSize layout value.

On return, the OutSize parameter is modified to the actual number of bytes placed in the output buffer.

When the OutSize parameter is specified as 0, the subroutines calculate the size of an output buffer large enough to contain the transformed text from the input buffer. The result will be returned in this field. The content of the buffers specifies by the InpBuf and OutBuf parameters, and the value of the InpSize parameter, remain unchanged.

Return Values
Upon successful completion, these subroutines return a value of 0. The index and InpSize parameters return the minimum set of code elements required to be redrawn.

Error Codes
If these subroutines fail, they return the following error codes:

- **LAYOUT_EILSEQ**: Shaping stopped due to an input code element that cannot be shaped. The index parameter indicates the code element that caused the error. This code element is either a valid code element that cannot be shaped according to the ShapeCharset layout value or an invalid code element not defined by the code set defined in the LayoutObject structure. Use the mbtowc or wctomb subroutine in the same locale as the LayoutObject structure to determine if the code element is valid.

- **LAYOUT_E2BIG**: The output buffer is too small and the source text was not processed. The index and InpSize parameters are not guaranteed on return.

- **LAYOUT EINVAL**: Shaping stopped due to an incomplete code element or shift sequence at the end of input buffer. The InpSize parameter indicates the number of code elements successfully transformed.
  
  **Note**: You can use this error code to determine the code element causing the error.

- **LAYOUT ERANGE**: Either the index parameter is outside the range as defined by the InpSize parameter, more than 15 embedding levels are in the source text, or the InpBuf parameter contains unbalanced Directional Format (Push/Pop).

Related Information

**Library**

Layout Library (libi18n.a)

**Syntax**

```c
#include <sys/lc_layout.h>

int layout_object_getvalue( LayoutObject layout_object, LayoutValues values, int *index);
```

**Description**

The `layout_object_getvalue` subroutine queries the current setting of layout values within the `LayoutObject` structure. The `layout_object` parameter specifies the `LayoutObject` structure created by the `layout_object_create` subroutine.

The name field of the LayoutValues structure contains the name of the layout value to be queried. The value field is a pointer to where the layout value is stored. The values are queried from the `LayoutObject` structure and represent its current state.

For example, if the layout value to be queried is of type T, the value parameter must be of type T*. If T itself is a pointer, the `layout_object_getvalue` subroutine allocates space to store the actual data. The caller must free this data by calling the `free(T)` subroutine with the returned pointer.

When setting the value field, an extra level of indirection is present that is not present using the `layout_object_setvalue` parameter. When you set a layout value of type T, the value field contains T. However, when querying the same layout value, the value field contains &T.

**Note:** If you are developing internationalized applications that may support multibyte locales, please see [Use of the libcur Package](#) in AIX 5L Version 5.3 General Programming Concepts: Writing and Debugging Programs

**Parameters**

- `layout_object`: Specifies the `LayoutObject` structure created by the `layout_object_create` subroutine.
- `values`: Specifies an array of layout values of type LayoutValueRec that are to be queried in the `LayoutObject` structure. The end of the array is indicated by `name=0`.
- `index`: Specifies a layout value to be queried. If the value cannot be queried, the `index` parameter causing the error is returned and the subroutine returns a non-zero value.

**Return Values**

Upon successful completion, the `layout_object_getvalue` subroutine returns a value of 0. All layout values were successfully queried.

**Error Codes**

If the `layout_object_getvalue` subroutine fails, it returns the following values:

- LAYOUT_EINVAL: The layout value specified by the `index` parameter is unknown or the `layout_object` parameter is invalid.
- LAYOUT_EMOMEM: Insufficient storage space is available.
### Examples

The following example queries whether the locale is bidirectional and gets the values of the in and out orientations.

```c
#include <sys/lc_layout.h>
#include <locale.h>

main()
{
    LayoutObject plh;
    int RC=0;
    LayoutValues layout;
    LayoutTextDescriptor Descr;
    int index;

    RC=layout_object_create(setlocale(LC_CTYPE,""),&plh); /* create object */
    if (RC) {printf("Create error !\n"); exit(0);}

    layout=malloc(3*sizeof(LayoutValueRec)); /* allocate layout array */
    layout[0].name=ActiveBidirection; /* set name */
    layout[1].name=Orientation; /* set name */
    layout[1].value=(caddr_t)&Descr; /* send address of memory to be allocated by function */
    layout[2].name=0; /* indicate end of array */
    RC=layout_object_getvalue(plh,layout,&index);
    if (RC) {printf("Getvalue error at %d \n",index); exit(0);}
    printf("ActiveBidirection = %d \n",*(layout[0].value)); /* print output*/
    printf("Orientation in = %x out = %x \n", Descr->in, Descr->out);

    free(layout); /* free layout array */
    free(Descr); /* free memory allocated by function */
    RC=layout_object_free(plh); /* free layout object */
    if (RC) printf("Free error !\n");
}
```

### Related Information

The "layout_object_create Subroutine" on page 678, "layout_object_editshape or wcslayout_object_editshape Subroutine" on page 679, "layout_object_free Subroutine" on page 690, "layout_object_setvalue Subroutine," "layout_object_shapeboxchars Subroutine" on page 686, and "layout_object_transform or wcslayout_object_transform Subroutine" on page 687.


---

### layout_object_setvalue Subroutine

#### Purpose

Sets the layout values of a `LayoutObject` structure.

#### Library

Layout Library (`libi18n.a`)

#### Syntax

```c
#include <sys/lc_layout.h>
```
int layout_object_setvalue(
    LayoutObject layout_object,
    LayoutValues values,
    int *index);

Description
The `layout_object_setvalue` subroutine changes the current layout values of the `LayoutObject` structure. The `layout_object` parameter specifies the `LayoutObject` structure created by the `layout_object_create` subroutine. The values are written into the `LayoutObject` structure and may affect the behavior of subsequent layout functions.

Note: Some layout values do alter internal states maintained by a `LayoutObject` structure.

The name field of the LayoutValueRec structure contains the name of the layout value to be set. The value field contains the actual value to be set. The value field is large enough to support all types of layout values. For more information on layout value types, see "Layout Values for the Layout Library" in AIX 5L Version 5.3 General Programming Concepts: Writing and Debugging Programs.

Note: If you are developing internationalized applications that may support multibyte locales, please see Use of the libcur Package in AIX 5L Version 5.3 General Programming Concepts: Writing and Debugging Programs

Parameters
- `layout_object` Specifies the `LayoutObject` structure returned by the `layout_object_create` subroutine.
- `values` Specifies an array of layout values of the type LayoutValueRec that this subroutine sets. The end of the array is indicated by name=0.
- `index` Specifies a layout value to be queried. If the value cannot be queried, the index parameter causing the error is returned and the subroutine returns a non-zero value. If an error is generated, a subset of the values may have been previously set.

Return Values
Upon successful completion, the `layout_object_setvalue` subroutine returns a value of 0. All layout values were successfully set.

Error Codes
If the `layout_object_setvalue` subroutine fails, it returns the following values:

- `LAYOUT_EINVAL` The layout value specified by the `index` parameter is unknown, its value is invalid, or the `layout_object` parameter is invalid.
- `LAYOUT_EMFILE` The (OPEN_MAX) file descriptors are currently open in the calling process.
- `LAYOUT_ENOMEM` Insufficient storage space is available.

Examples
The following example sets the TypeofText value to Implicit and the out value to Visual.

```c
#include <sys/lc_layout.h>
#include <locale.h>

main()
{
    LayoutObject plh;
    int RC=0;
    LayoutValues layout;
    LayoutTextDescriptor Descr;
```
int index;

RC=layout_object_create(setlocale(LC_CTYPE,""),&plh); /* create object */
if (RC) {printf("Create error !!\n"); exit(0);}

layout=malloc(2*sizeof(LayoutValueRec)); /*allocate layout array*/
Descr=malloc(sizeof(LayoutTextDescriptorRec)); /* allocate text descriptor */
layout[0].name=TypeOfText; /* set name */
layout[0].value=(caddr_t)Descr; /* set value */
layout[1].name=0; /* indicate end of array */

Descr->in=TEXT_IMPLICIT;
Descr->out=TEXT_VISUAL; RC=layout_object_setvalue(plh,layout,&index);
if (RC) printf("SetValue error at %d!!\n",index); /* check return code */
free(layout); /* free allocated memory */
free (Descr);
RC=layout_object_free(plh); /* free layout object */
if (RC) printf("Free error !!\n");

Related Information
The "layout_object_create Subroutine" on page 678, "layout_object_editshape or
wcslayout_object_editshape Subroutine" on page 679, "layout_object_free Subroutine" on page 690,
"layout_object_getvalue Subroutine" on page 682, "layout_object_shapeboxchars Subroutine," and
"layout_object_transform or wcslayout_object_transform Subroutine" on page 687.

Bidirectionality and Character Shaping and National Language Support Overview in AIX 5L Version 5.3

layout_object_shapeboxchars Subroutine

Purpose
Shapes box characters.

Library
Layout Library (libi18n.a)

Syntax
#include <sys/lc_layout.h>

int layout_object_shapeboxchars( [LayoutObject layout_object; const char *InpBuf; const size_t InpSize; char *OutBuf]

Description
The layout_object_shapeboxchars subroutine shapes box characters into the VT100 box character set.

Note: If you are developing internationalized applications that may support multibyte locales, please see
Use of the libcur Package in AIX 5L Version 5.3 General Programming Concepts: Writing and
Debugging Programs
Parameters

layout_object
Specifies the LayoutObject structure created by the layout_object_create subroutine.

InpBuf
Specifies the source text to be processed.

InpSize
Specifies the number of code elements to be processed.

OutBuf
Contains the shaped output text.

Return Values

Upon successful completion, this subroutine returns a value of 0.

Error Codes

If this subroutine fails, it returns the following values:

LAYOUT_EILSEQ
Shaping stopped due to an input code element that cannot be mapped into the VT100 box character set.

LAYOUT_ENVAL
Shaping stopped due to an incomplete code element or shift sequence at the end of the input buffer.

Related Information


layout_object_transform or wcslayout_object_transform Subroutine

Purpose

Transforms text according to the current layout values of a LayoutObject structure.

Library

Layout Library (libi18n.a)

Syntax

#include <sys/lc_layout.h>
int layout_object_transform (layout_object, InpBuf, InpSize, OutBuf, OutSize, InpToOut, OutToInp, BidiLvl)

int wcslayout_object_transform (layout_object, InpBuf, InpSize, OutBuf, OutSize, InpToOut, OutToInp, BidiLvl)
Description

The `layout_object_transform` and `wcslayout_object_transform` subroutines transform the text specified by the `InpBuf` parameter according to the current layout values in the `LayoutObject` structure. Any layout value whose type is `LayoutTextDescriptor` describes the attributes within the `InpBuf` and `OutBuf` parameters. If the attributes are the same as the `InpBuf` and `OutBuf` parameters themselves, a null transformation is done with respect to that specific layout value.

The output of these subroutines may be one or more of the following results depending on the setting of the respective parameters:

- **OutBuf, OutSize**: Any transformed data is stored in the `OutBuf` parameter.
- **InpToOut**: A cross reference from each code element of the `InpBuf` parameter to the transformed data.
- **OutToInp**: A cross reference to each code element of the `InpBuf` parameter from the transformed data.
- **BidiLvl**: A weighted value that represents the directional level of each code element of the `InpBuf` parameter. The level is dependent on the internal directional algorithm of the `LayoutObject` structure.

You can specify each of these output parameters as Null to indicate that no output is needed for the specific parameter. However, you should set at least one of these parameters to a nonNULL value to perform any significant work.

To perform shaping of a text string without reordering of code elements, set the `TypeOfText` layout value to `TEXT_VISUAL` and the in and out values of the `Orientation` layout value alike. These layout values are in the `LayoutObject` structure.

**Note:** If you are developing internationalized applications that may support multibyte locales, please see [Use of the libcur Package](https://publib.boulder.ibm.com/infocenter/aix/ic/length/) in AIX 5L Version 5.3 General Programming Concepts: Writing and Debugging Programs.

Parameters

- **layout_object**: Specifies the `LayoutObject` structure created by the `layout_object_create` subroutine.
- **InpBuf**: Specifies the source text to be processed. This parameter cannot be null.
- **InpSize**: Specifies the units of code elements processed associated with the bytes for the `layout_object_transform` and `wcslayout_object_transform` subroutines. A value of -1 indicates that the input is delimited by a null code element. On return, the value is modified to the actual number of code elements processed in the `InBuf` parameter. However, if the value in the `OutSize` parameter is zero, the value of the `InpSize` parameter is not changed.
- **OutBuf**: Contains the transformed data. You can specify this parameter as a null pointer to indicate that no transformed data is required.

The encoding of the `OutBuf` parameter depends on the `ShapeCharset` layout value defined in the `LayoutObject` structure. If the `ActiveShapeEditing` layout value is set to True, the encoding of the `OutBuf` parameter is the same as the code set of the locale associated with the `LayoutObject` structure.
OutSize Specifies the size of the output buffer in number of bytes. The output buffer should be large enough to contain the transformed result; otherwise, only a partial transformation is performed. If the ActiveShapeEditing layout value is set to True, the OutBuf parameter should be allocated to contain at least the number of code elements multiplied by the ShapeCharsetSize layout value.

On return, the OutSize parameter is modified to the actual number of bytes placed in this parameter.

When you specify the OutSize parameter as 0, the subroutine calculates the size of an output buffer to be large enough to contain the transformed text. The result is returned in this field. The content of the buffers specified by the InpBuf and OutBuf parameters, and a value of the InpSize parameter remains unchanged.

InpToOut Represents an array of values with the same number of code elements as the InpBuf parameter if InpToOut parameter is not a null pointer.

On output, the nth value in InpToOut parameter corresponds to the nth code element in InpBuf parameter. This value is the index in OutBuf parameter which identifies the transformed ShapeCharset element of the nth code element in InpBuf parameter. You can specify the InpToOut parameter as null if no index array from the InpBuf to OutBuf parameters is desired.

OutToInp Represents an array of values with the same number of code elements as contained in the OutBuf parameter if the OutToInp parameter is not a null pointer.

On output, the nth value in the OutToInp parameter corresponds to the nth ShapeCharset element in the OutBuf parameter. This value is the index in the InpBuf parameter which identifies the original code element of the nth ShapeCharset element in the OutBuf parameter. You can specify the OutToInp parameter as NULL if no index array from the OutBuf to InpBuf parameters is desired.

BidiLvl Represents an array of values with the same number of elements as the source text if the BidiLvl parameter is not a null pointer. The nth value in the BidiLvl parameter corresponds to the nth code element in the InpBuf parameter. This value is the level of this code element as determined by the bidirectional algorithm. You can specify the BidiLvl parameter as null if a levels array is not desired.

Return Values
Upon successful completion, these subroutines return a value of 0.

Error Codes
If these subroutines fail, they return the following values:

LAYOUT_EILSEQ Transformation stopped due to an input code element that cannot be shaped or is invalid. The InpSize parameter indicates the number of the code element successfully transformed.

Note: You can use this error code to determine the code element causing the error.

This code element is either a valid code element but cannot be shaped into the ShapeCharset layout value or is an invalid code element not defined by the code set of the locale of the LayoutObject structure. You can use the mbtowc and wctomb subroutines to determine if the code element is valid when used in the same locale as the LayoutObject structure.

LAYOUT_E2BIG The output buffer is full and the source text is not entirely processed.
Transformation stopped due to an incomplete code element or shift sequence at the end of the input buffer. The \textit{InpSize} parameter indicates the number of the code elements successfully transformed. 

\textbf{Note}: You can use this error code to determine the code element causing the error.

\textbf{LAYOUT_ERANGE}

More than 15 embedding levels are in the source text or the \textit{InpBuf} parameter contains unbalanced Directional Format (Push/Pop).

When the size of \textit{OutBuf} parameter is not large enough to contain the entire transformed text, the input text state at the end of the \textbf{LAYOUT_E2BIG} error code is returned. To resume the transformation on the remaining text, the application calls the \textit{layout_object_transform} subroutine with the same \textit{LayoutObject} structure, the same \textit{InpBuf} parameter, and \textit{InpSize} parameter set to 0.

\section*{Examples}

The following is an example of transformation of both directional re-ordering and shaping.

\textbf{Notes}:

1. Uppercase represent left-to-right characters; lowercase represent right-to-left characters.
2. \textit{xyz} represent the shapes of \textit{cde}.

\begin{verbatim}
Position: 0123456789
InpBuf: AB cde 12Z

Position: 0123456789
OutBuf: AB 12 zyxZ

Position: 0123456789
ToTarget: 0128765349

Position: 0123456789
ToSource: 0127865439

Position: 0123456789
BidiLevel: 0001111220
\end{verbatim}

\section*{Related Information}


\subsection*{layout_object_free Subroutine}

\textbf{Purpose}

Frees a \textit{LayoutObject} structure.

\textbf{Library}

Layout library (\textit{libi18n.a})

\textbf{Syntax}

\verb+#include <sys/lc_layout.h>+
int layout_object_free(layout_object)
LayoutObject layout_object;

Description
The layout_object_free subroutine releases all the resources of the LayoutObject structure created by the layout_object_create subroutine. The layout_object parameter specifies this LayoutObject structure.

Note: If you are developing internationalized applications that may support multibyte locales, please see Use of the libcur Package in AIX 5L Version 5.3 General Programming Concepts: Writing and Debugging Programs

Parameters
layout_object Specifies a LayoutObject structure returned by the layout_object_create subroutine.

Return Values
Upon successful completion, the layout_object_free subroutine returns a value of 0. All resources associated with the layout_object parameter are successfully deallocated.

Error Codes
If the layout_object_free subroutine fails, it returns the following error code:

LAYOUT_EFAULT Errors occurred while processing the request.

Related Information


ldahread Subroutine

Purpose
Reads the archive header of a member of an archive file.

Library
Object File Access Routine Library (libld.a)

Syntax
#include <stdio.h>
#include <ar.h>
#include <ldfcn.h>

int ldahread(LDFILE *ldPointer, ARCHDR *ArchiveHeader)
LDFILE *ldPointer;
ARCHDR *ArchiveHeader;
Description
If the `TYPE(ldPointer)` macro from the `ldfcn.h` file is the archive file magic number, the `ldahread` subroutine reads the archive header of the extended common object file currently associated with the `ldPointer` parameter into the area of memory beginning at the `ArchiveHeader` parameter.

Parameters
- `ldPointer` Points to the `LDFILE` structure that was returned as the result of a successful call to `ldopen` or `ldaopen`.
- `ArchiveHeader` Points to a `ARCHDR` structure.

Return Values
The `ldahread` subroutine returns a SUCCESS or FAILURE value.

Error Codes
The `ldahread` routine fails if the `TYPE(ldPointer)` macro does not represent an archive file, or if it cannot read the archive header.

Related Information
The `ldfhread` subroutine, `ldgetname` subroutine, `ldread`, `ldlinit`, or `ldlitem` subroutine, `ldshread` or `ldnshread` subroutine, `ldtbread` subroutine.

ldclose or ldaclose Subroutine

Purpose
Closes a common object file.

Library
Object File Access Routine Library (`libld.a`)

Syntax
```c
#include <stdio.h>
#include <ldfcn.h>

int ldclose(ldPointer)
LDFILE *ldPointer;

int ldaclose(ldPointer)
LDFILE *ldPointer;
```

Description
The `ldopen` and `ldclose` subroutines provide uniform access to both simple object files and object files that are members of archive files. Thus, an archive of common object files can be processed as if it were a series of simple common object files.
If the \texttt{ldfcn.h} file \texttt{TYPE(ldPointer)} macro is the magic number of an archive file, and if there are any more files in the archive, the \texttt{ldclose} subroutine reinitializes the \texttt{ldfcn.h} file \texttt{OFFSET(ldPointer)} macro to the file address of the next archive member and returns a failure value. The \texttt{ldfile} structure is prepared for a subsequent \texttt{ldopen}.

If the \texttt{TYPE(ldPointer)} macro does not represent an archive file, the \texttt{ldclose} subroutine closes the file and frees the memory allocated to the \texttt{ldfile} structure associated with \texttt{ldPointer}.

The \texttt{ldaclose} subroutine closes the file and frees the memory allocated to the \texttt{ldfile} structure associated with the \texttt{ldPointer} parameter regardless of the value of the \texttt{TYPE(ldPointer)} macro.

\textbf{Parameters}

\texttt{ldPointer} \hspace{1em} Pointer to the \texttt{LDFILE} structure that was returned as the result of a successful call to \texttt{ldopen} or \texttt{ldaopen}.

\textbf{Return Values}

The \texttt{ldclose} subroutine returns a \texttt{SUCCESS} or \texttt{FAILURE} value.

The \texttt{ldaclose} subroutine always returns a \texttt{SUCCESS} value and is often used in conjunction with the \texttt{ldaopen} subroutine.

\textbf{Error Codes}

The \texttt{ldclose} subroutine returns a failure value if there are more files to archive.

\textbf{Related Information}

The \texttt{ldaopen} or \texttt{ldopen} ("Idopen or ldaopen Subroutine" on page 701) subroutine.

\textbf{ldexp, ldexpf, or ldexpl Subroutine}

\textbf{Purpose}

Loads exponent of a floating-point number.

\textbf{Syntax}

```
#include <math.h>
float ldexpf (x exp)
float x;
int exp;

long double ldexpl (x, exp)
long double x;
int exp;

double ldexp (x, exp)
double x;
int exp;
```

\textbf{Description}

The \texttt{ldexpf}, \texttt{ldexpl}, and \texttt{ldexp} subroutines compute the quantity \(x \times 2^{exp}\).
An application wishing to check for error situations should set the `errno` global variable to zero and call `feclearexcept(FE_ALL_EXCEPT)` before calling these functions. Upon return, if `errno` is nonzero or `fetestexcept(FE_INVALID | FE_DIVBYZERO | FE_OVERFLOW | FE_UNDERFLOW)` is nonzero, an error has occurred.

**Parameters**

- **x** Specifies the value to be computed.
- **exp** Specifies the exponent of 2.

**Return Values**

Upon successful completion, the `ldexpf`, `ldexpl`, and `ldexp` subroutines return `x` multiplied by 2, raised to the power `exp`.

If the `ldexpf`, `ldexpl`, or `ldexp` subroutines would cause overflow, a range error occurs and the `ldexpf`, `ldexpl`, and `ldexp` subroutines return ±HUGE_VALF, ±HUGE_VALL, and ±HUGE_VAL (according to the sign of `x`), respectively.

If the correct value would cause underflow, and is not representable, a range error may occur, and 0.0 is returned.

If `x` is NaN, a NaN is returned.

If `x` is ±0 or Inf, `x` is returned.

If `exp` is 0, `x` is returned.

If the correct value would cause underflow, and is representable, a range error may occur and the correct value is returned.

**Error Codes**

If the result of the `ldexp` or `ldexpl` subroutine overflows, then +/- HUGE_VAL is returned, and the global variable `errno` is set to `ERANGE`.

If the result of the `ldexp` or `ldexpl` subroutine underflows, 0 is returned, and the `errno` global variable is set to a `ERANGE` value.

**Related Information**

- “feclearexcept Subroutine” on page 262, “fetestexcept Subroutine” on page 270, and “class, _class, finite, isnan, or unordered Subroutines” on page 167
- `math.h` in AIX 5L Version 5.3 Files Reference.

---

### ldfhread Subroutine

**Purpose**

Reads the file header of an XCOFF file.

**Library**

Object File Access Routine Library (`libld.a`)
Syntax
#include <stdio.h>
#include <ldfcn.h>

int ldfhread ([ldpointer], FileHeader)
LDFILE *ldpointer;
void *fileheader;

Description
The ldfhread subroutine reads the file header of the object file currently associated with the ldpointer parameter into the area of memory beginning at the FileHeader parameter. For AIX 4.3.2 and above, it is the responsibility of the calling routine to provide a pointer to a buffer large enough to contain the file header of the associated object file. Since the ldopen subroutine provides magic number information (via the HEADER(ldpointer).f_magic macro), the calling application can always determine whether the FileHeader pointer should refer to a 32-bit FILHDR or 64-bit FILHDR_64 structure.

Parameters
ldpointer Points to the LDFILE structure that was returned as the result of a successful call to ldopen or ldadopen subroutine.
FileHeader Points to a buffer large enough to accommodate a FILHDR structure, according to the object mode of the file being read.

Return Values
The ldfhread subroutine returns Success or Failure.

Error Codes
The ldfhread subroutine fails if it cannot read the file header.

Note: In most cases, the use of ldfhread can be avoided by using the HEADER (ldpointer) macro defined in the ldfcn.h file. The information in any field or fieldname of the header file may be accessed using the header (ldpointer) fieldname macro.

Examples
The following is an example of code that opens an object file, determines its mode, and uses the ldfhread subroutine to acquire the file header. This code would be compiled with both _XCOFF32_ and _XCOFF64_ defined:
#define _XCOFF32_
#define _XCOFF64_
#include <ldfcn.h>

/* for each FileName to be processed */
if ( (ldpointer = ldopen(fileName, ldpointer)) != NULL)
{
    FILHDR FileHead32;
    FILHDR_64 FileHead64;
    void *fileheader;

    if ( HEADER(ldpointer).f_magic == U802TOCMAGIC )
        FileHeader = &FileHead32;
    else if ( HEADER(ldpointer).f_magic == U803XT0CMAGIC )
        FileHeader = &FileHead64;
    else
FileHeader = NULL;

if ( FileHeader && ldfhread( ldPointer, FileHeader ) == SUCCESS) }
{
    /* ...successfully read header... */
    /* ...process according to magic number... */
}

Related Information
The ldahread ("ldahread Subroutine" on page 691) subroutine, ldgetname ("ldgetname Subroutine") subroutine, ldread, ldlinit, or ldlitem ("ldread, ldlinit, or ldlitem Subroutine" on page 698) subroutine, ldopen ("ldopen or ldaopen Subroutine" on page 701) subroutine, ldshread or ldnshread ("ldshread or ldnshread Subroutine" on page 704) subroutine, ldtbread ("ldtbread Subroutine" on page 708) subroutine.

ldgetname Subroutine

Purpose
Retrieves symbol name for common object file symbol table entry.

Library
Object File Access Routine Library (libld.a)

Syntax
#include <stdio.h>
#include <ldfcn.h>

char *ldgetname (ldPointer, Symbol)
LDFILE *ldPointer;
void *Symbol;

Description
The ldgetname subroutine returns a pointer to the name associated with Symbol as a string. The string is in a static buffer local to the ldgetname subroutine that is overwritten by each call to the ldgetname subroutine and must therefore be copied by the caller if the name is to be saved.

The common object file format handles arbitrary length symbol names with the addition of a string table. The ldgetname subroutine returns the symbol name associated with a symbol table entry for an XCOFF-format object file.

The calling routine to provide a pointer to a buffer large enough to contain a symbol table entry for the associated object file. Since the ldopen subroutine provides magic number information (via the HEADER(ldPointer),f_magic macro), the calling application can always determine whether the Symbol pointer should refer to a 32-bit SYMENT or 64-bit SYMENT_64 structure.

The maximum length of a symbol name is BUFSIZ, defined in the stdio.h file.
Parameters

**ldPointer**
Points to an LDFILE structure that was returned as the result of a successful call to the `ldopen` or `ldaopen` subroutine.

**Symbol**
Points to an initialized 32-bit or 64-bit SYMENT structure.

Error Codes

The ldgetname subroutine returns a null value (defined in the stdio.h file) for a COFF-format object file if the name cannot be retrieved. This situation can occur if one of the following is true:

- The string table cannot be found.
- The string table appears invalid (for example, if an auxiliary entry is handed to the ldgetname subroutine wherein the name offset lies outside the boundaries of the string table).
- The name’s offset into the string table is past the end of the string table.

Typically, the ldgetname subroutine is called immediately after a successful call to the ldtbread subroutine to retrieve the name associated with the symbol table entry filled by the ldtbread subroutine.

Examples

The following is an example of code that determines the object file type before making a call to the ldtbread and ldgetname subroutines.

```c
#define ___XCOFF32___
#define ___XCOFF64___

#include <ldfcn.h>

SYMENT  Symbol32;
SYMENT_64 Symbol64;
void    *Symbol;

if ( HEADER(ldPointer).f_magic == U802TOCMAGIC )
    Symbol = &Symbol32;
else if ( HEADER(ldPointer).f_magic == U64_TOCMAGIC )
    Symbol = &Symbol64;
else
    Symbol = NULL;

if ( (Symbol) )
    /* for each symbol in the symbol table */
    for ( symnum = 0 ; symnum < HEADER(ldPointer).f_nsyms ; symnum++ )
    {
        if ( ldtbread(ldPointer,symnum,Symbol) == SUCCESS )
        {
            char *name = ldgetname(ldPointer,Symbol)
            if ( name )
            {
                /* Got the name... */
                
            }
            /* Increment symnum by the number of auxiliary entries */
            if ( HEADER(ldPointer).f_magic == U802TOCMAGIC )
                symnum += Symbol32.n_numaux;
            else if ( HEADER(ldPointer).f_magic == U64_TOCMAGIC )
                symnum += Symbol64.n_numaux;
        }
    }
```
/* Should have been a symbol...indicate the error */
.
.
}

Related Information
The ldahread subroutine, ldhread subroutine, ldshread subroutine, ldnhread subroutine, ldttbread subroutine, ldlread subroutine, ldlinit subroutine, or ldlitem subroutine.

Subroutines Overview in AIX 5L Version 5.3 General Programming Concepts: Writing and Debugging Programs.

Idlread, Idlinit, or Idlitem Subroutine

Purpose
Manipulates line number entries of a common object file function.

Library
Object File Access Routine Library (libld.a)

Syntax
#include <stdio.h>
#include <ldfcn.h>

int ldlread (ldPointer, FunctionIndex, LineNumber, LineEntry)
LDFILE *ldPointer;
int FunctionIndex;
unsigned short LineNumber;
void *LineEntry;

int ldlinit (ldPointer, FunctionIndex)
LDFILE *ldPointer;
int FunctionIndex;

int ldlitem (ldPointer, LineNumber, LineEntry)
LDFILE *ldPointer;
unsigned short LineNumber;
void *LineEntry;

Description
The ldlread subroutine searches the line number entries of the XCOFF file currently associated with the ldPointer parameter. The ldlread subroutine begins its search with the line number entry for the beginning of a function and confines its search to the line numbers associated with a single function. The function is identified by the FunctionIndex parameter, the index of its entry in the object file symbol table. The ldlread subroutine reads the entry with the smallest line number equal to or greater than the LineNumber parameter into the memory beginning at the LineEntry parameter. It is the responsibility of the calling routine to provide a pointer to a buffer large enough to contain the line number entry for the associated object file type. Since the ldopen subroutine provides magic number information (via the HEADER(ldPointer).f_magic macro), the calling application can always determine whether the LineEntry pointer should refer to a 32-bit LINENO or 64-bit LINENO_64 structure.
The `ldlinit` and `ldlitem` subroutines together perform the same function as the `ldlread` subroutine. After an initial call to the `ldlread` or `ldlinit` subroutine, the `ldlitem` subroutine may be used to retrieve successive line number entries associated with a single function. The `ldlinit` subroutine simply locates the line number entries for the function identified by the `FunctionIndex` parameter. The `ldlitem` subroutine finds and reads the entry with the smallest line number equal to or greater than the `LineNumber` parameter into the memory beginning at the `LineEntry` parameter.

### Parameters
- **ldPointer**
  Points to the `LDFILE` structure that was returned as the result of a successful call to the `ldopen`, `lddopen`, or `ldaopen` subroutine.
- **LineNumber**
  Specifies the index of the first `LineNumber` parameter entry to be read.
- **LineEntry**
  Points to a buffer that will be filled in with a `LINENO` structure from the object file.
- **FunctionIndex**
  Points to the symbol table index of a function.

### Return Values
The `ldlread`, `ldlinit`, and `ldlitem` subroutines return a SUCCESS or FAILURE value.

### Error Codes
The `ldlread` subroutine fails if there are no line number entries in the object file, if the `FunctionIndex` parameter does not index a function entry in the symbol table, or if it finds no line number equal to or greater than the `LineNumber` parameter. The `ldlinit` subroutine fails if there are no line number entries in the object file or if the `FunctionIndex` parameter does not index a function entry in the symbol table. The `ldlitem` subroutine fails if it finds no line number equal to or greater than the `LineNumber` parameter.

### Related Information
The `ldahread` subroutine, `ldfhread` subroutine, `ldgetname` subroutine, `ldshread` or `ldnshread` subroutine, `ldtbread` subroutine.

### Idlseek or Idnlseek Subroutine

#### Purpose
Seeks to line number entries of a section of a common object file.

#### Library
Object File Access Routine Library (`libld.a`)

#### Syntax
```c
#include <stdio.h>
#include <ldfcn.h>

int ldseek (ldPointer, SectionIndex)
LDFILE *ldPointer;
unsigned short SectionIndex;
```
int ldnlseek (ldPointer, SectionName)
LDFILE *ldPointer;
char *SectionName;

Description
The ldlnseek subroutine seeks to the line number entries of the section specified by the SectionIndex parameter of the common object file currently associated with the ldPointer parameter. The first section has an index of 1.

The ldlnseek subroutine seeks to the line number entries of the section specified by the SectionName parameter.

Both subroutines determine the object mode of the associated file before seeking to the relocation entries of the indicated section.

Parameters
ldPointer Points to the LDFILE structure that was returned as the result of a successful call to the ldopen or ldaopen subroutine.
SectionIndex Specifies the index of the section whose line number entries are to be seeked to.
SectionName Specifies the name of the section whose line number entries are to be seeked to.

Return Values
The ldlnseek and ldlnseek subroutines return a SUCCESS or FAILURE value.

Error Codes
The ldlnseek subroutine fails if the SectionIndex parameter is greater than the number of sections in the object file. The ldlnseek subroutine fails if there is no section name corresponding with the SectionName parameter. Either function fails if the specified section has no line number entries or if it cannot seek to the specified line number entries.

Related Information
The ldohseek ([ldohseek Subroutine] subroutine, ldseek or ldrseek [ldrseek or ldnrseek Subroutine] subroutine, ldsseek or ldnsseek [lddsseek or ldnsseek Subroutine] on page 703) subroutine, ldrseek or ldnrseek [ldrseek or ldnrseek Subroutine] subroutine, ldtsseek [ldtsseek Subroutine] subroutine.

Subroutines, Example Programs, and Libraries in AIX 5L Version 5.3 General Programming Concepts: Writing and Debugging Programs.

Idohseek Subroutine

Purpose
Seeks to the optional file header of a common object file.

Library
Object File Access Routine Library (libld.a)

Syntax
#include <stdio.h>
#include <ldfcn.h>
int ldohseek (ldPointer)
LDFILE *ldPointer;

Description
The ldohseek subroutine seeks to the optional auxiliary header of the common object file currently associated with the ldPointer parameter. The subroutine determines the object mode of the associated file before seeking to the end of its file header.

Parameters
ldPointer Points to the LDFILE structure that was returned as the result of a successful call to ldopen or ldaopen subroutine.

Return Values
The ldohseek subroutine returns a SUCCESS or FAILURE value.

Error Codes
The ldohseek subroutine fails if the object file has no optional header, if the file is not a 32-bit or 64-bit object file, or if it cannot seek to the optional header.

Related Information
The ldiseek or ldnlseek subroutine, ldrseek or ldnrseek subroutine, ldsseek or ldnsseek subroutine, ldcbseek subroutine.

ldopen or ldaopen Subroutine

Purpose
Opens an object or archive file for reading.

Library
Object File Access Routine Library (libld.a)

Syntax
#include <stdio.h>
#include <ldfcn.h>
LDFILE *ldopen(FileName, ldPointer)
char *FileName;
LDFILE *ldPointer;

LDFILE *ldaopen(FileName, ldPointer)
char *FileName;
LDFILE *ldPointer;

LDFILE *lddopen(FileDescriptor, type, ldPointer)
int FileDescriptor;
char *type;
LDFILE *ldPointer;

**Description**

The `ldopen` and `ldclose` subroutines provide uniform access to both simple object files and object files that are members of archive files. Thus, an archive of object files can be processed as if it were a series of ordinary object files.

If the `ldPointer` is null, the `ldopen` subroutine opens the file named by the `FileName` parameter and allocates and initializes an `LDFILE` structure, and returns a pointer to the structure.

If the `ldPointer` parameter is not null and refers to an `LDFILE` for an archive, the structure is updated for reading the next archive member. In this case, and if the value of the `TYPE(ldPointer)` macro is the archive magic number `ARTYPE`.

The `ldopen` and `ldclose` subroutines are designed to work in concert. The `ldclose` subroutine returns failure only when the `ldPointer` refers to an archive containing additional members. Only then should the `ldopen` subroutine be called with a num-null `ldPointer` argument. In all other cases, in particular whenever a new `FileName` parameter is opened, the `ldopen` subroutine should be called with a null `ldPointer` argument.

If the value of the `ldPointer` parameter is not null, the `ldaopen` subroutine opens the `FileName` parameter again and allocates and initializes a new `LDFILE` structure, copying the `TYPE`, `OFFSET`, and `HEADER` fields from the `ldPointer` parameter. The `ldaopen` subroutine returns a pointer to the new `ldfile` structure. This new pointer is independent of the old pointer, `ldPointer`. The two pointers may be used concurrently to read separate parts of the object file. For example, one pointer may be used to step sequentially through the relocation information, while the other is used to read indexed symbol table entries.

The `lddopen` function accesses the previously opened file referenced by the `FileDescriptor` parameter. In all other respects, it functions the same as the `ldopen` subroutine.

For AIX 4.3.2 and above, the functions transparently open both 32-bit and 64-bit object files, as well as both small format and large format archive files. Once a file or archive is successfully opened, the calling application can examine the `HEADER(ldPointer).f_magic` field to check the magic number of the file or archive member associated with `ldPointer`. (This is necessary due to an archive potentially containing members that are not object files.) The magic numbers `U802TOCMAGIC` and (for AIX 4.3.2 and above) `U803XTOCMAGIC` are defined in the `ldfcn.h` file. If the value of `TYPE(ldPointer)` is the archive magic number `ARTYPE`, the flags field can be checked for the archive type. Large format archives will have the flag bit `AR_TYPE_BIG` set in `LDFLAGS(ldPointer)`. Large format archives are available on AIX 4.3 and later.

**Parameters**

- **FileName**
  Specify the file name of an object file or archive.

- **ldPointer**
  Points to an `LDFILE` structure.

- **FileDescriptor**
  Specifies a valid open file descriptor.

- **type**
  Points to a character string specifying the mode for the open file. The `fdopen` function is used to open the file.

**Error Codes**

Both the `ldopen` and `ldaopen` subroutines open the file named by the `FileName` parameter for reading. Both functions return a null value if the `FileName` parameter cannot be opened, or if memory for the `LDFILE` structure cannot be allocated.
A successful open does not ensure that the given file is a common object file or an archived object file.

Examples
The following is an example of code that uses the ldopen and ldclose subroutines:
/* for each FileName to be processed */
ldPointer = NULL;
do
if(ldPointer = ldopen(FileName, ldPointer)) != NULL)
    /* check magic number */
    /* process the file */
    
    while(ldclose(ldPointer) == FAILURE );

Related Information
The ldclose or ldaclose subroutine, fopen, fopen64, freopen, freopen64, or fdopen subroutine.

Subroutines, Example Programs, and Libraries in AIX 5L Version 5.3 General Programming Concepts: Writing and Debugging Programs.

Idrseek or ldnrseek Subroutine

Purpose
Seeks to the relocation entries of a section of an XCOFF file.

Library
Object File Access Routine Library (libld.a)

Syntax
#include <stdio.h>
#include <ldfcn.h>

int ldrseek (ldfile *ldPointer, SectionIndex)
ldfile *ldPointer;
unsigned short SectionIndex;

int ldnrseek (ldfile *ldPointer, SectionName)
ldfile *ldPointer;
char *SectionName;

Description
The ldrseek subroutine seeks to the relocation entries of the section specified by the SectionIndex parameter of the common object file currently associated with the ldPointer parameter.

The ldnrseek subroutine seeks to the relocation entries of the section specified by the SectionName parameter.

For AIX 4.3.2 and above, both subroutines determine the object mode of the associated file before seeking to the relocation entries of the indicated section.
Parameters

ldPointer
Points to an LDFILE structure that was returned as the result of a successful call to the ldopen, lddopen, or ldaopen subroutines.

SectionIndex
Specifies an index for the section whose relocation entries are to be sought.

SectionName
Specifies the name of the section whose relocation entries are to be sought.

Return Values

The ldrseek and ldnrseek subroutines return a SUCCESS or FAILURE value.

Error Codes

The ldrseek subroutine fails if the contents of the SectionIndex parameter are greater than the number of sections in the object file. The ldnrseek subroutine fails if there is no section name corresponding with the SectionName parameter. Either function fails if the specified section has no relocation entries or if it cannot seek to the specified relocation entries.

Note: The first section has an index of 1.

Related Information

The ldohseek ("ldohseek Subroutine" on page 700) subroutine, ldiseek or ldnlseek ("ldiseek or ldnlseek Subroutine" on page 699) subroutine, ldseek or ldnseek ("ldseek or ldnseek Subroutine" on page 706) subroutine, ldthseek ("ldthseek Subroutine" on page 709) subroutine.

Subroutines, Example Programs, and Libraries in AIX 5L Version 5.3 General Programming Concepts: Writing and Debugging Programs.

Idshread or ldnsrthread Subroutine

Purpose

Reads a section header of an XCOFF file.

Library

Object File Access Routine Library (libld.a)

Syntax

#include <stdio.h>
#include <ldfcn.h>

int ldshread (LDFILE *ldPointer, SectionIndex, SectionHead)
LDFILE *ldPointer;
unsigned short SectionIndex;
void *SectionHead;

int ldnshread (ldPointer, SectionName, SectionHead)
LDFILE *ldPointer;
char *SectionName;
void *SectionHead;
Description
The ldshread subroutine reads the section header specified by the SectionIndex parameter of the common object file currently associated with the ldPointer parameter into the area of memory beginning at the location specified by the SectionHead parameter.

The ldnshread subroutine reads the section header named by the SectionName argument into the area of memory beginning at the location specified by the SectionHead parameter. It is the responsibility of the calling routine to provide a pointer to a buffer large enough to contain the section header of the associated object file. Since the ldopen subroutine provides magic number information (via the HEADER(ldPointer).f_magic macro), the calling application can always determine whether the SectionHead pointer should refer to a 32-bit SCNHDR or 64-bit SCNHDR_64 structure.

Only the first section header named by the SectionName argument is returned by the ldshread subroutine.

Parameters
IdPointer Points to an LDFILE structure that was returned as the result of a successful call to the ldopen, ldlopen, or ldaopen subroutine.
SectionIndex Specifies the index of the section header to be read.
   Note: The first section has an index of 1.
SectionHead Points to a buffer large enough to accept either a 32-bit or a 64-bit SCNHDR structure, according to the object mode of the file being read.
SectionName Specifies the name of the section header to be read.

Return Values
The ldshread and ldnshread subroutines return a SUCCESS or FAILURE value.

Error Codes
The ldshread subroutine fails if the SectionIndex parameter is greater than the number of sections in the object file. The ldnshread subroutine fails if there is no section with the name specified by the SectionName parameter. Either function fails if it cannot read the specified section header.

Examples
The following is an example of code that opens an object file, determines its mode, and uses the ldnshread subroutine to acquire the .text section header. This code would be compiled with both __XCOFF32__ and __XCOFF64__ defined:

```c
#define __XCOFF32__
#define __XCOFF64__
#include <ldfcn.h>

/* for each FileName to be processed */
if ( (ldPointer = ldopen(FileName, ldPointer)) != NULL ) {
    SCNHDR SectionHead32;
    SCNHDR_64 SectionHead64;
    void *SectionHeader;
    if ( HEADER(ldPointer).f_magic == U802TOCMAGIC )
        SectionHeader = &SectionHead32;
    else if ( HEADER(ldPointer).f_magic == U803XTOCMAGIC )
        SectionHeader = &SectionHead64;
```
else
    SectionHeader = NULL;

if ( SectionHeader && ldnshread( ldPointer, ".text", SectionHeader ) == SUCCESS )
{
    /* ...successfully read header... */
    /* ...process according to magic number... */
}

Related Information
The ldahread subroutine, ldfhread subroutine, ldgetname subroutine, ldldread subroutine, ldlninit, or ldlinitem subroutine.

Subroutines, Example Programs, and Libraries in AIX 5L Version 5.3 General Programming Concepts: Writing and Debugging Programs.

Idsseek or Idnsseek Subroutine

Purpose
Seeks to an indexed or named section of a common object file.

Library
Object File Access Routine Library (libld.a)

Syntax
#include <stdio.h>
#include <ldfcn.h>

int ldsseek ( LDFILE *ldPointer, SectionIndex )
int ldnsseek ( LDFILE *ldPointer, SectionName );

Description
The ldsseek subroutine seeks to the section specified by the SectionIndex parameter of the common object file currently associated with the ldPointer parameter. The subroutine determines the object mode of the associated file before seeking to the indicated section.

The ldnsseek subroutine seeks to the section specified by the SectionName parameter.

Parameters

ldPointer Points to the LDFILE structure that was returned as the result of a successful call to the ldopen or ldaopen subroutine.

SectionIndex Specifies the index of the section whose line number entries are to be seeked to.

SectionName Specifies the name of the section whose line number entries are to be seeked to.
Return Values
The ldsseek and ldnsseek subroutines return a SUCCESS or FAILURE value.

Error Codes
The ldsseek subroutine fails if the SectionIndex parameter is greater than the number of sections in the object file. The ldnsseek subroutine fails if there is no section name corresponding with the SectionName parameter. Either function fails if there is no section data for the specified section or if it cannot seek to the specified section.

Note: The first section has an index of 1.

Related Information
The ldsseek or ldnsseek subroutine, ldohseek subroutine, ldrseek or ldnrseek subroutine, ldtsseek subroutine.

ldtbindex Subroutine

Purpose
Computes the index of a symbol table entry of a common object file.

Library
Object File Access Routine Library (libld.a)

Syntax
#include <stdio.h>
#include <ldfcn.h>

long ldtbindex (ldPointer)
LDFILE *ldPointer;

Description
The ldtbindex subroutine returns the index of the symbol table entry at the current position of the common object file associated with the ldPointer parameter.

The index returned by the ldtbindex subroutine may be used in subsequent calls to the ldtbread subroutine. However, since the ldtbindex subroutine returns the index of the symbol table entry that begins at the current position of the object file, if the ldtbindex subroutine is called immediately after a particular symbol table entry has been read, it returns the index of the next entry.

Parameters
ldPointer Points to the LDFILE structure that was returned as a result of a successful call to the ldopen or ldaopen subroutine.
Return Values
The ldtdbindex subroutine returns the value BADINDEX upon failure. Otherwise a value greater than or equal to zero is returned.

Error Codes
The ldtdbindex subroutine fails if there are no symbols in the object file or if the object file is not positioned at the beginning of a symbol table entry.

Note: The first symbol in the symbol table has an index of 0.

Related Information
The ldtdbread subroutine, ldtdbseek subroutine.

ldtdbread Subroutine

Purpose
Reads an indexed symbol table entry of a common object file.

Library
Object File Access Routine Library (libld.a)

Syntax
#include <stdio.h>
#include <ldfcn.h>

int ldtdbread (LDFILE *ldPointer, SymbolIndex, Symbol);

Description
The ldtdbread subroutine reads the symbol table entry specified by the SymbolIndex parameter of the common object file currently associated with the ldPointer parameter into the area of memory beginning at the Symbol parameter. It is the responsibility of the calling routine to provide a pointer to a buffer large enough to contain the symbol table entry of the associated object file. Since the ldopen subroutine provides magic number information (via the HEADER(ldPointer).f_magic macro), the calling application can always determine whether the Symbol pointer should refer to a 32-bit SYMENT or 64-bit SYMENT_64 structure.

Parameters
ldPointer Points to the LDFILE structure that was returned as the result of a successful call to the ldopen or ldaopen subroutine.
SymbolIndex Specifies the index of the symbol table entry to be read.
Symbol Points to either a 32-bit or a 64-bit SYMENT structure.
Return Values
The ldtbread subroutine returns a SUCCESS or FAILURE value.

Error Codes
The ldtbread subroutine fails if the SymbolIndex parameter is greater than or equal to the number of symbols in the object file, or if it cannot read the specified symbol table entry.

Note: The first symbol in the symbol table has an index of 0.

Related Information
The ldahread subroutine, ldthread subroutine, ldgetname subroutine, ldshread or ldnshread subroutine.

Subroutines, Example Programs, and Libraries in AIX 5L Version 5.3 General Programming Concepts: Writing and Debugging Programs.

ldtbseek Subroutine

Purpose
Seeks to the symbol table of a common object file.

Library
Object File Access Routine Library (libld.a)

Syntax
#include <stdio.h>
#include <ldfcn.h>

int ldtbseek (ldPointer)
LDFILE *ldPointer;

Description
The ldtbseek subroutine seeks to the symbol table of the common object file currently associated with the ldPointer parameter.

Parameters
ldPointer Points to the LDFILE structure that was returned as the result of a successful call to the ldopen or ldaopen subroutine.

Return Values
The ldtbseek subroutine returns a SUCCESS or FAILURE value.

Error Codes
The ldtbseek subroutine fails if the symbol table has been stripped from the object file or if the subroutine cannot seek to the symbol table.
Igamma, Igammaf, or Igammal Subroutine

Purpose
Computes the log gamma.

Syntax
#include <math.h>

extern int signgam;

double lgamma (double x);

float lgammaf (float x);

long double lgammal (long double x);

Description
The sign of Gamma (x) is returned in the external integer signgam.

The igamma, igammaf, and igammal subroutines are not reentrant. A function that is not required to be reentrant is not required to be thread-safe.

An application wishing to check for error situations should set the errno global variable to zero and call feclsetexcept(FE_ALL_EXCEPT) before calling these subroutines. Upon return, if errno is nonzero or fetestexcept(FE_INVALID | FE_DIVBYZERO | FE_OVERFLOW | FE_UNDERFLOW) is nonzero, an error has occurred.

Parameters
x Specifies the value to be computed.

Return Values
Upon successful completion, the igamma, igammaf, and igammal subroutines return the logarithmic gamma of x.

If x is a non-positive integer, a pole error shall occur and igamma, igammaf, and igammal will return +HUGE_VAL, +HUGE_VALL, and +HUGE_VALL.

If the correct value would cause overflow, a range error shall occur and igamma, igammaf, and igammal will return ±HUGE_VAL, ±HUGE_VALL, ±HUGE_VALL, respectively.

If x is NaN, a NaN is returned.
If \( x \) is 1 or 2, +0 is returned.

If \( x \) is \( \pm \text{Inf} \), +Inf is returned.

**Related Information**
- “exp, expf, or expl Subroutine” on page 244
- “fclearexcept Subroutine” on page 262
- “fetestexcept Subroutine” on page 270
- “class, _class, finite, isnan, or unordered Subroutines” on page 167

`math.h` in AIX 5L Version 5.3 Files Reference.

**lineout Subroutine**

**Purpose**
Formats a print line.

**Library**
None (provided by the print formatter)

**Syntax**

```c
#include <piostruct.h>

int lineout (FILE *fileptr);
```

**Description**

The `lineout` subroutine is invoked by the formatter driver only if the `setup` subroutine returns a non-null pointer. This subroutine is invoked for each line of the document being formatted. The `lineout` subroutine reads the input data stream from the `fileptr` parameter. It then formats and outputs the print line until it recognizes a situation that causes vertical movement on the page.

The `lineout` subroutine should process all characters to be printed and all printer commands related to horizontal movement on the page.

The `lineout` subroutine should not output any printer commands that cause vertical movement on the page. Instead, it should update the `vpos` (new vertical position) variable pointed to by the `shars_vars` structure that it shares with the formatter driver to indicate the new vertical position on the page. It should also refresh the `shar_vars` variables for vertical increment and vertical decrement (reverse line-feed) commands.

When the `lineout` subroutine returns, the formatter driver sends the necessary commands to the printer to advance to the new vertical position on the page. This position is specified by the `vpos` variable. The formatter driver automatically handles top and bottom margins, new pages, initial pages to be skipped, and progress reports to the `qdaemon` daemon.

The following conditions can cause vertical movements:
- Line-feed control character or variable line-feed control sequence
- Vertical-tab control character
- Form-feed control character
- Reverse line-feed control character
- A line too long for the printer that wraps to the next line

Other conditions unique to a specific printer also cause vertical movement.
Parameters

fileptr Specifies a file structure for the input data stream.

Return Values
Upon successful completion, the lineout subroutine returns the number of bytes processed from the input data stream. It excludes the end-of-file character and any control characters or escape sequences that result only in vertical movement on the page (for example, line feed or vertical tab).

If a value of 0 is returned and the value in the vpos variable pointed to by the shars_vars structure has not changed, or there are no more data bytes in the input data stream, the formatter driver assumes that printing is complete.

If the lineout subroutine detects an error, it uses the piomsgout subroutine to issue an error message. It then invokes the pioexit subroutine with a value of PIOEXITBAD.

Note: If either the piocmdout or piogetstr subroutine detects an error, it automatically issues its own error messages and terminates the print job.

Related Information
The piocmdout subroutine, pioexit subroutine, piogetstr subroutine, piomsgout subroutine, setup subroutine.


Print formatter example in AIX 5L Version 5.3 General Programming Concepts: Writing and Debugging Programs.

link Subroutine

Purpose
Creates an additional directory entry for an existing file.

Library
Standard C Library (libc.a)

Syntax
#include <unistd.h>

int link (const char *Path1, Path2);

Description
The link subroutine creates an additional hard link (directory entry) for an existing file. Both the old and the new links share equal access rights to the underlying object.

Parameters
Path1 Points to the path name of an existing file.
Path2 Points to the path name of the directory entry to be created.
Notes:
1. If Network File System (NFS) is installed on your system, these paths can cross into another node.
2. With hard links, both the Path1 and Path2 parameters must reside on the same file system. If Path1 is a symbolic link, an error is returned. Creating links to directories requires root user authority.

Return Values
Upon successful completion, the link subroutine returns a value of 0. Otherwise, a value of -1 is returned, and the errno global variable is set to indicate the error.

Error Codes
The link subroutine is unsuccessful if one of the following is true:

- EACCESS Indicates the requested link requires writing in a directory that denies write permission.
- EDQUOT Indicates the directory in which the entry for the new link is being placed cannot be extended, or disk blocks could not be allocated for the link because the user or group quota of disk blocks or i-nodes on the file system containing the directory has been exhausted.
- EEXIST Indicates the link named by the Path2 parameter already exists.
- EMLINK Indicates the file already has the maximum number of links.
- ENOENT Indicates the file named by the Path1 parameter does not exist.
- ENOSPC Indicates the directory in which the entry for the new link is being placed cannot be extended because there is no space left on the file system containing the directory.
- EPERM Indicates the file named by the Path1 parameter is a directory, and the calling process does not have root user authority.
- EROFS Indicates the requested link requires writing in a directory on a read-only file system.
- EXDEV Indicates the link named by the Path2 parameter and the file named by the Path1 parameter are on different file systems, or the file named by Path1 refers to a named STREAM.

The link subroutine can be unsuccessful for other reasons. See Appendix A, "Base Operating System Error Codes for Services That Require Path-Name Resolution," on page 1323 for a list of additional errors.

If NFS is installed on the system, the link subroutine is unsuccessful if the following is true:

- ETIMEDOUT Indicates the connection timed out.

Related Information
The symlink subroutine, [unlink](http://example.com/unlink) subroutine.

The [link](http://example.com/link) or [unlink](http://example.com/unlink) command, [ln](http://example.com/ln) command, [rm](http://example.com/rm) command.

Files, Directories, and File Systems for Programmers in AIX 5L Version 5.3 General Programming Concepts: Writing and Debugging Programs.

**lio_listio or lio_listio64 Subroutine**

The lio_listio or lio_listio64 subroutine includes information for the POSIX AIO lio_listio subroutine (as defined in the IEEE std 1003.1-2001), and the Legacy AIO lio_listio subroutine.

**POSIX AIO lio_listio Subroutine**

**Purpose**
Initiates a list of asynchronous I/O requests with a single call.
#include <aio.h>

int lio_listio(int mode, const struct aiocb ** restrict list, int nent, struct sigevent * restrict sig);

Description

The lio_listio subroutine initiates a list of I/O requests with a single function call.

The mode parameter takes one of the values (LIO_WAIT, LIO_NOWAIT or LIO_NOWAIT_AIOWAIT) declared in <aio.h> and determines whether the subroutine returns when the I/O operations have been completed, or as soon as the operations have been queued. If the mode parameter is set to LIO_WAIT, the subroutine waits until all I/O is complete and the sig parameter is ignored.

If the mode parameter is set to LIO_NOWAIT or LIO_NOWAIT_AIOWAIT, the subroutine returns immediately. If LIO_NOWAIT is set, asynchronous notification occurs, according to the sig parameter, when all I/O operations complete. If sig is NULL, no asynchronous notification occurs. If sig is not NULL, asynchronous notification occurs when all the requests in list have completed. If LIO_NOWAIT_AIOWAIT is set, the aio_nwait subroutine must be called for the aio control blocks to be updated. For more information, see the [aio_nwait Subroutine] on page 47.

The I/O requests enumerated by list are submitted in an unspecified order.

The list parameter is an array of pointers to aiocb structures. The array contains nent elements. The array may contain NULL elements, which are ignored.

The aio_lio_opcode field of each aiocb structure specifies the operation to be performed. The supported operations are LIO_READ, LIO_WRITE, and LIO_NOP; these symbols are defined in <aio.h>. The LIO_NOP operation causes the list entry to be ignored. If the aio_lio_opcode element is equal to LIO_READ, an I/O operation is submitted as if by a call to aio_read with the aiocbp argument equal to the address of the aiocb structure. If the aio_lio_opcode element is equal to LIO_WRITE, an I/O operation is submitted as if by a call to aio_write with the aiocbp argument equal to the address of the aiocb structure.

The aio_fildes member specifies the file descriptor on which the operation is to be performed.

The aio_buf member specifies the address of the buffer to or from which the data is transferred.

The aio_nbytes member specifies the number of bytes of data to be transferred.

The members of the aiocb structure further describe the I/O operation to be performed, in a manner identical to that of the corresponding aiocb structure when used by the aio_read and aio_write subroutines.

The nent parameter specifies how many elements are members of the list.

The behavior of the lio_listio subroutine is altered according to the definitions of synchronized I/O data integrity completion and synchronized I/O file integrity completion if synchronized I/O is enabled on the file associated with aio_fildes.

For regular files, no data transfer occurs past the offset maximum established in the open file description.
Parameters

mode
Determines whether the subroutine returns when the I/O operations are completed, or as soon as the operations are queued.

list
An array of pointers to aio control structures defined in the `aio.h` file.

nent
Specifies the length of the array.

sig
Determines when asynchronous notification occurs.

Execution Environment
The `lio_listio` and `lio_listio64` subroutines can be called from the process environment only.

Return Values

**EAGAIN**
The resources necessary to queue all the I/O requests were not available. The application may check the error status of each `aiocb` to determine the individual request(s) that failed.

The number of entries indicated by `nent` would cause the system-wide limit (AIO_MAX) to be exceeded.

**EINVAL**
The `mode` parameter is not a proper value, or the value of `nent` was greater than AIO_LISTIO_MAX.

**EINTR**
A signal was delivered while waiting for all I/O requests to complete during an LIO_WAIT operation. Since each I/O operation invoked by the `lio_listio` subroutine may provoke a signal when it completes, this error return may be caused by the completion of one (or more) of the very I/O operations being awaited. Outstanding I/O requests are not canceled, and the application examines each list element to determine whether the request was initiated, canceled, or completed.

**EIO**
One or more of the individual I/O operations failed. The application may check the error status for each `aiocb` structure to determine the individual request(s) that failed.

If the `lio_listio` subroutine succeeds or fails with errors of **EAGAIN**, **EINTR**, or **EIO**, some of the I/O specified by the list may have been initiated. If the `lio_listio` subroutine fails with an error code other than **EAGAIN**, **EINTR**, or **EIO**, no operations from the list were initiated. The I/O operation indicated by each list element can encounter errors specific to the individual read or write function being performed. In this event, the error status for each `aiocb` control block contains the associated error code. The error codes that can be set are the same as would be set by the read or write subroutines, with the following additional error codes possible:

**EAGAIN**
The requested I/O operation was not queued due to resource limitations.

**ECANCELED**
The requested I/O was canceled before the I/O completed due to an `aiocb` cancel request.

**EFBIG**
The `aiocb` argument is LIO_WRITE, the file is a regular file, `aio_nbytes` is greater than 0, and `aio_offset` is greater than or equal to the offset maximum in the open file description associated with `aio_fildes`.

**EINPROGRESS**
The requested I/O is in progress.

**EOVERFLOW**
The `aiocb` argument is set to LIO_READ, the file is a regular file, `aio_nbytes` is greater than 0, and the `aio_offset` argument is before the end-of-file and is greater than or equal to the offset maximum in the open file description associated with `aio_fildes`.

Related Information

"aio_cancel or aio_cancel64 Subroutine" on page 38, "aio_error or aio_error64 Subroutine" on page 42, "aio_read or aio_read64 Subroutine" on page 50, "aio_return or aio_return64 Subroutine" on page 55, "aio_suspend or aio_suspend64 Subroutine" on page 58, "aio_write or aio_write64 Subroutine" on page 61, "close Subroutine" on page 175, "exec: execl, execlp, execv, execve, execvp, or execvp Subroutine" on page 66.
Legacy AIO lio_listio Subroutine

Purpose
Initiates a list of asynchronous I/O requests with a single call.

Syntax

```c
#include <aio.h>

int lio_listio (cmd, list, nent, eventp)
int cmd, nent;
struct liocb *list[];
struct event *eventp;

int lio_listio64
(cmd, list, nent, eventp)
int cmd, nent; struct liocb64 *list;
struct event *eventp;
```

Description
The lio_listio subroutine allows the calling process to initiate the nent parameter asynchronous I/O requests. These requests are specified in the liocb structures pointed to by the elements of the list array. The call may block or return immediately depending on the cmd parameter. If the cmd parameter requests that I/O completion be asynchronously notified, a SIGIO signal is delivered when all I/O operations are completed.

The lio_listio64 subroutine is similar to the lio_listio subroutine except that it takes an array of pointers to liocb64 structures. This allows the lio_listio64 subroutine to specify offsets in excess of OFF_MAX (2 gigabytes minus 1).

In the large file enabled programming environment, lio_listio is redefined to be lio_listio64.

Note: The pointer to the event structure eventp parameter is currently not in use, but is included for future compatibility.
Parameters

The cmd parameter takes one of the following values:

- **LIO_WAIT**: Queues the requests and waits until they are complete before returning.
- **LIO_NOWAIT**: Queues the requests and returns immediately, without waiting for them to complete. The event parameter is ignored.
- **LIO_NOWAIT_AIOWAIT**: Queues the requests and returns immediately, without waiting for them to complete. The aio_nwait subroutine must be called for the aio control blocks to be updated.
- **LIO_ASYNC**: Queues the requests and returns immediately, without waiting for them to complete. An enhanced signal is delivered when all the operations are completed. Currently this command is not implemented.
- **LIO_ASIG**: Queues the requests and returns immediately, without waiting for them to complete. A SIGIO signal is generated when all the I/O operations are completed.

Points to an array of pointers to liocb structures. The structure array contains nent elements:

- **lio_aiocb**: The asynchronous I/O control block associated with this I/O request. This is an actual aiocb structure, not a pointer to one.
- **lio_fildes**: Identifies the file object on which the I/O is to be performed.
- **lio_opcode**: This field may have one of the following values defined in the /usr/include/sys/aio.h file:
  - **LIO_READ**: Indicates that the read I/O operation is requested.
  - **LIO_WRITE**: Indicates that the write I/O operation is requested.
  - **LIO_NOP**: Specifies that no I/O is requested (that is, this element will be ignored).

Specifies the number of entries in the array of pointers to listio structures.

Points to an event structure to be used when the cmd parameter is set to the LIO_ASYNC value. This parameter is currently ignored.

Execution Environment

The llio_listio and llio_listio64 subroutines can be called from the process environment only.

Return Values

When the llio_listio subroutine is successful, it returns a value of 0. Otherwise, it returns a value of -1 and sets the errno global variable to identify the error. The returned value indicates the success or failure of the llio_listio subroutine itself and not of the asynchronous I/O requests (except when the command is LIO_WAIT). The aio_error subroutine returns the status of each I/O request.

If the llio_listio subroutine succeeds or fails with errors of EAGAIN, EINTR, or EIO, some of the I/O specified by the list might have been initiated. If the llio_listio subroutine fails with an error code other than EAGAIN, EINTR, or EIO, no operations from the list were initiated. The I/O operation indicated by each list element can encounter errors specific to the individual read or write function being performed. In this
event, the error status for each `aiocb` control block contains the associated error code. The error codes that can be set are the same as would be set by the read or write subroutines, with the following additional error codes possible:

- **EAGAIN** Indicates that the system resources required to queue the request are not available. Specifically, the transmit queue may be full, or the maximum number of opens may have been reached.
- **EINVAL** Indicates that the `aio_whence` field does not have a valid value or that the resulting pointer is not valid.
- **EIO** One or more of the individual I/O operations failed. The application can check the error status for each `aiocb` structure to determine the individual request that failed.
- **EINTR** Indicates that a signal or event interrupted the `lio_listio` subroutine call.

**Related Information**
The `aio_cancel` or `aio_cancel64` ("aio_cancel or aio_cancel64 Subroutine" on page 38) subroutine, `aio_error` or `aio_error64` ("aio_error or aio_error64 Subroutine" on page 42) subroutine, `aio_read` or `aio_read64` ("aio_read or aio_read64 Subroutine" on page 50) subroutine, `aio_return` or `aio_return64` ("aio_return or aio_return64 Subroutine" on page 55) subroutine, `aio_suspend` or `aio_suspend64` ("aio_suspend or aio_suspend64 Subroutine" on page 58) subroutine, `aio_write` or `aio_write64` ("aio_write or aio_write64 Subroutine" on page 61) subroutine.

The `listea` subroutine retrieves the list of extended attribute names associated with the given `path` in the file system. The `list` is the set of (NULL-terminated) names, one after the other. Names of extended attributes to which the calling process does not have access might be omitted from the list. The length of the attribute name list is returned. The `flistea` subroutine is identical to `listea`, except that it takes a file descriptor instead of a path. The `llistea` subroutine is identical to `listea`, except, in the case of a symbolic link, the link itself is interrogated, not the file that it refers to.

**Listea Subroutine**

**Purpose**
Lists the extended attributes associated with a file.

**Syntax**

```c
#include <sys/ea.h>

ssize_t listea(const char *path, char *list, size_t size);
ssize_t flistea(int filedes, char *list, size_t size);
ssize_t llistea (const char *path, char *list, size_t size);
```

**Description**
Extended attributes are name:value pairs associated with the file system objects (such as files, directories, and symlinks). They are extensions to the normal attributes that are associated with all objects in the file system (that is, the `stat(2)` data).

Do not define an extended attribute name with eight characters prefix "(0xF8)SYSTEM(0xF8)". Prefix "(0xF8)SYSTEM(0xF8)" is reserved for system use only.

**Note:** The 0xF8 prefix represents a non-printable character.

The `listea` subroutine retrieves the list of extended attribute names associated with the given `path` in the file system. The `list` is the set of (NULL-terminated) names, one after the other. Names of extended attributes to which the calling process does not have access might be omitted from the list. The length of the attribute name list is returned. The `flistea` subroutine is identical to `listea`, except that it takes a file descriptor instead of a path. The `llistea` subroutine is identical to `listea`, except, in the case of a symbolic link, the link itself is interrogated, not the file that it refers to.
An empty buffer of size 0 can be passed into these calls to return the current size of the list of extended attribute names, which can be used to estimate whether the size of a buffer is sufficiently large to hold the list of names.

Parameters

- **path**: The path name of the file.
- **list**: A pointer to a buffer in which the list of attributes will be stored.
- **size**: The size of the buffer.
- **filedes**: A file descriptor for the file.

Return Values

If the `listea` subroutine succeeds, a nonnegative number is returned that indicates the length in bytes of the attribute name list. Upon failure, -1 is returned and **errno** is set appropriately.

Error Codes

- **EACCES**: Caller lacks read permission on the base file, or lacks the appropriate ACL privileges for named attribute **read**.
- **EFAULT**: A bad address was passed for **path** or **list**.
- **EFORMAT**: File system is capable of supporting EAs, but EAs are disabled.
- **ENOTSUP**: Extended attributes are not supported by the file system.
- **ERANGE**: The size of the value buffer is too small to hold the result.

Related Information

- “getea Subroutine” on page 359, removeea Subroutine, setea Subroutine, stateea Subroutine

### llrint, llrintf, or llrintl Subroutine

**Purpose**

Rounds to the nearest integer value using current rounding direction.

**Syntax**

```c
#include <math.h>

long long llrint (double x);
long long llrintf (float x);
long long llrintl (long double x);
```

**Description**

The `llrint`, `llrintf`, and `llrintl` subroutines round the `x` parameter to the nearest integer value, rounding according to the current rounding direction.

An application wishing to check for error situations should set the **errno** global variable to zero and call `feclearexcept(FE_ALL_EXCEPT)` before calling these subroutines. Upon return, if **errno** is nonzero or `fetestexcept(FE_INVALID | FE_DIVBYZERO | FE_OVERFLOW | FE_UNDERFLOW)` is nonzero, an error has occurred.
Parameters

\[ x \] Specifies the value to be rounded.

Return Values
Upon successful completion, the `llrint`, `llrintf`, and `llrintl` subroutines return the rounded integer value.

If \( x \) is NaN, a domain error occurs, and an unspecified value is returned.

If \( x \) is +Inf, a domain error occurs and an unspecified value is returned.

If \( x \) is −Inf, a domain error occurs and an unspecified value is returned.

If the correct value is positive and too large to represent as a `long long`, a domain error occur and an unspecified value is returned.

If the correct value is negative and too large to represent as a `long long`, a domain error occurs and an unspecified value is returned.

Related Information
- “feclearexcept Subroutine” on page 262
- “fetestexcept Subroutine” on page 270

`math.h` in AIX 5L Version 5.3 Files Reference.

Ilround, Iiroundf, or Iiroundl Subroutine

Purpose
Rounds to the nearest integer value.

Syntax
```
#include <math.h>

long long llround (x)
double x;

long long llroundf (x)
float x;

long long llroundl (x)
long double x;
```

Description
The `llround`, `llroundf`, or `llroundl` subroutines round the \( x \) parameter to the nearest integer value, rounding halfway cases away from zero, regardless of the current rounding direction.

An application wishing to check for error situations should set the `errno` global variable to zero and call `feclearexcept(FE_ALL_EXCEPT)` before calling these subroutines. Upon return, if `errno` is nonzero or `fetestexcept(FE_INVALID | FE_DIVBYZERO | FE_OVERFLOW | FE_UNDERFLOW)` is nonzero, an error has occurred.

Parameters

\[ x \] Specifies the value to be rounded.
Return Values
Upon successful completion, the _llround, _llroundf, or _llroundl subroutines return the rounded integer value.

If \( x \) is NaN, a domain error occurs, and an unspecified value is returned.

If \( x \) is +Inf, a domain error occurs and an unspecified value is returned.

If \( x \) is –Inf, a domain error occurs and an unspecified value is returned.

If the correct value is positive and too large to represent as a long long, a domain error occurs and an unspecified value is returned.

If the correct value is negative and too large to represent as a long long, a domain error occurs and an unspecified value is returned.

Related Information
"f EClearexcept Subroutine" on page 262 and "fetestexcept Subroutine" on page 270.

\texttt{math.h} in AIX 5L Version 5.3 Files Reference.

load Subroutine

Purpose
Loads a module into the current process.

Syntax
\begin{verbatim}
int *load (ModuleName, Flags, LibraryPath);
char *ModuleName;
uint Flags;
char *LibraryPath;
\end{verbatim}

Description
The \texttt{load} subroutine loads the specified module into the calling process's address space. A module can be a regular file or a member of an archive. When adding a new module to the address space of a 32-bit process, the load operation may cause the break value to change.

The \texttt{exec} subroutine is similar to the \texttt{load} subroutine, except that:
\begin{itemize}
  \item The \texttt{load} subroutine does not replace the current program with a new one.
  \item The \texttt{exec} subroutine does not have an explicit library path parameter; it has only the LIBPATH and \texttt{LD_LIBRARY_PATH} environment variables. Also, these library path environment variables are ignored when the program using the \texttt{exec} subroutine has more privilege than the caller (for example, in the case of a set-UID program).
\end{itemize}

A large application can be split up into one or more modules in one of two ways that allow execution within the same process. The first way is to create each of the application's modules separately and use \texttt{load} to explicitly load a module when it is needed. The other way is to specify the relationship between the modules when they are created by defining imported and exported symbols.
Modules can import symbols from other modules. Whenever symbols are imported from one or more other modules, these modules are automatically loaded to resolve the symbol references if the required modules are not already loaded, and if the imported symbols are not specified as deferred imports. These modules can be archive members in libraries or individual files and can have either shared or private file characteristics that control how and where they are loaded.

Shared modules (typically members of a shared library archive) are loaded into the shared library region, when their access permissions allow sharing, that is, when they have read-other permission. Private modules, and shared modules without the required permissions for sharing, are loaded into the process private region.

When the loader resolves a symbol, it uses the file name recorded with that symbol to find the module that exports the symbol. If the file name contains any / (slash) characters, it is used directly and must name an appropriate file or archive member. However, if the file name is a base name (contains no / characters), the loader searches the directories specified in the default library path for a file (i.e. a module or an archive) with that base name.

The LibraryPath is a string containing one or more directory path names separated by colons. See the section "Searching for Dependent Modules" for information on library path searching.

(This paragraph only applies to AIX 4.3.1 and previous releases.) When a process is executing under ptrace control, portions of the process’s address space are recopied after the load processing completes. For a 32-bit process, the main program text (loaded in segment 1) and shared library modules (loaded in segment 13) are recopied. Any breakpoints or other modifications to these segments must be reinserted after the load call. For a 64-bit process, shared library modules are recopied after a load call. The debugger will be notified by setting the W_SLWTED flag in the status returned by wait, so that it can reinsert breakpoints.

(This paragraph only applies to AIX 4.3.2 and later releases.) When a process executing under ptrace control calls load, the debugger is notified by setting the W_SLWTED flag in the status returned by wait. Any modules newly loaded into the shared library segments will be copied to the process’s private copy of these segments, so that they can be examined or modified by the debugger.

If the program calling the load subroutine was linked on AIX 4.2 or a later release, the load subroutine will call initialization routines (init routines) for the new module and any of its dependents if they were not already loaded.

Modules loaded by this subroutine are automatically unloaded when the process terminates or when the exec subroutine is executed. They are explicitly unloaded by calling the unload subroutine.

Searching for Dependent Modules

The load operation and the exec operation differ slightly in their dependent module search mechanism. When a module is added to the address space of a running process (the load operation), the rules outlined in the next section are used to find the named module. Note that dependency relationships may be loosely defined as a tree but recursive relationships between modules may also exist. The following components may used to create a complete library search path:

1. If the L_LIBPATH_EXEC flag is set, the library search path used at exec-time.
2. The value of the LibraryPath parameter if it is non-null. Note that a null string is a valid search path which refers to the current working directory. If the LibraryPath parameter is NULL, the value of the LIBPATH environment variable, or alternatively the LD_LIBRARY_PATH environment variable (if LIBPATH is not set), is used instead.
3. The library search path contained in the loader section of the module being loaded (the ModuleName parameter).
4. The library search path contained in the loader section of the module whose immediate dependents
   are being loaded. Note that this per-module information changes when searching for each module’s
   immediate dependents.

To find the ModuleName module, components 1 and 2 are used. To find dependents, components 1, 2, 3
and 4 are used in order. Note that if any modules that are already part of the running process satisfy the
dependency requirements of the newly loaded module(s), pre-existing modules are not loaded again.

For each colon-separated portion of the aggregate search specification, if the base name is not found the
search continues. The first instance of the base name found is used; if the file is not of the proper form, or
in the case of an archive does not contain the required archive member, or does not export a definition of
a required symbol, an error occurs. The library path search is not performed when either a relative or an
absolute path name is specified for a dependent module.

The library search path stored within the module is specified at link-edit time.

The load subroutine may cause the calling process to fail if the module specified has a very long chain of
dependencies, (for example, lib1.a, which depends on lib2.a, which depends on lib3.a, etc). This is
because the loader processes such relationships recursively on a fixed-size stack. This limitation is
exposed only when processing a dependency chain that has over one thousand elements.

Parameters

ModuleName

Points to the name of the module to be loaded. The module name consists of a path name,
and, an optional member name. If the path name contains at least one / character, the name is
used directly, and no directory searches are performed to locate the file. If the path name
contains no / characters, it is treated as a base name, and should be in one of the directories
listed in the library path.

The library path is either the value of the LibraryPath parameter if not a null value, or the value
of the LIBPATH environment variable (if set; otherwise, LD_LIBRARY_PATH environment
variable, if set) or the library path used at process exec time (if the L_LIBPATH_EXEC is set).
If no library path is provided, the module should be in the current directory.

The ModuleName parameter may explicitly name an archive member. The syntax is
pathname(member) where pathname follows the rules specified in the previous paragraph, and
member is the name of a specific archive member. The parentheses are a required portion of
the specification and no intervening spaces are allowed. If an archive member is named, the
L_LOADMEMBER flag must be added to the Flags parameter. Otherwise, the entire
ModuleName parameter is treated as an explicit filename.
Flags

Modifies the behavior of the load service as follows (see the ldr.h file). If no special behavior is required, set the value of the flags parameter to 0 (zero). For compatibility, a value of 1 (one) may also be specified.

L_LIBPATH_EXEC

Specifies that the library path used at process exec time should be prepended to any library path specified in the load call (either as an argument or environment variable). It is recommended that this flag be specified in all calls to the load subroutine.

L_LOADMEMBER

Indicates that the ModuleName parameter may specify an archive member. The ModuleName argument is searched for parentheses, and if found the parameter is treated as a filename/member name pair. If this flag is present and the ModuleName parameter does not contain parenthesis the entire ModuleName parameter is treated as a filename specification. Under either condition the filename is expected to be found within the library path or the current directory.

L_NOAUTODEFER

Specifies that any deferred imports in the module being loaded must be explicitly resolved by use of the loadbind subroutine. This allows unresolved imports to be explicitly resolved at a later time with a specified module. If this flag is not specified, deferred imports (marked for deferred resolution) are resolved at the earliest opportunity when any subsequently loaded module exports symbols matching unresolved imports.

LibraryPath

Points to a character string that specifies the default library search path.

If the LibraryPath parameter is NULL, the LIBPATH environment variable is used, if set; otherwise, the LD_LIBRARY_PATH environment variable is used. See the section "Searching for Dependent Modules" on page 722 for more information.

The library path is used to locate dependent modules that are specified as basenames (that is, their pathname components do not contain a / (slash) character).

Note the difference between setting the LibraryPath parameter to null, and having the LibraryPath parameter point to a null string (""). A null string is a valid library path which consists of a single directory: the current directory.

Return Values

Upon successful completion, the load subroutine returns the pointer to function for the entry point of the module. If the module has no entry point, the address of the data section of the module is returned.

Error Codes

If the load subroutine fails, a null pointer is returned, and errno global variable is set to indicate the error. The load subroutine fails if one or more of the following are true of a module to be explicitly or automatically loaded:

- **EACCES** Indicates the file is not an ordinary file, or the mode of the program file denies execution permission, or search permission is denied on a component of the path prefix.
- **EINVAL** Indicates the file or archive member has a valid magic number in its header, but the header is damaged or is incorrect for the machine on which the file is to be run.
- **ELOOP** Indicates too many symbolic links were encountered in translating the path name.
- **ENOEXEC** Indicates an error occurred when loading or resolving symbols for the specified module. This can be due to an attempt to load a module with an invalid XCOFF header, a failure to resolve symbols that were not defined as deferred imports or several other load time related problems. The loadquery subroutine can be used to return more information about the load failure. If the main program was linked on a AIX 4.2 or later system, and if runtime linking is used, the load subroutine will fail if the runtime linker could not resolve some symbols. In this case, errno will be set to ENOEXEC, but the loadquery subroutine will not return any additional information.
ENOMEM Indicates the program requires more memory than is allowed by the system-imposed maximum.

ETXTBSY Indicates the file is currently open for writing by some process.

ENAMETOOLONG Indicates a component of a path name exceeded 255 characters, or an entire path name exceeded 1023 characters.

ENOENT Indicates a component of the path prefix does not exist, or the path name is a null value. For the dlopen subroutine, RTLD_MEMBER is not used when trying to open a member within the archive file.

ENOTDIR Indicates a component of the path prefix is not a directory.

ESTALE Indicates the process root or current directory is located in a virtual file system that has been unmounted.

Related Information
The dlopen subroutine, exec subroutine, load subroutine, loadquery subroutine, ptrace subroutine, unload subroutine.

The ld command.

The Shared Library Overview and Subroutines Overview in AIX 5L Version 5.3 General Programming Concepts: Writing and Debugging Programs.

loadbind Subroutine

Purpose
Provides specific run-time resolution of a module’s deferred symbols.

Syntax
```c
int loadbind(Flag, ExportPointer, ImportPointer)
int Flag;
void *ExportPointer, *ImportPointer;
```

Description
The loadbind subroutine controls the run-time resolution of a previously loaded object module’s unresolved imported symbols.

The loadbind subroutine is used when two modules are loaded. Module A, an object module loaded at run time with the load subroutine, has designated that some of its imported symbols be resolved at a later time. Module B contains exported symbols to resolve module A’s unresolved imports.

To keep module A’s imported symbols from being resolved until the loadbind service is called, you can specify the load subroutine flag, L_NOAUTODEFER, when loading module A.

(This paragraph only applies to AIX 4.3.1 and previous releases.) When a 32-bit process is executing under ptrace control, portions of the process’s address space are recopied after the loadbind processing completes. The main program text (loaded in segment 1) and shared library modules (loaded in segment 13) are recopied. Any breakpoints or other modifications to these segments must be reinserted after the loadbind call.

(This paragraph only applies to AIX 4.3.2 and later releases.) When a 32-bit process executing under ptrace control calls loadbind, the debugger is notified by setting the W_SLWTED flag in the status returned by wait.
When a 64-bit process under `ptrace` control calls `loadbind`, the debugger is not notified and execution of the process being debugged continues normally.

**Parameters**

- **Flag**: Currently not used.
- **ExportPointer**: Specifies the function pointer returned by the `load` subroutine when module B was loaded.
- **ImportPointer**: Specifies the function pointer returned by the `load` subroutine when module A was loaded.

**Note**: The `ImportPointer` or `ExportPointer` parameter may also be set to any exported static data area symbol or function pointer contained in the associated module. This would typically be the function pointer returned from the `load` of the specified module.

**Return Values**

A 0 is returned if the `loadbind` subroutine is successful.

**Error Codes**

A -1 is returned if an error is detected, with the `errno` global variable set to an associated error code:

- **EINVAL**: Indicates that either the `ImportPointer` or `ExportPointer` parameter is not valid (the pointer to the `ExportPointer` or `ImportPointer` parameter does not correspond to a loaded program module or library).
- **ENOMEM**: Indicates that the program requires more memory than allowed by the system-imposed maximum.

After an error is returned by the `loadbind` subroutine, you may also use the `loadquery` subroutine to obtain additional information about the `loadbind` error.

**Related Information**

The `load` (*load Subroutine* on page 721) subroutine, `loadquery` (*loadquery Subroutine*) subroutine, and `unload` subroutine.

The `ld` command.

Subroutines Overview in AIX 5L Version 5.3 General Programming Concepts: Writing and Debugging Programs.

**loadquery Subroutine**

**Purpose**

Returns error information from the `load` or `exec` subroutine; also provides a list of object files loaded for the current process.

**Syntax**

```c
int loadquery( Flags, Buffer, BufferLength)
int Flags;
void *Buffer;
unsigned int BufferLength;
```
Description

The `loadquery` subroutine obtains detailed information about an error reported on the last `load` or `exec` subroutine executed by a calling process. The `loadquery` subroutine may also be used to obtain a list of object file names for all object files that have been loaded for the current process, or the library path that was used at process exec time.

Parameters

- **Buffer**
  - Points to a `Buffer` in which to store the information.
- **BufferLength**
  - Specifies the number of bytes available in the `Buffer` parameter.
- **Flags**
  - Specifies the action of the `loadquery` subroutine as follows:
    - `L_GETINFO`:
      - Returns a list of all object files loaded for the current process, and stores the list in the `Buffer` parameter. The object file information is contained in a sequence of `LD_INFO` structures as defined in the `sys/ldr.h` file. Each structure contains the module location in virtual memory and the path name that was used to load it into memory. The file descriptor field in the `LD_INFO` structure is not filled in by this function.
    - `L_GETMESSAGE`:
      - Returns detailed error information describing the failure of a previously invoked `load` or `exec` function, and stores the error message information in `Buffer`. Upon successful return from this function the beginning of the `Buffer` contains an array of character pointers. Each character pointer points to a string in the buffer containing a loader error message. The character array ends with a null character pointer. Each error message string consists of an ASCII message number followed by zero or more characters of error-specific message data. Valid message numbers are listed in the `sys/ldr.h` file.
      - You can format the error messages returned by the `L_GETMESSAGE` function and write them to standard error using the standard system command `/usr/sbin/execerror` as follows:
        ```c
        char *buffer[1024];
        buffer[0] = "execerror";
        buffer[1] = "name of program that failed to load";
        loadquery(L_GETMESSAGES, &buffer[2],
                sizeof buffer-2*sizeof(char*));
        execvp("/usr/sbin/execerror",buffer);
        ```
      - This sample code causes the application to terminate after the messages are written to standard error.
    - `L_GETLIBPATH`:
      - Returns the library path that was used at process exec time. The library path is a null terminated character string.
    - `L_GETXINFO`:
      - Returns a list of all object files loaded for the current process and stores the list in the `Buffer` parameter. The object file information is contained in a sequence of `LD_XINFO` structures as defined in the `sys/ldr.h` file. Each structure contains the module location in virtual memory and the path name that was used to load it into memory. The file descriptor field in the `LD_XINFO` structure is not filled in by this function.

Return Values

Upon successful completion, `loadquery` returns the requested information in the caller’s buffer specified by the `Buffer` and `BufferLength` parameters.

Error Codes

The `loadquery` subroutine returns with a return code of -1 and the `errno` global variable is set to one of the following when an error condition is detected:
ENOMEM  Indicates that the caller's buffer specified by the Buffer and BufferLength parameters is too small to return the information requested. When this occurs, the information in the buffer is undefined.

EINVAL  Indicates the function specified in the Flags parameter is not valid.

EFAULT  Indicates the address specified in the Buffer parameter is not valid.

Related Information
The exec subroutine, load subroutine, loadbind subroutine, unload subroutine.

The ld command.

localeconv Subroutine

Purpose
Sets the locale-dependent conventions of an object.

Library
Standard C Library (libc.a)

Syntax
#include <locale.h>
struct lconv *localeconv ( )

Description
The localeconv subroutine sets the components of an object using the lconv structure. The lconv structure contains values appropriate for the formatting of numeric quantities (monetary and otherwise) according to the rules of the current locale.

The fields of the structure with the type char * are strings, any of which (except decimal_point) can point to a null string, which indicates that the value is not available in the current locale or is of zero length. The fields with type char are nonnegative numbers, any of which can be the CHAR_MAX value which indicates that the value is not available in the current locale. The fields of the lconv structure include the following:

char *decimal_point  The decimal-point character used to format non-monetary quantities.
char *thousands_sep  The character used to separate groups of digits to the left of the decimal point in formatted non-monetary quantities.
char *grouping

A string whose elements indicate the size of each group of digits in formatted non-monetary quantities.

The value of the grouping field is interpreted according to the following:

**CHAR_MAX**

No further grouping is to be performed.

0  The previous element is to be repeatedly used for the remainder of the digits.

**other**  The value is the number of digits that comprise the current group. The next element is examined to determine the size of the next group of digits to the left of the current group.

char *int_curr_symbol

The international currency symbol applicable to the current locale, left-justified within a four-character space-padded field. The character sequences are in accordance with those specified in ISO 4217, "Codes for the Representation of Currency and Funds."

char *currency_symbol

The local currency symbol applicable to the current locale.

char *mon_decimal_point

The decimal point used to format monetary quantities.

char *mon_thousands_sep

The separator for groups of digits to the left of the decimal point in formatted monetary quantities.

char *mon_grouping

A string whose elements indicate the size of each group of digits in formatted monetary quantities.

The value of the mon_grouping field is interpreted according to the following:

**CHAR_MAX**

No further grouping is to be performed.

0  The previous element is to be repeatedly used for the remainder of the digits.

**other**  The value is the number of digits that comprise the current group. The next element is examined to determine the size of the next group of digits to the left of the current group.

char *positive_sign

The string used to indicate a nonnegative formatted monetary quantity.

char *negative_sign

The string used to indicate a negative formatted monetary quantity.

char int_frac_digits

The number of fractional digits (those to the right of the decimal point) to be displayed in a formatted monetary quantity.

char p_cs_precedes

Set to 1 if the specified currency symbol (the currency_symbol or int_curr_symbol field) precedes the value for a nonnegative formatted monetary quantity and set to 0 if the specified currency symbol follows the value for a nonnegative formatted monetary quantity.

char p_sep_by_space

Set to 1 if the currency_symbol or int_curr_symbol field is separated by a space from the value for a nonnegative formatted monetary quantity and set to 0 if the currency_symbol or int_curr_symbol field is not separated by a space from the value for a nonnegative formatted monetary quantity.

char n_cs_precedes

Set to 1 if the currency_symbol or int_curr_symbol field precedes the value for a negative formatted monetary quantity and set to 0 if the currency_symbol or int_curr_symbol field follows the value for a negative formatted monetary quantity.

char n_sep_by_space

Set to 1 if the currency_symbol or int_curr_symbol field is separated by a space from the value for a negative formatted monetary quantity and set to 0 if the currency_symbol or int_curr_symbol field is not separated by a space from the value for a negative formatted monetary quantity. Set to 2 if the symbol and the sign string are adjacent and separated by a blank character.

char p_sign_posn

Set to a value indicating the positioning of the positive sign (the positive_sign fields) for nonnegative formatted monetary quantity.
char n_sign_posn

Set to a value indicating the positioning of the negative sign (the negative_sign fields) for a negative formatted monetary quantity.

The values of the p_sign_posn and n_sign_posn fields are interpreted according to the following definitions:

- **0**: Parentheses surround the quantity and the specified currency symbol or international currency symbol.
- **1**: The sign string precedes the quantity and the currency symbol or international currency symbol.
- **2**: The sign string follows the quantity and currency symbol or international currency symbol.
- **3**: The sign string immediately precedes the currency symbol or international currency symbol.
- **4**: The sign string immediately follows the currency symbol or international currency symbol.

The following table illustrates the rules that can be used by three countries to format monetary quantities:

<table>
<thead>
<tr>
<th>Country</th>
<th>Formats</th>
</tr>
</thead>
<tbody>
<tr>
<td>Italy</td>
<td>Positive Format: L.1234</td>
</tr>
<tr>
<td></td>
<td>Negative Format: -L.1234</td>
</tr>
<tr>
<td></td>
<td>International Format: ITL.1234</td>
</tr>
<tr>
<td>Norway</td>
<td>Positive Format: krl.234.56</td>
</tr>
<tr>
<td></td>
<td>Negative Format: krl.234.56-</td>
</tr>
<tr>
<td></td>
<td>International Format: NOK 1.234.56</td>
</tr>
<tr>
<td>Switzerland</td>
<td>Positive Format: SFr.1.234.56</td>
</tr>
<tr>
<td></td>
<td>Negative Format: SFr.1.234.56C</td>
</tr>
<tr>
<td></td>
<td>International Format: CHF 1.234.56</td>
</tr>
</tbody>
</table>

The following table shows the values of the monetary members of the structure returned by the localeconv subroutine for these countries:

<table>
<thead>
<tr>
<th>struct localeconv</th>
<th>Countries</th>
</tr>
</thead>
<tbody>
<tr>
<td>char *in_curr_symbol</td>
<td>Italy: &quot;ITL.&quot;</td>
</tr>
<tr>
<td></td>
<td>Norway: &quot;NOK&quot;</td>
</tr>
<tr>
<td></td>
<td>Switzerland: &quot;CHF&quot;</td>
</tr>
<tr>
<td>struct localeconv</td>
<td>Countries</td>
</tr>
<tr>
<td>-------------------</td>
<td>-----------</td>
</tr>
<tr>
<td>char *currency_symbol</td>
<td></td>
</tr>
<tr>
<td>Italy:</td>
<td>&quot;L.&quot;</td>
</tr>
<tr>
<td>Norway:</td>
<td>&quot;kr&quot;</td>
</tr>
<tr>
<td>Switzerland:</td>
<td>&quot;SFrs.&quot;</td>
</tr>
<tr>
<td>char *mon_decimal_point</td>
<td></td>
</tr>
<tr>
<td>Italy:</td>
<td>&quot; &quot;</td>
</tr>
<tr>
<td>Norway:</td>
<td>&quot;.&quot;</td>
</tr>
<tr>
<td>Switzerland:</td>
<td>&quot;.&quot;</td>
</tr>
<tr>
<td>char *mon_thousands_sep</td>
<td></td>
</tr>
<tr>
<td>Italy:</td>
<td>&quot;.&quot;</td>
</tr>
<tr>
<td>Norway:</td>
<td>&quot;.&quot;</td>
</tr>
<tr>
<td>Switzerland:</td>
<td>&quot;.&quot;</td>
</tr>
<tr>
<td>char *mon_grouping</td>
<td></td>
</tr>
<tr>
<td>Italy:</td>
<td>&quot;\3&quot;</td>
</tr>
<tr>
<td>Norway:</td>
<td>&quot;\3&quot;</td>
</tr>
<tr>
<td>Switzerland:</td>
<td>&quot;\3&quot;</td>
</tr>
<tr>
<td>char *positive_sign</td>
<td></td>
</tr>
<tr>
<td>Italy:</td>
<td>&quot; &quot;</td>
</tr>
<tr>
<td>Norway:</td>
<td>&quot; &quot;</td>
</tr>
<tr>
<td>Switzerland:</td>
<td>&quot; &quot;</td>
</tr>
<tr>
<td>char *negative_sign</td>
<td></td>
</tr>
<tr>
<td>Italy:</td>
<td>&quot; &quot;</td>
</tr>
<tr>
<td>Norway:</td>
<td>&quot; &quot;</td>
</tr>
<tr>
<td>Switzerland:</td>
<td>&quot;C&quot;</td>
</tr>
<tr>
<td>char int_frac_digits</td>
<td></td>
</tr>
<tr>
<td>Italy:</td>
<td>0</td>
</tr>
<tr>
<td>Norway:</td>
<td>2</td>
</tr>
<tr>
<td>Switzerland:</td>
<td>2</td>
</tr>
<tr>
<td>char frac_digits</td>
<td></td>
</tr>
<tr>
<td>Italy:</td>
<td>0</td>
</tr>
<tr>
<td>Norway:</td>
<td>2</td>
</tr>
<tr>
<td>Switzerland:</td>
<td>2</td>
</tr>
<tr>
<td>struct localeconv</td>
<td>Countries</td>
</tr>
<tr>
<td>-------------------</td>
<td>-----------</td>
</tr>
<tr>
<td>char p_cs_precedes</td>
<td>Italy: 1</td>
</tr>
<tr>
<td></td>
<td>Norway: 1</td>
</tr>
<tr>
<td></td>
<td>Switzerland: 1</td>
</tr>
<tr>
<td>char p_sep_by_space</td>
<td>Italy: 0</td>
</tr>
<tr>
<td></td>
<td>Norway: 0</td>
</tr>
<tr>
<td></td>
<td>Switzerland: 0</td>
</tr>
<tr>
<td>char n_cs_precedes</td>
<td>Italy: 1</td>
</tr>
<tr>
<td></td>
<td>Norway: 1</td>
</tr>
<tr>
<td></td>
<td>Switzerland: 1</td>
</tr>
<tr>
<td>char n_sep_by_space</td>
<td>Italy: 0</td>
</tr>
<tr>
<td></td>
<td>Norway: 0</td>
</tr>
<tr>
<td></td>
<td>Switzerland: 0</td>
</tr>
<tr>
<td>char p_sign_posn</td>
<td>Italy: 1</td>
</tr>
<tr>
<td></td>
<td>Norway: 1</td>
</tr>
<tr>
<td></td>
<td>Switzerland: 1</td>
</tr>
<tr>
<td>char n_sign_posn</td>
<td>Italy: 1</td>
</tr>
<tr>
<td></td>
<td>Norway: 2</td>
</tr>
<tr>
<td></td>
<td>Switzerland: 2</td>
</tr>
</tbody>
</table>

Return Values

A pointer to the filled-in object is returned. In addition, calls to the `setlocale` subroutine with the `LC_ALL`, `LC_MONETARY` or `LC_NUMERIC` categories may cause subsequent calls to the `localeconv` subroutine to return different values based on the selection of the locale.

Note: The structure pointed to by the return value is not modified by the program but may be overwritten by a subsequent call to the `localeconv` subroutine.

Related Information

The "nl_langinfo Subroutine" on page 897, `rpmatch` subroutine, `setlocale` subroutine.

Subroutines, Example Programs, and Libraries in AIX 5L Version 5.3 General Programming Concepts: Writing and Debugging Programs.
**lockfx, lockf, flock, or lockf64 Subroutine**

**Purpose**
Locks and unlocks sections of open files.

**Libraries**
- **lockfx, lockf**: Standard C Library (libc.a)
- **flock**: Berkeley Compatibility Library (libbsd.a)

**Syntax**
```c
#include <fcntl.h>

int lockfx (FileDescriptor, Command, Argument)
int FileDescriptor;
int Command;
struct flock * Argument;

#include <sys/lockf.h>
#include <unistd.h>

int lockf (FileDescriptor, Request, Size)
int FileDescriptor;
int Request;
off_t Size;

int lockf64 (FileDescriptor, Request, Size)
int FileDescriptor;
int Request;
off64_t Size;

#include <sys/file.h>

int flock (FileDescriptor, Operation)
int FileDescriptor;
int Operation;
```

**Description**

**Attention:** Buffered I/O does not work properly when used with file locking. Do not use the standard I/O package routines on files that are going to be locked.

The **lockfx** subroutine locks and unlocks sections of an open file. The **lockfx** subroutine provides a subset of the locking function provided by the **fcntl** subroutine.

The **lockf** subroutine also locks and unlocks sections of an open file. However, its interface is limited to setting only write (exclusive) locks.
Although the `lockfx`, `lockf`, `flock`, and `fcntl` interfaces are all different, their implementations are fully integrated. Therefore, locks obtained from one subroutine are honored and enforced by any of the lock subroutines.

The `Operation` parameter to the `lockfx` subroutine, which creates the lock, determines whether it is a read lock or a write lock.

The file descriptor on which a write lock is being placed must have been opened with write access.

`lockf64` is equivalent to `lockf` except that a 64-bit lock request size can be given. For `lockf`, the largest value which can be used is `OFF_MAX`, for `lockf64`, the largest value is `LONGLONG_MAX`.

In the large file enabled programming environment, `lockf` is redefined to be `lock64`.

The `flock` subroutine locks and unlocks entire files. This is a limited interface maintained for BSD compatibility, although its behavior differs from BSD in a few subtle ways. To apply a shared lock, the file must be opened for reading. To apply an exclusive lock, the file must be opened for writing.

Locks are not inherited. Therefore, a child process cannot unlock a file locked by the parent process.

**Parameters**

<table>
<thead>
<tr>
<th>Argument</th>
<th>Command</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>A pointer to a structure of type <code>flock</code> defined in the <code>flock.h</code> file.</td>
<td>Specifies one of the following constants for the <code>lockfx</code> subroutine:</td>
<td></td>
</tr>
<tr>
<td></td>
<td>F_SETLK</td>
<td>Sets or clears a file lock. The <code>l_type</code> field of the <code>flock</code> structure indicates whether to establish or remove a read or write lock. If a read or write lock cannot be set, the <code>lockfx</code> subroutine returns immediately with an error value of <code>-1</code>.</td>
</tr>
<tr>
<td></td>
<td>F_SETLKW</td>
<td>Performs the same function as <code>F_SETLK</code> unless a read or write lock is blocked by existing locks. In that case, the process sleeps until the section of the file is free to be locked.</td>
</tr>
<tr>
<td></td>
<td>F_GETLK</td>
<td>Gets the first lock that blocks the lock described in the <code>flock</code> structure. If a lock is found, the retrieved information overwrites the information in the <code>flock</code> structure. If no lock is found that would prevent this lock from being created, the structure is passed back unchanged except that the <code>l_type</code> field is set to <code>F_UNLCK</code>.</td>
</tr>
<tr>
<td>A file descriptor returned by a successful <code>open</code> or <code>fcntl</code> subroutine, identifying the file to which the lock is to be applied or removed.</td>
<td>Specifies one of the following constants for the <code>flock</code> subroutine:</td>
<td></td>
</tr>
<tr>
<td>LOCK_SH</td>
<td>Apply a shared (read) lock.</td>
<td></td>
</tr>
<tr>
<td>LOCK_EX</td>
<td>Apply an exclusive (write) lock.</td>
<td></td>
</tr>
<tr>
<td>LOCK_NB</td>
<td>Do not block when locking. This value can be logically ORed with either <code>LOCK_SH</code> or <code>LOCK_EX</code>.</td>
<td></td>
</tr>
<tr>
<td>LOCK_UN</td>
<td>Remove a lock.</td>
<td></td>
</tr>
</tbody>
</table>
Request

Specifies one of the following constants for the lockf subroutine:

**F_ULOCK**

Unlocks a previously locked region in the file.

**F_LOCK**

Locks the region for exclusive (write) use. This request causes the calling process to sleep if the requested region overlaps a locked region, and to resume when granted the lock.

**F_TEST**

Tests to see if another process has already locked a region. The lockf subroutine returns 0 if the region is unlocked. If the region is locked, then -1 is returned and the errno global variable is set to EACCES.

**F_TLOCK**

Locks the region for exclusive use if another process has not already locked the region. If the region has already been locked by another process, the lockf subroutine returns a -1 and the errno global variable is set to EACCES.

**Size**

The number of bytes to be locked or unlocked for the lockf subroutine. The region starts at the current location in the open file, and extends forward if the Size value is positive and backward if the Size value is negative. If the Size value is 0, the region starts at the current location and extends forward to the maximum possible file size, including the unallocated space after the end of the file.

Return Values

Upon successful completion, a value of 0 is returned. Otherwise, a value of -1 is returned and the errno global variable is set to indicate the error.

Error Codes

The lockfx, lockf, and flock subroutines fail if one of the following is true:

**EBADF**

The FileDescriptor parameter is not a valid open file descriptor.

**EINVAL**

The function argument is not one of F_LOCK, F_TLOCK, F_TEST or F_ULOCK; or size plus the current file offset is less than 0.

**EINVAL**

An attempt was made to lock a fifo or pipe.

**EDEADLK**

The lock is blocked by a lock from another process. Putting the calling process to sleep while waiting for the other lock to become free would cause a deadlock.

**ENOLCK**

The lock table is full. Too many regions are already locked.

**EINVAL**

The command parameter was F_SETLK and the process received a signal while waiting to acquire the lock.

**EOVERFLOW**

The offset of the first, or if size is not 0 then the last, byte in the requested section cannot be represented correctly in an object of type off_t.

The lockfx and lockf subroutines fail if one of the following is true:

**EACCES**

The Command parameter is F_SETLK, the l_type field is F_RDLCK, and the segment of the file to be locked is already read-locked by another process.

**EACCES**

The Command parameter is F_SETLK, the l_type field is F_WRLCK, and the segment of a file to be locked is already read-locked or write-locked by another process.

The flock subroutine fails if the following is true:

**EWOULDBLOCK**

The file is locked and the LOCK_NB option was specified.
log10, log10f, or log10l Subroutine

Purpose
Computes the Base 10 logarithm.

Syntax
```c
#include <math.h>

float log10f (x)
float x;

long double log10l (x)
long double x;

double log10 (x)
double x;
```

Description
The log10f, log10l, and log10 subroutines compute the base 10 logarithm of the x parameter, log_{10} (x).

An application wishing to check for error situations should set errno to zero and call feclearexcept(FE_ALL_EXCEPT) before calling these subroutines. Upon return, if errno is nonzero or fetestexcept(FE_INVALID | FE_DIVBYZERO | FE_OVERFLOW | FE_UNDERFLOW) is nonzero, an error has occurred.

Parameters
- **x** Specifies the value to be computed.

Return Values
Upon successful completion, the log10, log10f, and log10l subroutines return the base 10 logarithm of x.

If x is ±0, a pole error occurs and log10, log10f, and log10l subroutines return -HUGE_VAL, -HUGE_VALF and -HUGE_VALL, respectively.

For finite values of x that are less than 0, or if x is -Inf, a domain error occurs, and a NaN is returned.

If x is NaN, a NaN is returned.

If x is 1, +0 is returned.

If x is +Inf, +Inf is returned.
Error Codes
When using the libm.a library:

log10 If the x parameter is less than 0, the log10 subroutine returns a NaNQ value and sets errno to EDOM. If x= 0, the log10 subroutine returns a -HUGE_VAL value and sets errno to ERANGE.

When using libmsaa.a(-lmsaa):

log10 If the x parameter is not positive, the log10 subroutine returns a -HUGE_VAL value and sets errno to EDOM. A message indicating DOMAIN error (or SING error when x = 0) is output to standard error.

log10 If x < 0, log10l returns the value NaNQ and sets errno to EDOM. If x equals 0, log10l returns the value -HUGE_VAL but does not modify errno.

Related Information
"feclearexcept Subroutine" on page 262, "fetestexcept Subroutine" on page 270, "class, class, finite, isnan, or unordered Subroutines" on page 167, and "madd, msub, mult, mdiv, pow, gcd, invert, rpow, msqrt, mcnp, move, min, oin, fin, m_in, mout, omout, fmout, m_out, sdiv, or itom Subroutine" on page 7/6.

math.h in AIX 5L Version 5.3 Files Reference.

log1p, log1pf, or log1pl Subroutine

Purpose
Computes a natural logarithm.

Syntax
#include <math.h>
float log1pf (x)
float x;

long double log1pl (x)
long double x;

double log1p (x)
double x;

Description
The log1pf, log1pl, and log1p subroutines compute \(\log_e (1.0 + x)\).

An application wishing to check for error situations should set the errno global variable to zero and call feclearexcept(FE_ALL_EXCEPT) before calling these subroutines. Upon return, if errno is nonzero or fetestexcept(FE_INVALID | FE_DIVBYZERO | FE_OVERFLOW | FE_UNDERFLOW) is nonzero, an error has occurred.

Parameters
x Specifies the value to be computed.
Return Values
Upon successful completion, the `log1pf`, `log1pl`, and `log1p` subroutines return the natural logarithm of 1.0 + x.

If x is -1, a pole error occurs and the `log1pf`, `log1pl`, and `log1p` subroutines return -HUGE_VALF, -HUGE_VALL, and -HUGE_VAL, respectively.

For finite values of x that are less than -1, or if x is -Inf, a domain error occurs, and a NaN is returned.

If x is NaN, a NaN is returned.

If x is ±0, or +Inf, x is returned.

If x is subnormal, a range error may occur and x should be returned.

Related Information
“feclearexcept Subroutine” on page 262 and “fetestexcept Subroutine” on page 270.

math.h in AIX 5L Version 5.3 Files Reference.

log2, log2f, or log2l Subroutine

Purpose
Computes base 2 logarithm.

Syntax
```c
#include <math.h>

double log2 (x);

double x;

gfloat log2f (x)

gfloat x;

glong double log2l (x)

glong double x;
```

Description
The `log2`, `log2f`, and `log2l` subroutines compute the base 2 logarithm of the x parameter, log2 (x).

An application wishing to check for error situations should set `errno` to zero and call `feclearexcept(FE_ALL_EXCEPT)` before calling these subroutines. Upon return, if `errno` is nonzero or `fetestexcept(FE_INVALID | FE_DIVBYZERO | FE_OVERFLOW | FE_UNDERFLOW)` is nonzero, an error has occurred.

Parameters
x Specifies the value to be computed.

Return Values
Upon successful completion, the `log2`, `log2f`, and `log2l` subroutines return the base 2 logarithm of x.
If \( x \) is \( \pm 0 \), a pole error occurs and the \( \log_2 \), \( \log_{2f} \), and \( \log_{2l} \) subroutines return -HUGE_VAL, -HUGE_VALF, and -HUGE_VALL, respectively.

For finite values of \( x \) that are less than 0, or if \( x \) is -Inf, a domain error occurs, and a NaN is returned.

If \( x \) is NaN, a NaN is returned.

If \( x \) is 1, +0 is returned.

If \( x \) is +Inf, \( x \) is returned.

**Related Information**

“feclearexcept Subroutine” on page 262 and “fetestexcept Subroutine” on page 270.

`math.h` in AIX 5L Version 5.3 Files Reference.

### logbf, logbl, or logb Subroutine

**Purpose**

Computes the radix-independent exponent.

**Syntax**

```c
#include <math.h>

float logbf (x)
float x;

long double logbl (x)
long double x;

double logb (x)
double x;
```

**Description**

The \( \logbf \) and \( \logbl \) subroutines compute the exponent of \( x \), which is the integral part of \( \log r |x| \), as a signed floating-point value, for nonzero \( x \); where \( r \) is the radix of the machine's floating-point arithmetic. For AIX, FLT_RADIX \( r=2 \).

If \( x \) is subnormal, it is treated as though it were normalized; thus for finite positive \( x \):

\[
1 \leq x \cdot \text{FLT\_RADIX}^{-\logb(x)} < \text{FLT\_RADIX}
\]

An application wishing to check for error situations should set `errno` to zero and call `feclearexcept(FE_ALL_EXCEPT)` before calling these subroutines. Upon return, if `errno` is nonzero or `fetestexcept(FE_INVALID | FE_DIVBYZERO | FE_OVERFLOW | FE_UNDERFLOW)` is nonzero, an error has occurred.

**Note:** When the \( x \) parameter is finite and not zero, the \( \logb \ (x) \) subroutine satisfies the following equation:

\[
1 \leq \text{scalb} (|x|, -(\text{int}) \logb(x)) < 2
\]

**Parameters**

- \( x \) Specifies the value to be computed.
Return Values
Upon successful completion, the logbf and logbl subroutines return the exponent of x.

If x is ±0, a pole error occurs and the logbf and logbl subroutines return -HUGE_VALF and -HUGE_VALL, respectively.

If x is NaN, a NaN is returned.

If x is ±Inf, +Inf is returned.

Error Codes
The logb function returns -HUGE_VAL when the x parameter is set to a value of 0 and sets errno to EDOM.

Related Information
"feclearexcept Subroutine" on page 262 and "fetestexcept Subroutine" on page 270.

math.h in AIX 5L Version 5.3 Files Reference.

log, logf, or logl Subroutine

Purpose
Computes the natural logarithm.

Syntax
#include <math.h>

float logf (x)
float x;

long double logl (x)
long double x;

double log (x)
double x;

Description
The logf, logl, and log subroutines compute the natural logarithm of the x parameter, log_e (x).

An application wishing to check for error situations should set the errno global variable to zero and call feclearexcept(FE_ALL_EXCEPT) before calling these subroutines. Upon return, if errno is nonzero or fetestexcept(FE_INVALID | FE_DIVBYZERO | FE_OVERFLOW | FE_UNDERFLOW) is nonzero, an error has occurred.

Parameters
x Specifies the value to be computed.

Return Values
Upon successful completion, the logf, logl, and log subroutines return the natural logarithm of x.
If \( x \) is \( \pm 0 \), a pole error occurs and the \texttt{logf}, \texttt{logl}, and \texttt{log} subroutines return \texttt{-HUGE VALF} and \texttt{-HUGE VALL}, and \texttt{-HUGE VAL}, respectively.

For finite values of \( x \) that are less than 0, or if \( x \) is \(-\text{Inf} \), a domain error occurs, and a NaN is returned.

If \( x \) is NaN, a NaN is returned.

If \( x \) is 1, +0 is returned.

If \( x \) is +Inf, \( x \) is returned.

**Error Codes**

When using the \texttt{libm.a} library:

\texttt{log} If the \( x \) parameter is less than 0, the \texttt{log} subroutine returns a \texttt{NaNQ} value and sets \texttt{errno} to \texttt{EDOM}. If \( x = 0 \), the \texttt{log} subroutine returns a \texttt{-HUGE VAL} value but does not modify \texttt{errno}.

When using \texttt{libmsaa.a}-\texttt{-lmsaa}:

\texttt{log} If the \( x \) parameter is not positive, the \texttt{log} subroutine returns a \texttt{-HUGE VAL} value, and sets \texttt{errno} to a \texttt{EDOM} value. A message indicating DOMAIN error (or SING error when \( x = 0 \)) is output to standard error.

\texttt{log} If \( x < 0 \), the \texttt{logl} subroutine returns a \texttt{NaNQ} value.

**Related Information**

“\texttt{exp, expf, or expl Subroutine}” on page 244, “\texttt{feclearexcept Subroutine}” on page 262, “\texttt{fetestexcept Subroutine}” on page 270, “\texttt{class, _class, finite, isnan, or unordered Subroutines}” on page 167, and “\texttt{log10, log10f, or log10l Subroutine}” on page 736.

\[ \texttt{math.h} \text{ in AIX 5L Version 5.3 Files Reference.} \]

**loginfailed Subroutine**

**Purpose**
Records an unsuccessful login attempt.

**Library**
Security Library (\texttt{libc.a})

**Syntax**

\begin{verbatim}
#include <usersec.h>
int loginfailed (User, Host, Tty, Reason)
char *User;
char *Host;
char *Tty;
int Reason;
\end{verbatim}

**Note:** This subroutine is not thread-safe.
**Description**

The `loginfailed` subroutine performs the processing necessary when an unsuccessful login attempt occurs. If the specified user name is not valid, the `UNKNOWN_USER` value is substituted for the user name. This substitution prevents passwords entered as the user name from appearing on screen.

The following attributes in `/etc/security/lastlog` file are updated for the specified user, if the user name is valid:

- `time_last_unsuccessful_login`: Contains the current time.
- `tty_last_unsuccessful_login`: Contains the value specified by the `Tty` parameter.
- `host_last_unsuccessful_login`: Contains the value specified by the `Host` parameter, or the local hostname if the `Host` parameter is a null value.
- `unsuccessful_login_count`: Indicates the number of unsuccessful login attempts. The `loginfailed` subroutine increments this attribute by one for each failed attempt.

A login failure audit record is cut to indicate that an unsuccessful login attempt occurred. A `utmp` entry is appended to `/etc/security/failedlogin` file, which tracks all failed login attempts.

If the current unsuccessful login and the previously recorded unsuccessful logins constitute too many unsuccessful login attempts within too short of a time period (as specified by the `logindisable` and `logininterval` port attributes), the port is locked. When a port is locked, a PORT_Locked audit record is written to inform the system administrator that the port has been locked.

If the login retry delay is enabled (as specified by the `logindelay` port attribute), a sleep occurs before this subroutine returns. The length of the sleep (in seconds) is determined by the `logindelay` value multiplied by the number of unsuccessful login attempts that occurred in this process.

**Parameters**

- **User**: Specifies the user’s login name who has unsuccessfully attempted to login.
- **Host**: Specifies the name of the host from which the user attempted to login. If the `Host` parameter is Null, the name of the local host is used.
- **Tty**: Specifies the name of the terminal on which the user attempted to login.
- **Reason**: Specifies a reason code for the login failure. Valid values are `AUDIT_FAIL` and `AUDIT_FAIL_AUTH` defined in the `sys/audit.h` file.

**Security**

Access Control: The calling process must have access to the account information in the user database and the port information in the port database.

**File Accessed:**

<table>
<thead>
<tr>
<th>Mode</th>
<th>File</th>
</tr>
</thead>
<tbody>
<tr>
<td>r</td>
<td><code>/etc/security/user</code></td>
</tr>
<tr>
<td>rw</td>
<td><code>/etc/security/lastlog</code></td>
</tr>
<tr>
<td>r</td>
<td><code>/etc/security/login.cfg</code></td>
</tr>
<tr>
<td>rw</td>
<td><code>/etc/security/portlog</code></td>
</tr>
<tr>
<td>w</td>
<td><code>/etc/security/failedlogin</code></td>
</tr>
</tbody>
</table>

742  Technical Reference, Volume 1: Base Operating System and Extensions
Auditing Events:

<table>
<thead>
<tr>
<th>Event</th>
<th>Information</th>
</tr>
</thead>
<tbody>
<tr>
<td>USER_Login</td>
<td>username</td>
</tr>
<tr>
<td>PORT_Locked</td>
<td>portname</td>
</tr>
</tbody>
</table>

Return Values
Upon successful completion, the `loginfailed` subroutine returns a value of 0. If an error occurs, a value of -1 is returned and errno is set to indicate the error.

Error Codes
The `loginfailed` subroutine fails if one or more of the following values is true:

- **EACCES**: The current process does not have access to the user or port database.
- **EPERM**: The current process does not have permission to write an audit record.

Related Information
The `authenticate` subroutine, `getpcrd` subroutine, `getpenv` subroutine, `loginrestrictions` subroutine, `loginsuccess` subroutine, `setpcrd` subroutine, `setpenv` subroutine.

List of Security and Auditing Services in AIX 5L Version 5.3 General Programming Concepts: Writing and Debugging Programs.

Subroutines Overview in AIX 5L Version 5.3 General Programming Concepts: Writing and Debugging Programs.

loginrestrictions Subroutine

Purpose
Determines if a user is allowed to access the system.

Library
Security Library (`libc.a`)

Syntax
```c
#include <usersec.h>
#include <login.h>

int loginrestrictions (Name, Mode, Tty, Msg)
char * Name;
int Mode;
char * Tty;
char ** Msg;
```

Note: This subroutine is not thread-safe.
Description

The `loginrestrictions` subroutine determines if the user specified by the `Name` parameter is allowed to access the system. The `Mode` parameter gives the mode of account usage and the `Tty` parameter defines the terminal used for access. The `Msg` parameter returns an informational message explaining why the `loginrestrictions` subroutine failed.

This subroutine is unsuccessful if any of the following conditions exists:

- The user's account has expired as defined by the `expires` user attribute.
- The user's account has been locked as defined by the `account_locked` user attribute.
- The user attempted too many unsuccessful logins as defined by the `loginretries` user attribute.
- The user is not allowed to access the given terminal as defined by the `tty` user attribute.
- The user is not allowed to access the system at the present time as defined by the `logintimes` user attribute.
- The `Mode` parameter is set to the `S_LOGIN` value or the `S_RLOGIN` value, and too many users are logged in as defined by the `maxlogins` system attribute.
- The `Mode` parameter is set to the `S_LOGIN` value and the user is not allowed to log in as defined by the `login` user attribute.
- The `Mode` parameter is set to the `S_RLOGIN` value and the user is not allowed to log in from the network as defined by the `rlogin` user attribute.
- The `Mode` parameter is set to the `S_SU` value and other users are not allowed to use the `su` command as defined by the `su` user attribute, or the group ID of the current process cannot use the `su` command to switch to this user as defined by the `sugroups` user attribute.
- The `Mode` parameter is set to the `S_DAEMON` value and the user is not allowed to run processes from the `cron` or `src` subsystem as defined by the `daemon` user attribute.
- The terminal is locked as defined by the `locktime` port attribute.
- The user cannot use the terminal to access the system at the present time as defined by the `logintimes` port attribute.
- The user is not the root user and the `/etc/nologin` file exists.

Note: The `loginrestrictions` subroutine is not safe in a multi-threaded environment. To use `loginrestrictions` in a threaded application, the application must keep the integrity of each thread.

Parameters

**Name**
Specifies the user's login name whose account is to be validated.

**Mode**
Specifies the mode of usage. Valid values as defined in the `login.h` file are listed below. The `Mode` parameter has a value of 0 or one of the following values:

- **S_LOGIN**
  Verifies that local logins are permitted for this account.

- **S_SU**
  Verifies that the `su` command is permitted and the current process has a group ID that can invoke the `su` command to switch to the account.

- **S_DAEMON**
  Verifies that the account can invoke daemon or batch programs through the `src` or `cron` subsystems.

- **S_RLOGIN**
  Verifies that the account can be used for remote logins through the `rlogin` or `telnet` programs.

**Tty**
Specifies the terminal of the originating activity. If this parameter is a null pointer or a null string, no tty origin checking is done.

**Msg**
Returns an informative message indicating why the `loginrestrictions` subroutine failed. Upon return, the value is either a pointer to a valid string within memory allocated storage or a null value. If a message is displayed, it is provided based on the user interface.
Security

Access Control: The calling process must have access to the account information in the user database and the port information in the port database.

File Accessed:

<table>
<thead>
<tr>
<th>Mode</th>
<th>Files</th>
</tr>
</thead>
<tbody>
<tr>
<td>r</td>
<td>/etc/security/user</td>
</tr>
<tr>
<td>r</td>
<td>/etc/security/login.cfg</td>
</tr>
<tr>
<td>r</td>
<td>/etc/security/portlog</td>
</tr>
<tr>
<td>r</td>
<td>/etc/passwd</td>
</tr>
</tbody>
</table>

Return Values

If the account is valid for the specified usage, the `loginrestrictions` subroutine returns a value of 0. Otherwise, a value of -1 is returned, the `errno` global value is set to the appropriate error code, and the `Msg` parameter returns an informative message explaining why the specified account usage is invalid.

Error Codes

The `loginrestrictions` subroutine fails if one or more of the following values is true:

- **ENOENT**: The user specified does not have an account.
- **ESTALE**: The user’s account is expired.
- **EPERM**: The user’s account is locked, the specified terminal is locked, the user has had too many unsuccessful login attempts, or the user cannot log in because the `/etc/nologin` file exists.
- **EACCES**: One of the following conditions exists:
  - The specified terminal does not have access to the specified account.
  - The `Mode` parameter is the `S_SU` value and the current process is not permitted to use the `su` command to access the specified user.
  - Access to the account is not permitted in the specified mode.
  - Access to the account is not permitted at the current time.
  - Access to the system with the specified terminal is not permitted at the current time.
- **EAGAIN**: The `Mode` parameter is either the `S_LOGIN` value or the `S_RLOGIN` value, and all the user licenses are in use.
- **EINVAL**: The `Mode` parameter has a value other than `S_LOGIN`, `S_SU`, `S_DAEMON`, `S_RLOGIN`, or 0.

Related Information

The `authenticate` subroutine, `getpcred` subroutine, `getenv` subroutine, `loginsuccess` subroutine, `setpcred` subroutine, `setenv` subroutine.

The `cron` daemon.

The `login` command, `rlogin` command, `telnet`, `tn`, or `tn3270` command, `su` command.

List of Security and Auditing Subroutines

Subroutines, Example Programs, and Libraries

AIX 5L Version 5.3 General Programming Concepts: Writing and Debugging Programs.
loginrestrictionsx Subroutine

Purpose
Determines, in multiple methods, if a user is allowed to access the system.

Library
Security Library (libc.a)

Syntax
#include <usersec.h>
#include <login.h>

int loginrestrictionsx (Name, Mode, Tty, Message, State)
char *Name;
int Mode;
char *Tty;
char **Message;
char **State;

Description
The loginrestrictionsx subroutine determines if the user specified by the Name parameter is allowed to access the system. The Mode parameter gives the mode of account usage, and the Tty parameter defines the terminal used for access. The Msg parameter returns an informational message explaining why the loginrestrictionsx subroutine failed. The user's SYSTEM attribute determines the administrative domains to examine for permission.

The State parameter contains information about the login restrictions for the user. A call to the authenticatex subroutine will not use an administrative domain for authentication if an earlier call to loginrestrictionsx indicated that the user was unable to log in using that administrative domain's authentication data. The result is that administrative domains that are used for authentication must permit the user to log in. The State parameter returned by loginrestrictionsx can be used as input to a subsequent call to the authenticatex subroutine.

This subroutine is unsuccessful if any of the following conditions exists:
- The user's account has been locked as defined by the account_locked user attribute.
- The user's account has expired as defined by the expires user attribute.
- The Mode parameter is set to the S_LOGIN value or the S_RLOGIN value, and too many users are logged in as defined by the maxlogins system attribute.
- The Mode parameter is not set to the S_SU or S_DAEMON value, and the user is not allowed to log in to the current host as defined by the user's hostallowedlogin and hostdeniedlogin attributes.
- The user is not allowed to access the system at the present time as defined by the logintimes user attribute.
- The user attempted too many unsuccessful logins as defined by the loginretries user attribute.
- The user is not allowed to access the given terminal or network protocol as defined by the ttys user attribute. This test is not performed when the Mode parameter is set to the S_DAEMON value.
- The Mode parameter is set to the S_LOGIN value, and the user is not allowed to log in as defined by the login user attribute.
- The Mode parameter is set to the S_RLOGIN value and the user is not allowed to log in from the network as defined by the rlogin user attribute.
- The Mode parameter is set to the \texttt{S\_SU} value, and other users are not allowed to use the \texttt{su} command as defined by the \texttt{su} user attribute; or, the group ID of the current process cannot use the \texttt{su} command to switch to this user as defined by the \texttt{sugroups} user attribute.
- The \textit{Mode} parameter is set to the \texttt{S\_DAEMON} value, and the user is not allowed to run processes from the \texttt{cron} or \texttt{src} subsystem as defined by the \texttt{daemon} user attribute.
- The terminal is locked as defined by the \texttt{locktime} port attribute.
- The user cannot use the terminal to access the system at the present time as defined by the \texttt{logintimes} port attribute.
- The user is not the root user, and the \texttt{/etc/nologin} file exists.

Additional restrictions can be enforced by loadable authentication modules for any administrative domain used in the user’s \texttt{SYSTEM} attribute.

**Parameters**

*Name* Specifies the user’s login name whose account is to be validated.

*Mode* Specifies the mode of usage. The valid values in the following list are defined in the \texttt{login.h} file. The \textit{Mode} parameter has a value of 0 or one of the following values:

- \texttt{S\_LOGIN} Verifies that local logins are permitted for this account.
- \texttt{S\_SU} Verifies that the \texttt{su} command is permitted and the current process has a group ID that can invoke the \texttt{su} command to switch to the account.
- \texttt{S\_DAEMON} Verifies that the account can invoke daemon or batch programs through the \texttt{src} or \texttt{cron} subsystems.
- \texttt{S\_RLOGIN} Verifies that the account can be used for remote logins through the \texttt{rlogind} or \texttt{telnetd} programs.

*Tty* Specifies the terminal of the originating activity. If this parameter is a null pointer or a null string, no tty origin checking is done. The \textit{Tty} parameter can also have the value \texttt{RSH} or \texttt{REXEC} to indicate that the caller is the \texttt{rsh} or \texttt{rexec} command.

*Message* Returns an informative message indicating why the \texttt{loginrestrictionsx} subroutine failed. Upon return, the value is either a pointer to a valid string within memory-allocated storage or a null value. If a message is displayed, it is provided based on the user interface.

*State* Points to a pointer that the \texttt{loginrestrictionsx} subroutine allocates memory for and fills in. The \textit{State} parameter can also be the result of an earlier call to the \texttt{authenticatex} subroutine. The \textit{State} parameter contains information about the results of the \texttt{loginrestrictionsx} subroutine for each term in the user’s \texttt{SYSTEM} attribute. The calling application is responsible for freeing this memory when it is no longer needed for a subsequent call to the \texttt{authenticatex}, \texttt{passwdexpiredx}, or \texttt{chpassx} subroutines.

**Security**

Access Control: The calling process must have access to the account information in the user database and the port information in the port database.

Files accessed:

<table>
<thead>
<tr>
<th>Mode</th>
<th>File</th>
</tr>
</thead>
<tbody>
<tr>
<td>r</td>
<td>\texttt{/etc/security/user}</td>
</tr>
<tr>
<td>r</td>
<td>\texttt{/etc/security/login.cfg}</td>
</tr>
<tr>
<td>r</td>
<td>\texttt{/etc/security/portlog}</td>
</tr>
<tr>
<td>r</td>
<td>\texttt{/etc/passwd}</td>
</tr>
</tbody>
</table>
Return Values
If the account is valid for the specified usage, the `loginrestrictions` subroutine returns a value of 0. Otherwise, a value of -1 is returned, the `errno` global value is set to the appropriate error code, and the `Message` parameter returns an informative message explaining why the specified account usage is invalid.

Error Codes
If the `loginrestrictions` subroutine fails if one of the following values is true:

**EACCESS**
One of the following conditions exists:
- The specified terminal does not have access to the specified account.
- The `Mode` parameter is the `S_SU` value, and the current process is not permitted to use the `su` command to access the specified user.
- Access to the account is not permitted in the specified mode.
- Access to the account is not permitted at the current time.
- Access to the system with the specified terminal is not permitted at the current time.

**EAGAIN**
The `Mode` parameter is either the `S_LOGIN` value or the `S_RLOGIN` value, and all the user licenses are in use.

**EINVAL**
The `Mode` parameter has a value other than `S_LOGIN`, `S_SU`, `S_DAEMON`, `S_RLOGIN`, or 0.

**ENOENT**
The user specified does not have an account.

**EPERM**
The user’s account is locked, the specified terminal is locked, the user has had too many unsuccessful login attempts, or the user cannot log in because the `/etc/nologin` file exists.

**ESTALE**
The user’s account is expired.

Related Information

The `cron` Daemon.

The `login Command`, `rlogin Command`, `telnet, tn, or tn3270 Command`, `su Command`.

List of Security and Auditing Subroutines in AIX 5L Version 5.3 General Programming Concepts: Writing and Debugging Programs.

Subroutines, Example Programs, and Libraries in AIX 5L Version 5.3 General Programming Concepts: Writing and Debugging Programs.

loginsuccess Subroutine

**Purpose**
Records a successful log in.

**Library**
Security Library (`libc.a`)
Syntax

```c
#include <usersec.h>
int loginsuccess (User, Host, Tty, Msg)
    char * User;
    char * Host;
    char * Tty;
    char ** Msg;
```

Note: This subroutine is not thread-safe.

Description

The `loginsuccess` subroutine performs the processing necessary when a user successfully logs into the system. This subroutine updates the following attributes in the `/etc/security/lastlog` file for the specified user:

- `time_last_login`: Contains the current time.
- `tty_last_login`: Contains the value specified by the `Tty` parameter.
- `host_last_login`: Contains the value specified by the `Host` parameter or the local host name if the `Host` parameter is a null value.
- `unsuccessful_login_count`: Indicates the number of unsuccessful login attempts. The `loginsuccess` subroutine resets this attribute to a value of 0.

Additionally, a login success audit record is cut to indicate in the audit trail that this user has successfully logged in.

A message is returned in the `Msg` parameter that indicates the time, host, and port of the last successful and unsuccessful login. The number of unsuccessful login attempts since the last successful login is also provided to the user.

Parameters

- **User**: Specifies the login name of the user who has successfully logged in.
- **Host**: Specifies the name of the host from which the user logged in. If the `Host` parameter is a null value, the name of the local host is used.
- **Tty**: Specifies the name of the terminal which the user used to log in.
- **Msg**: Returns a message indicating the delete time, host, and port of the last successful and unsuccessful logins. The number of unsuccessful login attempts since the last successful login is also provided. Upon return, the value is either a pointer to a valid string within memory allocated storage or a null pointer. It is the responsibility of the calling program to `free()` the returned storage.

Security

Access Control: The calling process must have access to the account information in the user database.

File Accessed:

<table>
<thead>
<tr>
<th>Mode</th>
<th>File</th>
</tr>
</thead>
<tbody>
<tr>
<td>rw</td>
<td><code>/etc/security/lastlog</code></td>
</tr>
</tbody>
</table>

Auditing Events:

<table>
<thead>
<tr>
<th>Event</th>
<th>Information</th>
</tr>
</thead>
<tbody>
<tr>
<td>USER_Login</td>
<td><code>username</code></td>
</tr>
</tbody>
</table>
Return Values
Upon successful completion, the **loginsuccess** subroutine returns a value of 0. Otherwise, a value of -1 is returned and the **errno** global value is set to indicate the error.

Error Codes
The **loginsuccess** subroutine fails if one or more of the following values is true:

- **ENOENT**  The specified user does not exist.
- **EACCES**  The current process does not have write access to the user database.
- **EPERM**  The current process does not have permission to write an audit record.

Error Codes
The **loginsuccess** subroutine fails if one or more of the following values is true:

- **ENOENT**  The specified user does not exist.
- **EACCES**  The current process does not have write access to the user database.
- **EPERM**  The current process does not have permission to write an audit record.

Related Information
The **authenticate** subroutine, **getpcred** subroutine, **getpenv** subroutine, **loginfailed** subroutine, **loginrestrictions** subroutine, **setpcred** subroutine, **setpenv** subroutine.

List of Security and Auditing Services in AIX 5L Version 5.3 General Programming Concepts: Writing and Debugging Programs.

Subroutines Overview in AIX 5L Version 5.3 General Programming Concepts: Writing and Debugging Programs.

### Ipar_get_info Subroutine

**Purpose**
Retrieves the calling partition’s characteristics.

**Syntax**
```c
#include <sys/dr.h>

int Ipar_get_info (command, lparinfo, bufsize);
int command;
void *lparinfo;
size_t bufsize;
```

**Description**
The **Ipar_get_info** subroutine retrieves LPAR and Micro-Partitioning attributes of both low-frequency use and high-frequency use. Because the low-frequency attributes, as defined in the **Ipar_info_format1** structure, are static in nature, a reboot is required to effect any change. The high-frequency attributes, as defined in the **Ipar_info_format2** structure, can be changed dynamically at any time either by the platform or through DLPAR procedures. The latter provides a mechanism for notifying applications of changes. The signature of this system call, its parameter types, and the order of the member fields in both the **Ipar_info_format1** and **Ipar_info_format2** structures are specific to the AIX platform.

To see the complete structures of **Ipar_info_format1** and **Ipar_info_format2**, see the **dr.h** header file.

The **Ipar_get_info** system call provides information about the operating system environment, including the following:

- Type of partition: dedicated processor partition or micro-partition
- Type of micro-partition: capped or uncapped
- Variable capacity weight of micro-partition

750  Technical Reference, Volume 1: Base Operating System and Extensions
- Partition name and number
- SMT-capable partition
- SMT-enabled partition
- Minimum, desired, online, and maximum number of virtual processors
- Minimum, online, and maximum number of logical processors
- Minimum, desired, online, and maximum entitled processor capacity
- Minimum, desired, online (megabytes), and maximum number of logical memory blocks (LMBs)
- Maximum number of potential installed physical processors in the server, including unlicensed and potentially hot-pluggable
- Number of active licensed installed physical processors in the server
- Number of processors in the shared processor pool

Parameters

command
- Specifies whether the user wants format1 or format2 details.

lparinfo
- Pointer to the user-allocated buffer that is passed in.

bufsize
- Size of the structure that is passed in.

Return Values

Upon success, the lpar_get_info subroutine returns a value of 0. Upon failure, a value of -1 is returned, and errno is set to indicate the appropriate error.

Error Codes

EFAULT
- Buffer size is smaller than expected.
EINVAL
- Invalid input parameter.
ENOTSUP
- The platform does not support this operation.

Related Information

The kpar_get_info kernel service.

lpar_set_resources Subroutine

Purpose

Modifies the calling partition's characteristics.

Library

Standard C Library (lib.c)

Syntax

#include <sys/dr.h>

int lpar_set_resources ( lpar_resource_id lpar_resource )
int lpar_resource_id;
void *lpar_resource;

Description

The lpar_set_resources subroutine modifies the configuration attributes (dynamic resources) on a current partition indicated by the lpar_resource_id. The pointer to a value of the dynamic resource indicated by
lpar_resource_id is passed to this call in lpar_resource. This subroutine modifies one partition dynamic resource at a time. To reconfigure multiple resources, multiple calls must be made. The following resources for the calling partition can be modified:

- Entitled Capacity
- Variable Capacity Weight
- Number of online virtual CPUs
- Number of available memory in MB

These resource IDs are defined in the <sys/dr.h> header file. In order to modify the Entitled Capacity and Variable Capacity Weight attributes, ensure that the current partition is an SPLPAR partition; otherwise, an error is returned. The lpar_set_resources subroutine can only be called in a process owned by a root user (super user) or a user with the CAP_EWLM_AGENT capability; otherwise, an error is returned.

Parameters

lpar_resource_id
Identifies the dynamic resource whose value is being changed.

lpar_resource
Pointer to a new value of the dynamic resource identified by the lpar_resource_id.

Security

The lpar_set_resources subroutine can only be called in a process owned by a root user (super user) or a user with the CAP_EWLM_AGENT capability.

Return Values

Upon success, the lpar_set_resources subroutine returns a value of 0. Upon failure, a negative value is returned, and errno is set to the appropriate error.

Error Codes

EINVAL
Invalid configuration parameters.

EPERM
Insufficient authority.

EEXIST
Resource already exists.

EBUSY
Resource is busy.

EAGAIN
Resource is temporarily unavailable.

ENOMEM
Resource allocation failed.

ENOTRETRY
Resource is not ready.

ENOTSUP
Operation is not supported.

EFAULT/EIO
Operation failed because of an I/O error.

EINPROGRESS
Operation in progress.

ENXIO
Resource is not available.

ERANGE
Parameter value is out of range.

All others
Internal error.

lrint, lrintf, or lrintl Subroutine

Purpose

Rounds to nearest integer value using the current rounding direction.

Syntax

#include <math.h>

long lrint( []

double \( x \);
long lrintf (\( x \))
float \( x \);
long lrintl (\( x \))
long double \( x \);

**Description**
The `lrint`, `lrintf`, and `lrintl` subroutines round the \( x \) parameter to the nearest integer value, rounding according to the current rounding direction.

An application wishing to check for error situations should set the `errno` global variable to zero and call `feclearexcept(FE_ALL_EXCEPT)` before calling these subroutines. Upon return, if `errno` is nonzero or `fetestexcept(FE_INVALID | FE_DIVBYZERO | FE_OVERFLOW | FE_UNDERFLOW)` is nonzero, an error has occurred.

**Parameters**

\( x \)  
Specifies the value to be rounded.

**Return Values**
Upon successful completion, the `lrint`, `lrintf`, and `lrintl` subroutines return the rounded integer value.

If \( x \) is NaN, a domain error occurs and an unspecified value is returned.

If \( x \) is +Inf, a domain error occurs and an unspecified value is returned.

If \( x \) is -Inf, a domain error occurs and an unspecified value is returned.

If the correct value is positive and too large to represent as a long, a domain error occurs and an unspecified value is returned.

If the correct value is negative and too large to represent as a long, a domain error occurs and an unspecified value is returned.

**Related Information**

“feclearexcept Subroutine” on page 262, “fetestexcept Subroutine” on page 270, and “lrint, lrintf, or lrintl Subroutine” on page 719.

**math.h** in AIX 5L Version 5.3 Files Reference.

---

**lround, lroundf, or lroundl Subroutine**

**Purpose**
Rounds to the nearest integer value.

**Syntax**

```c
#include <math.h>

long lround (\( x \))
long lroundf (\( x \))
long lroundl (\( x \))
```
float x;
long lroundl (x)
long double x;

Description
The **lround**, **lroundf**, and **lroundl** subroutines round the \( x \) parameter to the nearest integer value, rounding halfway cases away from zero, regardless of the current rounding direction.

An application wishing to check for error situations should set the **errno** global variable to zero and call **feclearexcept(FE_ALL_EXCEPT)** before calling these subroutines. Upon return, if **errno** is nonzero or **fetestexcept(FE_INVALID | FE_DIVBYZERO | FE_OVERFLOW | FE_UNDERFLOW)** is nonzero, an error has occurred.

Parameters
\( x \) Specifies the value to be rounded.

Return Values
Upon successful completion, the **lround**, **lroundf**, and **lroundl** subroutines return the rounded integer value.

If \( x \) is NaN, a domain error occurs and an unspecified value is returned.

If \( x \) is +\( \infty \), a domain error occurs and an unspecified value is returned.

If \( x \) is -\( \infty \), a domain error occurs and an unspecified value is returned.

If the correct value is positive and too large to represent as a **long**, a domain error occurs and an unspecified value is returned.

If the correct value is negative and too large to represent as a **long**, a domain error occurs and an unspecified value is returned.

Related Information
"**feclearexcept Subroutine**" on page 262, "**fetestexcept Subroutine**" on page 270, and "**lround, lroundf, or lroundl Subroutine**" on page 720.

**math.h** in **AIX 5L Version 5.3 Files Reference**.

Isearch or Ifind Subroutine

**Purpose**
Performs a linear search and update.

**Library**
Standard C Library (**libc.a**)
Syntax

```c
void *lsearch (Key, Base, NumberOfElementsPointer, Width, ComparisonPointer) const
void *Key;
void *Base;
size_t Width, *NumberOfElementsPointer;
int (*ComparisonPointer) (const void*, const void*);
void *lfind (Key, Base, NumberOfElementsPointer, Width, ComparisonPointer) const
void *Key, Base;
size_t Width, *NumberOfElementsPointer;
int (*ComparisonPointer) (const void*, const void*);
```

Description

Warning: Undefined results can occur if there is not enough room in the table for the lsearch subroutine to add a new item.

The lsearch subroutine performs a linear search.

The algorithm returns a pointer to a table where data can be found. If the data is not in the table, the program adds it at the end of the table.

The lfind subroutine is identical to the lsearch subroutine, except that if the data is not found, it is not added to the table. In this case, a NULL pointer is returned.

The pointers to the Key parameter and the element at the base of the table should be of type pointer-to-element and cast to type pointer-to-character. The value returned should be cast into type pointer-to-element.

The comparison function need not compare every byte; therefore, the elements can contain arbitrary data in addition to the values being compared.

Parameters

- **Base**
  - Points to the first element in the table.
- **ComparisonPointer**
  - Specifies the name (that you supply) of the comparison function (strcmp, for example). It is called with two parameters that point to the elements being compared.
- **Key**
  - Specifies the data to be sought in the table.
- **NumberOfElementsPointer**
  - Points to an integer containing the current number of elements in the table. This integer is incremented if the data is added to the table.
- **Width**
  - Specifies the size of an element in bytes.

The comparison function compares its parameters and returns a value as follows:

- If the first parameter equals the second parameter, the ComparisonPointer parameter returns a value of 0.
- If the first parameter does not equal the second parameter, the ComparisonPointer parameter returns a value of 1.

Return Values

If the sought entry is found, both the lsearch and lfind subroutines return a pointer to it. Otherwise, the lfind subroutine returns a null pointer and the lsearch subroutine returns a pointer to the newly added element.
Related Information
The `bsearch` subroutine, `hsearch` subroutine, `qsort` subroutine, `tsearch` subroutine.


Searching and Sorting Example Program and Subroutines Overview in AIX 5L Version 5.3 General Programming Concepts: Writing and Debugging Programs.

**Iseek, Ilseek or Iseek64 Subroutine**

**Purpose**
Moves the read-write file pointer.

**Library**
Standard C Library (`libc.a`)

**Syntax**

```c
off_t lseek (FileDescriptor, Offset, Whence);
int FileDescriptor, whence;
off_t Offset;

offset_t llseek (FileDescriptor, Offset, Whence);
int FileDescriptor, whence;
Offset;

off64_t lseek64 (FileDescriptor, Offset, Whence);
int FileDescriptor, whence;
off64_t Offset;
```

**Description**
The `Iseek`, `Ilseek`, and `Iseek64` subroutines set the read-write file pointer for the open file specified by the `FileDescriptor` parameter. The `Iseek` subroutine limits the `Offset` to `OFF_MAX`.

In the large file enabled programming environment, `Iseek` subroutine is redefined to `Iseek64`.

If the `FileDescriptor` parameter refers to a shared memory object, the `Iseek` subroutine fails with `EINVAL`.

**Parameters**

- `FileDescriptor` Specifies a file descriptor obtained from a successful `open` or `fcntl` subroutine.
- `Offset` Specifies a value, in bytes, that is used in conjunction with the `Whence` parameter to set the file pointer. A negative value causes seeking in the reverse direction.
- `Whence` Specifies how to interpret the `Offset` parameter by setting the file pointer associated with the `FileDescriptor` parameter to one of the following variables:

  - `SEEK_SET` Sets the file pointer to the value of the `Offset` parameter.
  - `SEEK_CUR` Sets the file pointer to its current location plus the value of the `Offset` parameter.
  - `SEEK_END` Sets the file pointer to the size of the file plus the value of the `Offset` parameter.
Return Values
Upon successful completion, the resulting pointer location, measured in bytes from the beginning of the file, is returned. If either the lseek or llseek subroutines are unsuccessful, a value of -1 is returned and the errno global variable is set to indicate the error.

Error Codes
The lseek or llseek subroutines are unsuccessful and the file pointer remains unchanged if any of the following are true:

- EBADF: The FileDescriptor parameter is not an open file descriptor.
- EINVAL: The resulting offset would be greater than the maximum offset allowed for the file or device associated with FileDescriptor. The lseek subroutine was used with a file descriptor obtained from a call to the shm_open subroutine.
- EINVAL: Whence is not one of the supported values.
- EOVERFLOW: The resulting offset is larger than can be returned properly.
- ESPIPE: The FileDescriptor parameter is associated with a pipe (FIFO) or a socket.

Files
/usr/include/unistd.h defines standard macros, data types and subroutines.

Related Information
The fcntl subroutine, lseek, rewind, ftell, fgetpos, or fsetpos subroutine, fseek, fseeko, fseeko64, rewind, ftell, ftello, ftello64, fgetpos, fgetpos64, fsetpos, or fsetpos64 subroutine, open, openx, or creat subroutine, read, readx, readv, or readvx subroutine, write, writex, writev, or writevx subroutine.

File Systems and Directories in AIX 5L Version 5.3 General Programming Concepts: Writing and Debugging Programs.

Ivm_querylv Subroutine
Purpose
Queries a logical volume and returns all pertinent information.

Library
Logical Volume Manager Library (liblvm.a)

Syntax
#include <lvm.h>

int lvm_querylv (LV_ID, QueryLV, PVName);
struct lv_id *LV_ID;
struct querylv **QueryLV;
char *PVName;
Description

**Note:** The `lvm_querylv` subroutine uses the `sysconfig` system call, which requires root user authority, to query and update kernel data structures describing a volume group. You must have root user authority to use the `lvm_querylv` subroutine.

The `lvm_querylv` subroutine returns information for the logical volume specified by the `LV_ID` parameter.

The `querylv` structure, found in the `lvm.h` file, is defined as follows:

```c
struct querylv {
    char lvname[LVM_NAMESIZ];
    struct unique_id vg_id;
    long maxsize;
    long mirror_policy;
    long lv_state;
    long currentsize;
    long ppsize;
    long permissions;
    long bb_relocation;
    long write_verify;
    long mirwrt_consist;
    long open_close;
    struct pp *mirrors[LVM_NUMCOPIES];
    unsigned int stripe_exp;
    unsigned int striping_width;
}
struct pp {
    struct unique_id pv_id;
    long lp_num;
    long pp_num;
    long ppstate;
}
```

### Field Description

- **lvname**
  Specifications the special file name of the logical volume and can be either the full path name or a single file name that must reside in the `/dev` directory (for example, `rhd1`). All name fields must be null-terminated strings of from 1 to `LVM_NAMESIZ` bytes, including the null byte. If a raw or character device is not specified for the `lvname` field, the Logical Volume Manager (LVM) will add an `r` to the file name to have a raw device name. If there is no raw device entry for this name, the LVM will return the `LVM_NOTCHARDEV` error code.

- **vg_id**
  Specifies the unique ID of the volume group that contains the logical volume.

- **maxsize**
  Indicates the maximum size in logical partitions for the logical volume and must be in the range of 1 to `LVM_MAXLPS`.

- **mirror_policy**
  Specifies how the physical copies are written. The `mirror_policy` field should be either `LVM_SEQUENTIAL` or `LVM_PARALLEL` to indicate how the physical copies of a logical partition are to be written when there is more than one copy.

- **lv_state**
  Specifies the current state of the logical volume and can have any of the following bit-specific values ORed together:

  - **LVM_LVDEFINED**
    The logical volume is defined.

  - **LVM_LVSTALE**
    The logical volume contains stale partitions.

- **currentsize**
  Indicates the current size in logical partitions of the logical volume. The size, in bytes, of every physical partition is $2^{ppsize}$.

- **ppsize**
  Specifies the size of the physical partitions of all physical volumes in the volume group.
Table 1: Logical Volume Properties

<table>
<thead>
<tr>
<th>Field</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>permissions</td>
<td>Specifies the permission assigned to the logical volume and can be one of the following values:</td>
</tr>
<tr>
<td></td>
<td><strong>LVM_RDONLY</strong>&lt;br&gt;Access to this logical volume is read only.</td>
</tr>
<tr>
<td></td>
<td><strong>LVM_RDWR</strong>&lt;br&gt;Access to this logical volume is read/write.</td>
</tr>
<tr>
<td></td>
<td><strong>LVM_NORELOC</strong>&lt;br&gt;Bad blocks will not be relocated.</td>
</tr>
<tr>
<td></td>
<td><strong>LVM_RELOC</strong>&lt;br&gt;Bad blocks will be relocated.</td>
</tr>
<tr>
<td>bb_relocation</td>
<td>Specifies if bad block relocation is desired and is one of the following values:</td>
</tr>
<tr>
<td></td>
<td><strong>LVM_NOVERIFY</strong>&lt;br&gt;Write verification is not performed for this logical volume.</td>
</tr>
<tr>
<td></td>
<td><strong>LVM_VERIFY</strong>&lt;br&gt;Write verification is performed on all writes to the logical volume.</td>
</tr>
<tr>
<td>write_verify</td>
<td>Specifies if write verification for the logical volume is desired and returns one of the following values:</td>
</tr>
<tr>
<td>mirwrt_consist</td>
<td>Indicates whether mirror-write consistency recovery will be performed for this logical volume.</td>
</tr>
<tr>
<td></td>
<td>The LVM always ensures data consistency among mirrored copies of a logical volume during normal I/O processing. For every write to a logical volume, the LVM generates a write request for every mirror copy. A problem arises if the system crashes in the middle of processing a mirrored write (before all copies are written). If mirror write consistency recovery is requested for a logical volume, the LVM keeps additional information to allow recovery of these inconsistent mirrors. Mirror write consistency recovery should be performed for most mirrored logical volumes. Logical volumes, such as page space, that do not use the existing data when the volume group is re-varied on do not need this protection.</td>
</tr>
<tr>
<td></td>
<td>Values for the <strong>mirwrt_consist</strong> field are:</td>
</tr>
<tr>
<td></td>
<td><strong>LVM_CONSIST</strong>&lt;br&gt;Mirror-write consistency recovery will be done for this logical volume.</td>
</tr>
<tr>
<td></td>
<td><strong>LVM_NOCONSIST</strong>&lt;br&gt;Mirror-write consistency recovery will not be done for this logical volume.</td>
</tr>
<tr>
<td>open_close</td>
<td>Specifies if the logical volume is opened or closed. Values for this field are:</td>
</tr>
<tr>
<td></td>
<td><strong>LVM_QLV_NOTOPEN</strong>&lt;br&gt;The logical volume is closed.</td>
</tr>
<tr>
<td></td>
<td><strong>LVM_QLVOPEN</strong>&lt;br&gt;The logical volume is opened by one or more processes.</td>
</tr>
<tr>
<td>mirrors</td>
<td>Specifies an array of pointers to partition map lists (physical volume id, logical partition number, physical partition number, and physical partition state for each copy of the logical partitions for the logical volume). The ppstate field can be <strong>LVM_PPFREE</strong>, <strong>LVM_PPALLOC</strong>, or <strong>LVM_PPSTALE</strong>. If a logical partition does not contain any copies, its pv_id, lp_num, and pp_num fields will contain zeros.</td>
</tr>
<tr>
<td></td>
<td>The logical volume is not striped, the <strong>stripe_exp</strong> field is 0.</td>
</tr>
<tr>
<td>stripe_exp</td>
<td>Specifies the log base 2 of the logical volume strip size (the strip size multiplied by the number of disks in an array equals the stripe size). For example, $2^{20}$ is 1048576 (that is, 1 MB). Therefore, if the strip size is 1 MB, the <strong>stripe_exp</strong> field is 20. If the logical volume is not striped, the <strong>stripe_exp</strong> field is 0.</td>
</tr>
<tr>
<td>stripe_width</td>
<td>Specifies the number of disks that form the striped logical volume. If the logical volume is not striped, the <strong>striping_width</strong> field is 0.</td>
</tr>
</tbody>
</table>
The **PVName** parameter enables the user to query from a volume group descriptor area on a specific physical volume instead of from the Logical Volume Manager’s (LVM) most recent, in-memory copy of the descriptor area. This method should only be used if the volume group is varied off.

**Note:** The data returned is not guaranteed to be the most recent or correct, and it can reflect a back-level descriptor area.

The **PVName** parameter should specify either the full path name of the physical volume that contains the descriptor area to query, or a single file name that must reside in the /dev directory (for example, **rhdisk1**). This parameter must be a null-terminated string between 1 and **LVM_NAMESIZ** bytes, including the null byte, and must represent a raw device entry. If a raw or character device is not specified for the **PVName** parameter, the LVM adds an r to the file name to have a raw device name. If there is no raw device entry for this name, the LVM returns the **LVM_NOTCHARDEV** error code.

If a **PVName** parameter is specified, only the **minor_num** field of the **LV_ID** parameter need be supplied. The LVM fills in the **vg_id** field and returns it to the user. If the user wishes to query from the LVM’s in-memory copy, the **PVName** parameter should be set to null. When using this method of query, the volume group must be varied on, or an error is returned.

**Note:** As long as the **PVName** parameter is not null, the LVM will attempt a query from a physical volume and **not** from its in-memory copy of data.

In addition to the **PVName** parameter, the caller passes the ID of the logical volume to be queried (**LV_ID** parameter) and the address of a pointer to the **querylv** structure, specified by the **QueryLV** parameter. The LVM separately allocates the space needed for the **querylv** structure and the struct **pp** arrays, and returns the **querylv** structure’s address in the pointer variable passed in by the user. The user is responsible for freeing the space by first freeing the struct **pp** pointers in the **mirrors** array and then freeing the **querylv** structure.

**Attention:** To prevent corruption when there are many **pp** arrays, the caller of **lvm_querylv** must set **QueryLV->mirrors k != NULL**.

**Parameters**

- **LV_ID**: Points to an **lv_id** structure that specifies the logical volume to query.
- **QueryLV**: Contains the address of a pointer to the **querylv** structure.
- **PVName**: Names the physical volume from which to use the volume group descriptor for the query. This parameter can also be null.

**Return Values**

If the **lvm_querylv** subroutine is successful, it returns a value of 0.

**Error Codes**

If the **lvm_querylv** subroutine does not complete successfully, it returns one of the following values:

- **LVM_ALLOCERR**: The subroutine could not allocate enough space for the complete buffer.
- **LVM_INVALID_MIN_NUM**: The minor number of the logical volume is not valid.
- **LVM_INVALID_PARAM**: A parameter passed into the routine is not valid.
- **LVM_INV_DEVENT**: The device entry for the physical volume specified by the **Pvname** parameter is not valid and cannot be checked to determine if it is raw.
- **LVM_NOTCHARDEV**: The physical volume name given does not represent a raw or character device.
- **LVM_OFFLINE**: The volume group containing the logical volume to query was offline.

If the query originates from the varied-on volume group’s current volume group descriptor area, one of the following error codes is returned:
LVM_DALVOPN The volume group reserved logical volume could not be opened.
LVM_MAPFBSY The volume group is currently locked because system management on the volume group is being done by another process.
LVM_MAPFOPN The mapped file, which contains a copy of the volume group descriptor area used for making changes to the volume group, could not be opened.
LVM_MAPFRDWR The mapped file could not be read or written.

If a physical volume name has been passed, requesting that the query originate from a specific physical volume, one of the following error codes is returned:

LVM_BADBBDIR The bad-block directory could not be read or written.
LVM_LVMRECErr The LVM record, which contains information about the volume group descriptor area, could not be read.
LVM_NOPVVGDA There are no volume group descriptor areas on the physical volume specified.
LVM_NOTVGMEM The physical volume specified is not a member of a volume group.
LVM_PVDAREAD An error occurred while trying to read the volume group descriptor area from the specified physical volume.
LVM_PVOPNERR The physical volume device could not be opened.
LVM_VGDA_BB A bad block was found in the volume group descriptor area located on the physical volume that was specified for the query. Therefore, a query cannot be done from the specified physical volume.

Related Information
List of Logical Volume Subroutines and Logical Volume Programming Overview in AIX 5L Version 5.3
General Programming Concepts: Writing and Debugging Programs.

lvm_querypv Subroutine

Purpose
Queries a physical volume and returns all pertinent information.

Library
Logical Volume Manager Library (liblvm.a)

Syntax
#include <lvm.h>

int lvm_querypv (VG_ID, PV_ID, QueryPV, PVName)
struct unique_id * VG_ID;
struct unique_id * PV_ID;
struct querypv ** QueryPV;
char * PVName;

Description

Note: The lvm_querypv subroutine uses the sysconfig system call, which requires root user authority, to query and update kernel data structures describing a volume group. You must have root user authority to use the lvm_querypv subroutine.

The lvm_querypv subroutine returns information on the physical volume specified by the PV_ID parameter.
The `querypv` structure, defined in the `lvm.h` file, contains the following fields:

```c
struct querypv {
    long ppsize;
    long pv_state;
    long pp_count;
    long alloc_ppcount;
    long pvnum_vgdas;
    struct pp_map *pp_map;
    char hotspare;
    struct unique_id pv_id;
    long freespace;
}
struct pp_map {
    long pp_state;
    struct lv_id lv_id;
    long lp_num;
    long copy;
    struct unique_id fst_alt_vol;
    long fst_alt_part;
    struct unique_id snd_alt_vol;
    long snd_alt_part;
}
```

<table>
<thead>
<tr>
<th>Field</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ppsize</td>
<td>Specifies the size of the physical partitions, which is the same for all partitions within a volume group. The size in bytes of a physical partition is $2^{ppsize}$.</td>
</tr>
<tr>
<td>pv_state</td>
<td>Contains the current state of the physical volume.</td>
</tr>
<tr>
<td>pp_count</td>
<td>Contains the total number of physical partitions on the physical volume.</td>
</tr>
<tr>
<td>alloc_ppcount</td>
<td>Contains the number of allocated physical partitions on the physical volume.</td>
</tr>
</tbody>
</table>
Field | Description
--- | ---
pp_map | Points to an array that has entries for each physical partition of the physical volume. Each entry in this array will contain the pp_state that specifies the state of the physical partition (LVM_PPFREE, LVM_PPALLOC, or LVM_PPSTALE) and the lv_id field, the ID of the logical volume that it is a member of. The pp_map array also contains the physical volume IDs (fst_alt_vol and snd_alt_vol) and the physical partition numbers (fst_alt_part and snd_alt_part) for the first and second alternate copies of the physical partition, and the logical partition number (lp_num) that the physical partition corresponds to.

If the physical partition is free (that is, not allocated), all of its pp_map fields will be zero.

fst_alt_vol | Contains zeros if the logical partition has only one physical copy.

fst_alt_part | Contains zeros if the logical partition has only one physical copy.

snd_alt_vol | Contains zeros if the logical partition has only one or two physical copies.

snd_alt_part | Contains zeros if the logical partition has only one or two physical copies.

copy | Specifies which copy of a logical partition this physical partition is allocated to. This field will contain one of the following values:

LVM_PRIMARY | Primary and only copy of a logical partition

LVM_PRIMOF2 | Primary copy of a logical partition with two physical copies

LVM_PRIMOF3 | Primary copy of a logical partition with three physical copies

LVM_SCNDOF2 | Secondary copy of a logical partition with two physical copies

LVM_SCNDOF3 | Secondary copy of a logical partition with three physical copies

LVM_TERTOF3 | Tertiary copy of a logical partition with three physical copies.

pvnum_vgdas | Contains the number of volume group descriptor areas (0, 1, or 2) that are on the specified physical volume.

hotspare | Specifies that the physical volume is a hotspare.

pv_id | Specifies the physical volume identifier.

freespace | Specifies the number of physical partitions in the volume group.

The PVName parameter enables the user to query from a volume group descriptor area on a specific physical volume instead of from the Logical Volume Manager’s (LVM) most recent, in-memory copy of the descriptor area. This method should only be used if the volume group is varied off. The data returned is not guaranteed to be most recent or correct, and it can reflect a back level descriptor area.

The PVname parameter should specify either the full path name of the physical volume that contains the descriptor area to query or a single file name that must reside in the /dev directory (for example, rhdisk1). This field must be a null-terminated string of from 1 to LVM_NAMESIZ bytes, including the null byte, and represent a raw or character device. If a raw or character device is not specified for the PVName parameter, the LVM will add an r to the file name in order to have a raw device name. If there is no raw device entry for this name, the LVM will return the LVM_NOTCHARDEV error code. If a PVName is specified, the volume group identifier, VG_ID, will be returned by the LVM through the VG_ID parameter.
passed in by the user. If the user wishes to query from the LVM in-memory copy, the *PVName* parameter should be set to null. When using this method of query, the volume group must be varied on, or an error will be returned.

**Note:** As long as the *PVName* is not null, the LVM will attempt a query from a physical volume and not from its in-memory copy of data.

In addition to the *PVName* parameter, the caller passes the *VG_ID* parameter, indicating the volume group that contains the physical volume to be queried, the unique ID of the physical volume to be queried, the *PV_ID* parameter, and the address of a pointer of the type *QueryPV*. The LVM will separately allocate enough space for the *querypv* structure and the struct *pp_map* array and return the address of the *querypv* structure in the *QueryPV* pointer passed in. The user is responsible for freeing the space by freeing the struct *pp_map* pointer and then freeing the *QueryPV* pointer.

**Parameters**

- **VG_ID** Points to a *unique_id* structure that specifies the volume group of which the physical volume to query is a member.
- **PV_ID** Points to a *unique_id* structure that specifies the physical volume to query.
- **QueryPV** Specifies the address of a pointer to a *querypv* structure.
- **PVName** Names a physical volume from which to use the volume group descriptor area for the query. This parameter can be null.

**Return Values**

The *lvm_querypv* subroutine returns a value of 0 upon successful completion.

**Error Codes**

If the *lvm_querypv* subroutine fails it returns one of the following error codes:

- **LVM_ALLOCERR** The routine cannot allocate enough space for a complete buffer.
- **LVM_INVALID_PARAM** An invalid parameter was passed into the routine.
- **LVM_INV_DEVENT** The device entry for the physical volume is invalid and cannot be checked to determine if it is raw.
- **LVM_OFFLINE** The volume group specified is offline and should be online.

If the query originates from the varied-on volume group’s current volume group descriptor area, one of the following error codes may be returned:

- **LVM_DALVOPN** The volume group reserved logical volume could not be opened.
- **LVM_MAPFBSY** The volume group is currently locked because system management on the volume group is being done by another process.
- **LVM_MAPFOPN** The mapped file, which contains a copy of the volume group descriptor area used for making changes to the volume group, could not be opened.
- **LVM_MAPFRDWR** Either the mapped file could not be read, or it could not be written.

If a physical volume name has been passed, requesting that the query originate from a specific physical volume, then one of the following error codes may be returned:

- **LVM_BADBBDIR** The bad-block directory could not be read or written.
- **LVM_LVMRECERR** The LVM record, which contains information about the volume group descriptor area, could not be read.
- **LVM_NOPPVVGDA** There are no volume group descriptor areas on this physical volume.
- **LVM_NOTCHARDEV** A device is not a raw or character device.
LVM_NOTVGMEM  The physical volume is not a member of a volume group.
LVM_PVDAREAD  An error occurred while trying to read the volume group descriptor area from the specified physical volume.
LVM_PVOPNERR  The physical volume device could not be opened.
LVM_VGDA_BB  A bad block was found in the volume group descriptor area located on the physical volume that was specified for the query. Therefore, a query cannot be done from the specified physical volume.

Related Information
List of Logical Volume Subroutines and Logical Volume Programming Overview in AIX 5L Version 5.3
General Programming Concepts: Writing and Debugging Programs.

lvm_queryvg Subroutine

Purpose
Queries a volume group and returns pertinent information.

Library
Logical Volume Manager Library (liblvm.a)

Syntax
#include <lvm.h>

int lvm_queryvg (VG_ID, QueryVG, PVName);
struct unique_id *VG_ID;
struct queryvg **QueryVG;
char *PVName;

Description
Note: The lvm_queryvg subroutine uses the sysconfig system call, which requires root user authority, to query and update kernel data structures describing a volume group. You must have root user authority to use the lvm_queryvg subroutine.

The lvm_queryvg subroutine returns information on the volume group specified by the VG_ID parameter.

The queryvg structure, found in the lvm.h file, contains the following fields:

```c
struct queryvg {
    long maxlvs;
    long ppsize;
    long freespace;
    long num_lvs;
    long num_pvds;
    long total_vgdas;
    struct lv_array *lvs;
    struct pv_array *pvds;
    short conc_capable;
    short default_mode;
    short conc_status;
    unsigned int maxpvds;
    unsigned int maxpvpvppps;
    unsigned int maxvgpps;
};
```

```c
struct pv_array {
    struct unique_id pv_id;
};
```
char state;
char res[3];
long pnum_vgdas;
}

struct lv_array {
    struct lv_id lv_id;
    char lvname[LVM_NAMESIZ];
    char state;
    char res[3];
}

<table>
<thead>
<tr>
<th>Field</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>maxlvs</td>
<td>Specifies the maximum number of logical volumes allowed in the volume group.</td>
</tr>
<tr>
<td>ppsize</td>
<td>Specifies the size of all physical partitions in the volume group. The size in bytes of each physical partitions is 2 to the power of the ppsize field.</td>
</tr>
<tr>
<td>freespace</td>
<td>Contains the number of free physical partitions in this volume group.</td>
</tr>
<tr>
<td>num_lvs</td>
<td>Indicates the number of logical volumes.</td>
</tr>
<tr>
<td>num_pvs</td>
<td>Indicates the number of physical volumes.</td>
</tr>
<tr>
<td>total_vgdas</td>
<td>Specifies the total number of volume group descriptor areas for the entire volume group.</td>
</tr>
<tr>
<td>lvs</td>
<td>Points to an array of unique IDs, names, and states of the logical volumes in the volume group.</td>
</tr>
<tr>
<td>pvs</td>
<td>Points to an array of unique IDs, states, and the number of volume group descriptor areas for each of the physical volumes in the volume group.</td>
</tr>
<tr>
<td>conc_capable</td>
<td>Indicates that the volume group was created concurrent mode capable if the value is equal to 1.</td>
</tr>
<tr>
<td>default_mode</td>
<td>The behavior of this value is undefined.</td>
</tr>
<tr>
<td>conc_status</td>
<td>Indicates that the volume group is varied on in concurrent mode.</td>
</tr>
<tr>
<td>maxpvs</td>
<td>Specifies the maximum number of physical volumes allowed in the volume group.</td>
</tr>
<tr>
<td>maxpvppps</td>
<td>Specifies the maximum number of physical partitions allowed for a physical volume in the volume group.</td>
</tr>
<tr>
<td>maxvgpps</td>
<td>Specifies the maximum number of physical partitions allowed for the entire volume group.</td>
</tr>
</tbody>
</table>

The PVName parameter enables the user to query from a descriptor area on a specific physical volume instead of from the Logical Volume Manager’s (LVM) most recent, in-memory copy of the descriptor area. This method should only be used if the volume group is varied off. The data returned is not guaranteed to be most recent or correct, and it can reflect a back level descriptor area. The Pvname parameter should specify either the full path name of the physical volume that contains the descriptor area to query or a single file name that must reside in the /dev directory (for example, rhdisk1). The name must represent a raw device. If a raw or character device is not specified for the PVName parameter, the Logical Volume Manager will add an r to the file name in order to have a raw device name. If there is no raw device entry for this name, the LVM returns the LVM_NOTCHARDEV error code. This field must be a null-terminated string of from 1 to LVM_NAMESIZ bytes, including the null byte. If a PVName is specified, the LVM will return the VG_ID to the user through the VG_ID pointer passed in. If the user wishes to query from the LVM in-memory copy, the PVName parameter should be set to null. When using this method of query, the volume group must be varied on, or an error will be returned.

Note: As long as the PVName parameter is not null, the LVM will attempt a query from a physical volume and not its in-memory copy of data.

In addition to the PVName parameter, the caller passes the unique ID of the volume group to be queried (VG_ID) and the address of a pointer to a queryvg structure. The LVM will separately allocate enough space for the queryvg structure, as well as the lv_array and pv_array structures, and return the address of the completed structure in the QueryVG parameter passed in by the user. The user is responsible for freeing the space by freeing the lv and pv pointers and then freeing the QueryVG pointer.
Parameters

VG_ID  Points to a unique_id structure that specifies the volume group to be queried.
QueryVG Specifies the address of a pointer to the queryvg structure.
PVName Specifies the name of the physical volume that contains the descriptor area to query and must be the name of a raw device.

Return Values

The lvm_queryvg subroutine returns a value of 0 upon successful completion.

Error Codes

If the lvm_queryvg subroutine fails it returns one of the following error codes:

LVM_ALLOCERR  The subroutine cannot allocate enough space for a complete buffer.
LVM_FORCEOFF  The volume group has been forcefully varied off due to a loss of quorum.
LVM_INVALID_PARAM  An invalid parameter was passed into the routine.
LVM_OFFLINE  The volume group is offline and should be online.

If the query originates from the varied-on volume group’s current volume group descriptor area, one of the following error codes may be returned:

LVM_DALVOPN  The volume group reserved logical volume could not be opened.
LVM_INV_DEVENT  The device entry for the physical volume specified by the PVName parameter is invalid and cannot be checked to determine if it is raw.
LVM_MAPFBSY  The volume group is currently locked because system management on the volume group is being done by another process.
LVM_MAPFOPN  The mapped file, which contains a copy of the volume group descriptor area used for making changes to the volume group, could not be opened.
LVM_MAPFRDWR  Either the mapped file could not be read, or it could not be written.
LVM_NOTCHARDEV  A device is not a raw or character device.

If a physical volume name has been passed, requesting that the query originate from a specific physical volume, one of the following error codes may be returned:

LVM_BADBBDIR  The bad-block directory could not be read or written.
LVM_LVMRECERR  The LVM record, which contains information about the volume group descriptor area, could not be read.
LVM_NOPVVGDA  There are no volume group descriptor areas on this physical volume.
LVM_NOTVGMEM  The physical volume is not a member of a volume group.
LVM_PVDAREAD  An error occurred while trying to read the volume group descriptor area from the specified physical volume.
LVM_PVOPNERR  The physical volume device could not be opened.
LVM_VGDA_BB  A bad block was found in the volume group descriptor area located on the physical volume that was specified for the query. Therefore, a query cannot be done from this physical volume.

Related Information

List of Logical Volume Subroutines and Logical Volume Programming Overview in AIX 5L Version 5.3 General Programming Concepts: Writing and Debugging Programs.
**Ivm_queryvgs Subroutine**

**Purpose**
Queries volume groups and returns information to online volume groups.

**Library**
Logical Volume Manager Library (*liblvm.a*)

**Syntax**
```c
#include <lvm.h>

int lvm_queryvgs (QueryVGS, Kmid);
struct queryvgs **QueryVGS;
mid_t Kmid;
```

**Description**

**Note:** The `lvm_queryvgs` subroutine uses the `sysconfig` system call, which requires root user authority, to query and update kernel data structures describing a volume group. You must have root user authority to use the `lvm_queryvgs` subroutine.

The `lvm_queryvgs` subroutine returns the volume group IDs and major numbers for all volume groups in the system that are online.

The caller passes the address of a pointer to a `queryvgs` structure, and the Logical Volume Manager (LVM) allocates enough space for the structure and returns the address of the structure in the pointer passed in by the user. The caller also passes in a `Kmid` parameter, which identifies the entry point of the logical device driver module:

```c
struct queryvgs {
    long num_vgs;
    struct {
        long major_num
        struct unique_id vg_id;
    } vgs [LVM_MAXVGS];
}
```

<table>
<thead>
<tr>
<th>Field</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>num_vgs</td>
<td>Contains the number of online volume groups on the system. The <code>vgs</code> is an array of the volume group IDs and major numbers of all online volume groups in the system.</td>
</tr>
</tbody>
</table>

**Parameters**

- **QueryVGS** Points to the `queryvgs` structure.
- **Kmid** Identifies the address of the entry point of the logical volume device driver module.

**Return Values**
The `lvm_queryvgs` subroutine returns a value of 0 upon successful completion.

**Error Codes**
If the `lvm_queryvgs` subroutine fails, it returns one of the following error codes:
- **LVM_ALLOCERR** The routine cannot allocate enough space for the complete buffer.
LVM_INVALID_PARAM  An invalid parameter was passed into the routine.

LVM_INVCONFIG  An error occurred while attempting to configure this volume group into the kernel. This error will normally result if the module ID is invalid, if the major number given is already in use, or if the volume group device could not be opened.

Related Information

List of Logical Volume Subroutines and Logical Volume Programming Overview in AIX 5L Version 5.3
General Programming Concepts: Writing and Debugging Programs.

malloc, free, realloc, calloc, mallopt, mallinfo, mallinfo_heap, alloca, valloc, or posix_memalign Subroutine

Purpose
Provides a complete set of memory allocation, reallocation, deallocation, and heap management tools.

Libraries
Berkeley Compatibility Library (libbsd.a)

Standard C Library (libc.a)

Malloc Subsystem APIs

- malloc
- free
- realloc
- calloc
- mallopt
- mallinfo
- mallinfo_heap
- alloca
- valloc
- posix_memalign

malloc

Syntax

```c
#include <stdlib.h>

void *malloc (Size)
size_t Size;
```

Description
The malloc subroutine returns a pointer to a block of memory of at least the number of bytes specified by the Size parameter. The block is aligned so that it can be used for any type of data. Undefined results occur if the space assigned by the malloc subroutine is overrun.

Parameters

- Size  Specifies the size, in bytes, of memory to allocate.
Return Values
Upon successful completion, the malloc subroutine returns a pointer to space suitably aligned for the storage of any type of object. If the size requested is 0, malloc returns NULL in normal circumstances. However, if the program was compiled with the macro LINUX_SOURCE_COMPAT defined, malloc returns a valid pointer to a space of size 0.

If the request cannot be satisfied for any reason, malloc returns NULL.

Error Codes
EINVAL 0 bytes was requested (in a mode other than Linux mode), or an internal error was detected.
ENOMEM Insufficient storage space is available to service the request.

free
Syntax
#include <stdlib.h>

void free (Pointer)
void * Pointer;

Description
The free subroutine deallocates a block of memory previously allocated by the malloc subsystem. Undefined results occur if Pointer is not an address that has previously been allocated by the malloc subsystem, or if Pointer has already been deallocated. If Pointer is NULL, no action occurs.

Parameters
Pointer Specifies a pointer to space previously allocated by the malloc subsystem.

Return Values
The free subroutine does not return a value. Upon successful completion with nonzero arguments, realloc returns a pointer to the (possibly moved) allocated space. If Size is 0 and Pointer non-NULL, no action occurs.

Error Codes
The free subroutine does not set errno.

realloc
Syntax
#include <stdlib.h>

void * realloc (Pointer, Size)
void * Pointer;
size_t Size;
Description
The realloc subroutine changes the size of the memory object pointed to by Pointer to the number of bytes specified by the Size parameter. The Pointer parameter must point to an address returned by a malloc subsystem allocation routine, and must not have been deallocated previously. Undefined results occur if Pointer does not meet these criteria.

The contents of the memory object remain unchanged up to the lesser of the old and new sizes. If the current memory object cannot be enlarged to satisfy the request, the realloc subroutine acquires a new memory object and copies the existing data to the new space. The old memory object is then freed. If no memory object can be acquired to accommodate the request, the object remains unchanged.

If Pointer is NULL, realloc is equivalent to a malloc of the same size.

If Size is 0 and Pointer is not NULL, realloc is equivalent to a free of the same size.

Parameters

| Pointer | Specifies a Pointer to space previously allocated by the malloc subsystem. |
| Size    | Specifies the new size, in bytes, of the memory object. |

Return Values
Upon successful completion with nonzero arguments, realloc returns a pointer to the (possibly moved) allocated space. If Size is 0 and Pointer non-NULL, return behavior is equivalent to free. If Pointer is NULL and Size is nonzero, return behavior is equivalent to malloc.

Error Codes
EINVAL
0 bytes was requested (in a mode other than Linux mode), or an internal error was detected.
ENOMEM
Insufficient storage space is available to service the request.

calloc

Syntax
#include <stdlib.h>

void *calloc (NumberOfElements, ElementSize);
size_t NumberOfElements;
size_t ElementSize;

Description
The calloc subroutine allocates space for an array containing NumberOfElements objects. The ElementSize parameter specifies the size of each element in bytes. After the array has been allocated, all bits are initialized to 0.

The order and contiguity of storage allocated by successive calls to the calloc subroutine is unspecified. The pointer returned points to the first (lowest) byte address of the allocated space. The allocated space is aligned so that it can be used for any type of data. Undefined results occur if the space assigned by the calloc subroutine is overrun.
Parameters

NumberofElements Specifies the number of elements in the array.
ElementSize Specifies the size, in bytes, of each element in the array.

Return Values

Upon successful completion, the calloc subroutine returns a pointer to the allocated, zero-initialized array. If the size requested is 0, calloc returns NULL in normal circumstances. However, if the program was compiled with the macro _LINUX_SOURCE_COMPAT defined, calloc returns a valid pointer to a space of size 0.

If the request cannot be satisfied for any reason, calloc returns NULL.

Error Codes

EINVAL 0 bytes was requested (in a mode other than Linux mode), or an internal error was detected.
ENOMEM Insufficient storage space is available to service the request.

mallopt

Syntax

#include <malloc.h>
#include <stdlib.h>

int mallopt (Command, Value);
int Command;
int Value;

Description

The mallopt subroutine is provided for source-level compatibility with the System V malloc subroutine. The mallopt subroutine supports the following commands:

<table>
<thead>
<tr>
<th>Command</th>
<th>Value</th>
<th>Effect</th>
</tr>
</thead>
<tbody>
<tr>
<td>M_MXFAST</td>
<td>0</td>
<td>If called before any other malloc subsystem subroutine, this enables the Default allocation policy for the process.</td>
</tr>
<tr>
<td>M_MXFAST</td>
<td>1</td>
<td>If called before any other malloc subsystem subroutine, this enables the 3.1 allocation policy for the process.</td>
</tr>
<tr>
<td>M_DISCLAIM</td>
<td>0</td>
<td>If called while the Default Allocator is enabled, all free memory in the process heap is disclaimed.</td>
</tr>
<tr>
<td>M_MALIGN</td>
<td>N</td>
<td>If called at runtime, sets the default malloc allocation alignment to the value N. The N value must be a power of 2 (greater than or equal to the size of a pointer).</td>
</tr>
</tbody>
</table>

Parameters

Command Specifies the mallopt command to be executed.
Value Specifies the size of each element in the array.
Return Values
Upon successful completion, `mallopt` returns 0. Otherwise, 1 is returned. If an invalid alignment is requested (one that is not a power of 2), `mallopt` fails with a return value of 1, although subsequent calls to `malloc` are unaffected and continue to provide the alignment value from before the failed `mallopt` call.

Error Codes
The `mallopt` subroutine does not set `errno`.

`mallinfo`

Syntax
```c
#include <malloc.h>
#include <stdlib.h>

struct mallinfo mallinfo();
```

Description
The `mallinfo` subroutine can be used to obtain information about the heap managed by the `malloc` subsystem.

Return Values
The `mallinfo` subroutine returns a structure of type `struct mallinfo`, filled in with relevant information and statistics about the heap. The contents of this structure can be interpreted using the definition of `struct mallinfo` in `/usr/include/malloc.h`.

Error Codes
The `mallinfo` subroutine does not set `errno`.

`mallinfo_heap`

Syntax
```c
#include <malloc.h>
#include <stdlib.h>

struct mallinfo_heap mallinfo_heap (Heap);
```

int Heap;

Description
In a multiheap context, the `mallinfo_heap` subroutine can be used to obtain information about a specific heap managed by the `malloc` subsystem.

Parameters

`Heap` Specifies which heap to query.

Return Values
`mallinfo_heap` returns a structure of type `struct mallinfo_heap`, filled in with relevant information and statistics about the heap. The contents of this structure can be interpreted using the definition of `struct mallinfo_heap` in `/usr/include/malloc.h`. 
Error Codes
The mallinfo_heap subroutine does not set errno.

alloca

Syntax
#include <stdlib.h>

char *alloca (Size)
int Size;

Description
The alloca subroutine returns a pointer to a block of memory of at least the number of bytes specified by the Size parameter. The space is allocated from the stack frame of the caller and is automatically freed when the calling subroutine returns.

If alloca is used in code compiled with the C++ compiler, #pragma alloca must be added to the source before the reference to alloca. Alternatively, the -ma compiler option can be used during compilation.

Parameters
Size Specifies the size, in bytes, of memory to allocate.

Return Values
The alloca subroutine returns a pointer to space of the requested size.

Error Codes
The alloca subroutine does not set errno.

valloc

Syntax
#include <stdlib.h>

void *valloc (Size)
size_t Size;

Description
The valloc subroutine is supported as a compatibility interface in the Berkeley Compatibility Library (libbsd.a), as well as in libc.a. The valloc subroutine has the same effect as malloc, except that the allocated memory is aligned to a multiple of the value returned by sysconf (_SC_PAGESIZE).

Parameters
Size Specifies the size, in bytes, of memory to allocate.
Return Values
Upon successful completion, `valloc` returns a pointer to a memory object that is `Size` bytes in length, aligned to a page-boundary. Undefined results occur if the space assigned by the `valloc` subroutine is overrun.

If the request cannot be satisfied for any reason, `valloc` returns NULL.

Error Codes

<table>
<thead>
<tr>
<th>Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EINVAL</td>
<td>0 bytes was requested (in a mode other than Linux mode), or an internal error was detected.</td>
</tr>
<tr>
<td>ENOMEM</td>
<td>Insufficient storage space is available to service the request.</td>
</tr>
</tbody>
</table>

posix_memalign

Syntax

```c
#include <stdlib.h>

int posix_memalign(void **Pointer2Pointer, Align, Size);
void ** Pointer2Pointer;
size_t Align;
size_t Size;
```

Description
The `posix_memalign` subroutine allocates `Size` bytes of memory aligned on a boundary specified by `Align`. The address of this memory is stored in `Pointer2Pointer`.

Parameters

- `Pointer2Pointer` Specifies the location in which the address should be copied.
- `Align` Specifies the alignment of the allocated memory, in bytes. The `Align` parameter must be a power-of-two multiple of the size of a pointer.
- `Size` Specifies the size, in bytes, of memory to allocate.

Return Values
Upon successful completion, `posix_memalign` returns 0. Otherwise, an error number is returned to indicate the error.

Error Codes

- EINVAL The value of `Align` is not a power-of-two multiple of the size of a pointer.
- ENOMEM Insufficient storage space is available to service the request.

Related Information
The `_end`, `_etext`, or `_edata` ("_end, _etext, or _edata Identifier" on page 223) identifier.
madd, msub, mult, mdiv, pow, gcd, invert, rpow, msqrt, mcmp, move, min, omin, fmin, m_in, mout, omout, fmout, m_out, sdiv, or itom

Subroutine

Purpose
Multiple-precision integer arithmetic.

Library
Berkeley Compatibility Library (libbsd.a)

Syntax
#include <mp.h>
#include <stdio.h>

typedef struct mint {
  int Length;
  short * Value
} MINT;

madd(a, b, c)
msub(a, b, c)
mult(a, b, c)
mdiv(a, b, q, r)
pow(a, b, n, c)
gcd(a, b, c)
invert(a, b, c)
rpow(a, n, c)
msqrt(a, b, r)
mcmp(a, b)
move(a, b)
min(a)
omin(a)
fmin(a, f)
m_in(a, n, f)
mout(a)
omout(a)
fmout(a, f)
m_out(a, n, f)
MINT *q, *b, *c, *m, *q, *r;
FILE *f;
int n;
sdiv(a, n, q, r)
MINT *a, *q;
short n;
short *r;
MINT *itom(n)

Description
These subroutines perform arithmetic on integers of arbitrary Length. The integers are stored using the defined type MINT. Pointers to a MINT can be initialized using the itom subroutine, which sets the initial Value to n. After that, space is managed automatically by the subroutines.

The madd subroutine, msub subroutine, and mult subroutine assign to c the sum, difference, and product, respectively, of a and b.

The mdiv subroutine assigns to q and r the quotient and remainder obtained from dividing a by b.
The \texttt{sdiv} subroutine is like the \texttt{mdiv} subroutine except that the divisor is a short integer $n$ and the remainder is placed in a short whose address is given as $r$.

The \texttt{msqrt} subroutine produces the integer square root of $a$ in $b$ and places the remainder in $r$.

The \texttt{rpow} subroutine calculates in $c$ the value of $a$ raised to the (regular integral) power $n$, while the \texttt{pow} subroutine calculates this with a full multiple precision exponent $b$ and the result is reduced modulo $m$.

\textbf{Note:} The \texttt{pow} subroutine is also present in the IEEE Math Library, \texttt{libm.a}, and the System V Math Library, \texttt{libmsaa.a}. The \texttt{pow} subroutine in \texttt{libm.a} or \texttt{libmsaa.a} may be loaded in error unless the \texttt{libbsd.a} library is listed before the \texttt{libm.a} or \texttt{libmsaa.a} library on the command line.

The \texttt{gcd} subroutine returns the greatest common denominator of $a$ and $b$ in $c$, and the \texttt{invert} subroutine computes $c$ such that $a^*c \mod b=1$, for $a$ and $b$ relatively prime.

The \texttt{mcmp} subroutine returns a negative, 0, or positive integer value when $a$ is less than, equal to, or greater than $b$, respectively.

The \texttt{move} subroutine copies $a$ to $b$. The \texttt{min} subroutine and \texttt{mout} subroutine do decimal input and output while the \texttt{omin} subroutine and \texttt{omout} subroutine do octal input and output. More generally, the \texttt{fmin} subroutine and \texttt{fmout} subroutine do decimal input and output using file $f$, and the \texttt{m_in} subroutine and \texttt{m_out} subroutine do inputs and outputs with arbitrary radix $n$. On input, records should have the form of strings of digits terminated by a new line; output records have a similar form.

Programs that use the multiple-precision arithmetic functions must link with the \texttt{libbsd.a} library.

\texttt{pow} is also the name of a standard math library routine.

\subsection*{Parameters}

\begin{itemize}
  \item \texttt{Length} Specifies the length of an integer.
  \item \texttt{Value} Specifies the initial value to be used in the routine.
  \item \texttt{a} Specifies the first operand of the multiple-precision routines.
  \item \texttt{b} Specifies the second operand of the multiple-precision routines.
  \item \texttt{c} Contains the integer result.
  \item \texttt{f} A pointer of the type \texttt{FILE} that points to input and output files used with input/output routines.
  \item \texttt{m} Indicates modulo.
  \item \texttt{n} Provides a value used to specify radix with \texttt{m_in} and \texttt{m_out}, power with \texttt{rpow}, and divisor with \texttt{sdiv}.
  \item \texttt{q} Contains the quotient obtained from \texttt{mdiv}.
  \item \texttt{r} Contains the remainder obtained from \texttt{mdiv}, \texttt{sdiv}, and \texttt{msqrt}.
\end{itemize}

\subsection*{Error Codes}

Error messages and core images are displayed as a result of illegal operations and running out of memory.

\subsection*{Files}

\texttt{/usr/lib/libbsd.a} Object code library.

\section*{Related Information}

The \texttt{bc} command, \texttt{dc} command.
Subroutines Overview in AIX 5L Version 5.3 General Programming Concepts: Writing and Debugging Programs.

madvisesubroutine

Purpose
Advises the system of expected paging behavior.

Library
Standard C Library (libc.a).

Syntax
#include <sys/types.h>
#include <sys/mman.h>

int madvises(addr, len, behav)
caddr_t addr;
size_t len;
int behav;

Description
The madvisesubroutine permits a process to advise the system about its expected future behavior in referencing a mapped file region or anonymous memory region.

The madvisesubroutine has no functionality and is supported for compatibility only.

Parameters
addr Specifies the starting address of the memory region. Must be a multiple of the page size returned by the sysconf subroutine using the_SC_PAGE_SIZE value for the Name parameter.

len Specifies the length, in bytes, of the memory region. If the len value is not a multiple of page size as returned by the sysconf subroutine using the_SC_PAGE_SIZE value for the Name parameter, the length of the region will be rounded up to the next multiple of the page size.

behav Specifies the future behavior of the memory region. The following values for the behav parameter are defined in the /usr/include/sys/mman.h file:

<table>
<thead>
<tr>
<th>Value</th>
<th>Paging Behavior Message</th>
</tr>
</thead>
<tbody>
<tr>
<td>MADV_NORMAL</td>
<td>The system provides no further special treatment for the memory region.</td>
</tr>
<tr>
<td>MADV_RANDOM</td>
<td>The system expects random page references to that memory region.</td>
</tr>
<tr>
<td>MADV_SEQUENTIAL</td>
<td>The system expects sequential page references to that memory region.</td>
</tr>
<tr>
<td>MADV_WILLNEED</td>
<td>The system expects the process will need these pages.</td>
</tr>
<tr>
<td>MADV_DON'TNEED</td>
<td>The system expects the process does not need these pages.</td>
</tr>
<tr>
<td>MADV_SPACEAVAIL</td>
<td>The system will ensure that memory resources are reserved.</td>
</tr>
</tbody>
</table>
Return Values
When successful, the *madvise* subroutine returns 0. Otherwise, it returns -1 and sets the *errno* global variable to indicate the error.

Error Codes
If the *madvise* subroutine is unsuccessful, the *errno* global variable can be set to one of the following values:

- **EINVAL**: The *behav* parameter is invalid.
- **ENOSPC**: The *behav* parameter specifies **MADV_SPACEAVAIL** and resources cannot be reserved.

Related Information
The *mmap* subroutine, **sysconf** subroutine.

List of Memory Manipulation Services and Understanding Paging Space Programming Requirements in **AIX 5L Version 5.3 General Programming Concepts: Writing and Debugging Programs**.

### makecontext or swapcontext Subroutine

**Purpose**
Modifies the context specified by *ucp*.

**Library**
(libc.a)

**Syntax**

```c
#include <ucontext.h>

void makecontext (ucontext_t *ucp, (void *(void) (), int argc, ...);
int swapcontext (uncontext_t *oucp, const uncontext_t *(ucp);
```

**Description**
The *makecontext* subroutine modifies the context specified by *ucp*, which has been initialized using *getcontext* subroutine. When this context is resumed using *swapcontext* subroutine or *setcontext* subroutine, program execution continues by calling *func* parameter, passing it the arguments that follow *argc* in the *makecontext* subroutine.

Before a call is made to *makecontext* subroutine, the context being modified should have a stack allocated for it. The value of *argc* must match the number of integer argument passed to *func* parameter, otherwise the behavior is undefined.

The *uc_link* member is used to determine the context that will be resumed when the context being modified by *makecontext* subroutine returns. The *uc_link* member should be initialized prior to the call to *makecontext* subroutine.

The *swapcontext* subroutine function saves the current context in the context structure pointed to by *oucp* parameter and sets the context to the context structure pointed to by *ucp*.

**Parameters**

- **ucp**: A pointer to a user structure.
A pointer to a user structure.

A pointer to a function to be called when ucp is restored.
The number of arguments being passed to func parameter.

**Return Values**
On successful completion, swapcontext subroutine returns 0. Otherwise, a value of -1 is returned and errno is set to indicate the error.

-1 Not successful and the errno global variable is set to one of the following error codes.

**Error Codes**

**ENOMEM** The ucp argument does not have enough stack left to complete the operation.

**Related Information**
The exec subroutine, exit subroutine, wait subroutine, getcontext subroutine, sigaction subroutine, and sigprocmask subroutine.

---

### matherr Subroutine

**Purpose**
Math error handling function.

**Library**
System V Math Library (libmsaa.a)

**Syntax**

```c
#include <math.h>
int matherr (x)
    struct exception *x;
```

**Description**
The matherr subroutine is called by math library routines when errors are detected.

You can use matherr or define your own procedure for handling errors by creating a function named matherr in your program. Such a user-designed function must follow the same syntax as matherr. When an error occurs, a pointer to the exception structure will be passed to the user-supplied matherr function. This structure, which is defined in the math.h file, includes:

```c
int type;
char *name;
double arg1, arg2, retval;
```
Parameters

type Specifies an integer describing the type of error that has occurred from the following list defined by the math.h file:

- **DOMAIN**  
  Argument domain error
- **SING**  
  Argument singularity
- **OVERFLOW**  
  Overflow range error
- **UNDERFLOW**  
  Underflow range error
- **TLOSS**  
  Total loss of significance
- **PLOSS**  
  Partial loss of significance.

name Points to a string containing the name of the routine that caused the error.
arg1 Points to the first argument with which the routine was invoked.
arg2 Points to the second argument with which the routine was invoked.
retval Specifies the default value that is returned by the routine unless the user’s matherr function sets it to a different value.

Return Values

If the user’s matherr function returns a non-zero value, no error message is printed, and the errno global variable will not be set.

Error Codes

If the function matherr is not supplied by the user, the default error-handling procedures, described with the math library routines involved, will be invoked upon error. In every case, the errno global variable is set to EDOM or ERANGE and the program continues.

Related Information

The bessel: j0, j1, jn, y0, y1, yn subroutine, exp, expm1, log, log10, log1p, pow subroutine, lgamma subroutine, hypot, cabs subroutine, sinh, cosh, tanh subroutine.

Subroutines Overview in AIX 5L Version 5.3 General Programming Concepts: Writing and Debugging Programs.

MatchAllAuths, MatchAnyAuths, MatchAllAuthsList, or MatchAnyAuthsList Subroutine

Purpose

Compare authorizations.

Library

Security Library (libc.a)

Syntax

```c
#include <usersec.h>
```
int MatchAllAuths(CommaListOfAuths)
char *CommaListOfAuths;
int MatchAllAuthsList(CommaListOfAuths, NSListOfAuths)
char *CommaListOfAuths;
char *NSListOfAuths;

int MatchAnyAuths(CommaListOfAuths)
char *CommaListOfAuths;

int MatchAnyAuthsList(CommaListOfAuths, NSListOfAuths)
char *CommaListOfAuths;
char *NSListOfAuths;

Description
The MatchAllAuthsList subroutine compares the CommaListOfAuths against the NSListOfAuths. It returns a non-zero value if all the authorizations in CommaListOfAuths are contained in NSListOfAuths. The MatchAllAuths subroutine calls the MatchAllAuthsList subroutine passing in the results of the GetUserAuths subroutine in place of NSListOfAuths. If NSListOfAuths contains the OFF keyword, MatchAllAuthsList will return a zero value. If NSListOfAuths contains the ALL keyword and not the OFF keyword, MatchAllAuthsList will return a non-zero value.

The MatchAnyAuthsList subroutine compares the CommaListOfAuths against the NSListOfAuths. It returns a non-zero value if one or more of the authorizations in CommaListOfAuths are contained in NSListOfAuths. The MatchAnyAuths subroutine calls the MatchAnyAuthsList subroutine passing in the results of the GetUserAuths subroutine in place of NSListOfAuths. If NSListOfAuths contains the OFF keyword, MatchAnyAuthsList will return a zero value. If NSListOfAuths contains the ALL keyword and not the OFF keyword, MatchAnyAuthsList will return a non-zero value.

Parameters

CommaListOfAuths:
Specifies one or more authorizations, each separated by a comma.

NSListOfAuths:
Specifies zero or more authorizations. Each authorization is null terminated. The last entry in the list must be a null string.

Return Values
The subroutines return a non-zero value if a proper match was found. Otherwise, they will return zero. If an error occurs, the subroutines will return zero and set errno to indicate the error. If the subroutine returns zero and no error occurred, errno is set to zero.

mblen Subroutine

Purpose
Determines the length in bytes of a multibyte character.

Library
Standard C Library (libc.a)

Syntax
#include <stdlib.h>

int mblen(const char *MbString, size_t Number);
Description
The `mblen` subroutine determines the length, in bytes, of a multibyte character.

Parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>Mbstring</code></td>
<td>Points to a multibyte character string.</td>
</tr>
<tr>
<td><code>Number</code></td>
<td>Specifies the maximum number of bytes to consider.</td>
</tr>
</tbody>
</table>

Return Values
The `mblen` subroutine returns 0 if the `MbString` parameter points to a null character. It returns -1 if a character cannot be formed from the number of bytes specified by the `Number` parameter. If `MbString` is a null pointer, 0 is returned.

Related Information
The “mbslen Subroutine” on page 789, “mbstowcs Subroutine” on page 795, and “mbtowc Subroutine” on page 796.

Subroutines, Example Programs, and Libraries in AIX 5L Version 5.3 General Programming Concepts: Writing and Debugging Programs.


mbrlen Subroutine

Purpose
Get number of bytes in a character (restartable).

Library
Standard Library (`libc.a`)

Syntax
```c
#include <wchar.h>
size_t mbrlen (const char *s, size_t n, mbstate_t *ps )
```

Description
If `s` is not a null pointer, `mbrlen` determines the number of bytes constituting the character pointed to by `s`. It is equivalent to:
```c
mbstate_t internal;
mbrtowc(NULL, s, n, ps != NULL ? ps : &internal);
```

If `ps` is a null pointer, the `mbrlen` function uses its own internal `mbstate_t` object, which is initialized at program startup to the initial conversion state. Otherwise, the `mbstate_t` object pointed to by `ps` is used to completely describe the current conversion state of the associated character sequence. The implementation will behave as if no function defined in this specification calls `mbrlen`.

The behavior of this function is affected by the LC_CTYPE category of the current locale.
Return Values

The `mbrlen` function returns the first of the following that applies:

- **0** If the next `n` or fewer bytes complete the character that corresponds to the null wide-character
- **Positive** If the next `n` or fewer bytes complete a valid character; the value returned is the number of bytes that complete the character.
- **(size_t)-2** If the next `n` bytes contribute to an incomplete but potentially valid character, and all `n` bytes have been processed. When `n` has at least the value of the MB_CUR_MAX macro, this case can only occur if `s` points at a sequence of redundant shift sequences (for implementations with state-dependent encodings).
- **(size_t)-1** If an encoding error occurs, in which case the next `n` or fewer bytes do not contribute to a complete and valid character. In this case, EILSEQ is stored in `errno` and the conversion state is undefined.

Error Codes

The `mbrlen` function may fail if:

- **EINVAL** `ps` points to an object that contains an invalid conversion state.
- **EILSEQ** Invalid character sequence is detected.

Related Information

The `mbsinit ("mbsinit Subroutine" on page 788)` subroutine, `mbtowc ("mbtowc Subroutine")` subroutine.

mbtowc Subroutine

Purpose

Convert a character to a wide-character code (restartable).

Library

Standard Library (`libc.a`)

Syntax

```c
#include <wchar.h>

size_t mbtowc (wchar_t * pwc, const char * s, size_t n, mbstate_t * ps);
```

Description

If `s` is a null pointer, the `mbtowc` function is equivalent to the call:

```c
mbtowc(NULL, ' ', 1, ps)
```

In this case, the values of the arguments `pwc` and `n` are ignored.

If `s` is not a null pointer, the `mbtowc` function inspects at most `n` bytes beginning at the byte pointed to by `s` to determine the number of bytes needed to complete the next character (including any shift sequences). If the function determines that the next character is completed, it determines the value of the corresponding wide-character and then, if `pwc` is not a null pointer, stores that value in the object pointed to by `pwc`. If the corresponding wide-character is the null wide-character, the resulting state described is the initial conversion state.

If `ps` is a null pointer, the `mbtowc` function uses its own internal `mbstate_t` object, which is initialized at program startup to the initial conversion state. Otherwise, the `mbstate_t` object pointed to by `ps` is used to
completely describe the current conversion state of the associated character sequence. The implementation will behave as if no function defined in this specification calls `mbrtowc`.

The behavior of this function is affected by the LC_CTYPE category of the current locale.

**Return Values**
The `mbrtowc` function returns the first of the following that applies:

- **0** If the next n or fewer bytes complete the character that corresponds to the null wide-character (which is the value stored).
- **positive** If the next n or fewer bytes complete a valid character (which is the value stored); the value returned is the number of bytes that complete the character.
- **(size_t)-2** If the next n bytes contribute to an incomplete but potentially valid character, and all n bytes have been processed (no value is stored). When n has at least the value of the MB_CUR_MAX macro, this case can only occur if s points at a sequence of redundant shift sequences (for implementations with state-dependent encodings).
- **(size_t)-1** If an encoding error occurs, in which case the next n or fewer bytes do not contribute to a complete and valid character (no value is stored). In this case, EILSEQ is stored in errno and the conversion state is undefined.

**Error Codes**
The `mbrtowc` function may fail if:

- **EINVAL** `ps` points to an object that contains an invalid conversion state.
- **EILSEQ** Invalid character sequence is detected.

**Related Information**
The `mbsinit` subroutine.

### mbsadvance Subroutine

**Purpose**
Advances to the next multibyte character.

**Note:** The `mbsadvance` subroutine is specific to the manufacturer. It is not defined in the POSIX, ANSI, or X/Open standards. Use of this subroutine may affect portability.

**Library**
Standard C Library (`libc.a`)

**Syntax**
```c
#include <mbstr.h>

char *mbsadvance (S)
const char *S;
```

**Description**
The `mbsadvance` subroutine locates the next character in a multibyte character string. The LC_CTYPE category affects the behavior of the `mbsadvance` subroutine.
Parameters

$S$ Contains a multibyte character string.

Return Values

If the $S$ parameter is not a null pointer, the mbsadvance subroutine returns a pointer to the next multibyte character in the string pointed to by the $S$ parameter. The character at the head of the string pointed to by the $S$ parameter is skipped. If the $S$ parameter is a null pointer or points to a null string, a null pointer is returned.

Examples

To find the next character in a multibyte string, use the following:

```c
#include <mbstr.h>
#include <locale.h>
#include <stdlib.h>

main()
{
    char *mbs, *pmbs;
    (void) setlocale(LC_ALL, "");
    /*
    ** Let mbs point to the beginning of a multi-byte string.
    */
    pmbs = mbs;
    while(pmbs){
        pmbs = mbsadvance(mbs);
        /* pmbs points to the next multi-byte character 
        ** in mbs */
    }
}
```

Related Information

The mbsinvalid subroutine.

Subroutines, Example Programs, and Libraries in AIX 5L Version 5.3 General Programming Concepts: Writing and Debugging Programs.


mbcat, mbscmp, or mbscpy Subroutine

Purpose

Performs operations on multibyte character strings.

Library

Standard C Library (libc.a)

Syntax

```c
#include <mbstr.h>
char *mbcat(MbString1, MbString2)
char *MbString1, *MbString2;
int mbscmp(MbString1, MbString2)
char *MbString1, *MbString2;
```
char *mbscpy(MbString1, MbString2)
char *MbString1, *MbString2;

Description
The mbscat, mbscmp, and mbscpy subroutines operate on null-terminated multibyte character strings.

The mbscat subroutine appends multibyte characters from the MbString2 parameter to the end of the MbString1 parameter, appends a null character to the result, and returns MbString1.

The mbscmp subroutine compares multibyte characters based on their collation weights as specified in the LC_COLLATE category. The mbscmp subroutine compares the MbString1 parameter to the MbString2 parameter, and returns an integer greater than 0 if MbString1 is greater than MbString2. It returns 0 if the strings are equivalent and returns an integer less than 0 if MbString1 is less than MbString2.

The mbscpy subroutine copies multibyte characters from the MbString2 parameter to the MbString1 parameter and returns MbString1. The copy operation terminates with the copying of a null character.

Related Information
The mbsncat, mbsncmp, mbsncpy subroutine, wcscat, wcscmp, wcscpy subroutine.

mbschr Subroutine

Purpose
Locates a character in a multibyte character string.

Library
Standard C Library (libc.a)

Syntax
#include <mbstr.h>

char *mbschr(MbString, MbCharacter)
char *MbString;
mbchar_t MbCharacter;

Description
The mbschr subroutine locates the first occurrence of the value specified by the MbCharacter parameter in the string pointed to by the MbString parameter. The MbCharacter parameter specifies a multibyte character represented as an integer. The terminating null character is considered to be part of the string.

The LC_CTYPE category affects the behavior of the mbschr subroutine.
Parameters

MbString Points to a multibyte character string.
MbCharacter Specifies a multibyte character represented as an integer.

Return Values
The mbschr subroutine returns a pointer to the value specified by the MbCharacter parameter within the multibyte character string, or a null pointer if that value does not occur in the string.

Related Information
The "mbspbrk Subroutine" on page 791, "mbsrchr Subroutine" on page 792, "mbstomb Subroutine" on page 794, wcschr subroutine. Subroutines, Example Programs, and Libraries in AIX 5L Version 5.3 General Programming Concepts: Writing and Debugging Programs.


mbsinit Subroutine

Purpose
Determine conversion object status.

Library
Standard Library (libc.a)

Syntax
#include <wchar.h>
int mbsinit (const mbstate_t * p) ;

Description
If ps is not a null pointer, the mbsinit function determines whether the object pointed to by ps describes an initial conversion state.

The mbstate_t object is used to describe the current conversion state from a particular character sequence to a wide-character sequence (or vice versa) under the rules of a particular setting of the LC_CTYPE category of the current locale.

The initial conversion state corresponds, for a conversion in either direction, to the beginning of a new character sequence in the initial shift state. A zero valued mbstate_t object is at least one way to describe an initial conversion state. A zero valued mbstate_t object can be used to initiate conversion involving any character sequence, in any LC_CTYPE category setting.

Return Values
The mbsinit function returns non-zero if ps is a null pointer, or if the pointed-to object describes an initial conversion state; otherwise, it returns zero.

If an mbstate_t object is altered by any of the functions described as restartable, and is then used with a different character sequence, or in the other conversion direction, or with a different LC_CTYPE category setting than on earlier function calls, the behavior is undefined.
Related Information

The "mbrlen Subroutine" on page 783, "mbrtowc Subroutine" on page 784, wctomb subroutine, "mbsrtowcs Subroutine" on page 793, wcsrtombs subroutine.

mbsinvalid Subroutine

Purpose
Validates characters of multibyte character strings.

Note: The mbsinvalid subroutine is specific to the manufacturer. It is not defined in the POSIX, ANSI, or X/Open standards. Use of this subroutine may affect portability.

Library
Standard C Library (libc.a)

Syntax
#include <mbstr.h>

char *mbsinvalid (const char *S);

Description
The mbsinvalid subroutine examines the string pointed to by the S parameter to determine the validity of characters. The LC_CTYPE category affects the behavior of the mbsinvalid subroutine.

Parameters
S Contains a multibyte character string.

Return Values
The mbsinvalid subroutine returns a pointer to the byte following the last valid multibyte character in the S parameter. If all characters in the S parameter are valid multibyte characters, a null pointer is returned. If the S parameter is a null pointer, the behavior of the mbsinvalid subroutine is unspecified.

Related Information
The "mbsadvance Subroutine" on page 785.

mbslen Subroutine

Purpose
Determines the number of characters (code points) in a multibyte character string.

Note: The mbslen subroutine is specific to the manufacturer. It is not defined in the POSIX, ANSI, or X/Open standards. Use of this subroutine may affect portability.
Library
Standard C Library (libc.a)

Syntax
#include <stdlib.h>

size_t mbslen(MbString)
char *mbs;

Description
The mbslen subroutine determines the number of characters (code points) in a multibyte character string. The LC_CTYPE category affects the behavior of the mbslen subroutine.

Parameters
MbString Points to a multibyte character string.

Return Values
The mbslen subroutine returns the number of multibyte characters in a multibyte character string. It returns 0 if the MbString parameter points to a null character or if a character cannot be formed from the string pointed to by this parameter.

Related Information
The mblen (“mblen Subroutine” on page 782) subroutine, mbstowcs (“mbstowcs Subroutine” on page 795) subroutine, mbtowc (“mbtowc Subroutine” on page 796) subroutine.

mbsncat, mbsncmp, or mbsncpy Subroutine

Purpose
Performs operations on a specified number of null-terminated multibyte characters.

Note: These subroutines are specific to the manufacturer. They are not defined in the POSIX, ANSI, or X/Open standards. Use of these subroutines may affect portability.
int mbsncmp(MbString1, MbString2, Number)
char *MbString1, *MbString2;
size_t Number;
char *mbsncpy(MbString1, MbString2, Number)
char *MbString1, *MbString2;
size_t Number;

Description
The mbsncat, mbsncmp, and mbsncpy subroutines operate on null-terminated multibyte character strings.

The mbsncat subroutine appends up to the specified maximum number of multibyte characters from the MbString2 parameter to the end of the MbString1 parameter, appends a null character to the result, and then returns the MbString1 parameter.

The mbsncmp subroutine compares the collation weights of multibyte characters. The LC_COLLATE category specifies the collation weights for all characters in a locale. The mbsncmp subroutine compares up to the specified maximum number of multibyte characters from the MbString1 parameter to the MbString2 parameter. It then returns an integer greater than 0 if MbString1 is greater than MbString2. It returns 0 if the strings are equivalent. It returns an integer less than 0 if MbString1 is less than MbString2.

The mbsncpy subroutine copies up to the value of the Number parameter of multibyte characters from the MbString2 parameter to the MbString1 parameter and returns MbString1. If MbString2 is shorter than Number multi-byte characters, MbString1 is padded out to Number characters with null characters.

Parameters
MbString1 Contains a multibyte character string.
MbString2 Contains a multibyte character string.
Number Specifies a maximum number of characters.

Related Information
The “mbscat, mbscmp, or mbscpy Subroutine” on page 786, “mbscat, mbscmp, or mbscpy Subroutine” on page 786, “mbscat, mbscmp, or mbscpy Subroutine” on page 786, wcsncat subroutine, wcsncmp subroutine, wcsncpy subroutine.

Subroutines, Example Programs, and Libraries in AIX 5L Version 5.3 General Programming Concepts: Writing and Debugging Programs.


mbspbrk Subroutine

Purpose
Locates the first occurrence of multibyte characters or code points in a string.

Note: The mbspbrk subroutine is specific to the manufacturer. It is not defined in the POSIX, ANSI, or X/Open standards. Use of this subroutine may affect portability.

Library
Standard C Library (libc.a)
Syntax

```c
#include <mbstr.h>

char *mbspbrk(MbString1, MbString2)
char *MbString1, *MbString2;
```

Description

The `mbspbrk` subroutine locates the first occurrence in the string pointed to by the `MbString1` parameter, of any character of the string pointed to by the `MbString2` parameter.

Parameters

- **MbString1**: Points to the string being searched.
- **MbString2**: Pointer to a set of characters in a string.

Return Values

The `mbspbrk` subroutine returns a pointer to the character. Otherwise, it returns a null character if no character from the string pointed to by the `MbString2` parameter occurs in the string pointed to by the `MbString1` parameter.

Related Information

- `mbschr` subroutine
- `mbstomb` subroutine
- `wcspbrk` subroutine
- `wcswcs` subroutine

Subroutines, Example Programs, and Libraries in AIX 5L Version 5.3 General Programming Concepts: Writing and Debugging Programs.


---

**mbsrchr Subroutine**

Purpose

Locates a character or code point in a multibyte character string.

Library

Standard C Library (`libc.a`)

Syntax

```c
#include <mbstr.h>

char *mbsrchr(MbString, MbCharacter)
char *MbString;
int MbCharacter;
```

Description

The `mbsrchr` subroutine locates the last occurrence of the `MbCharacter` parameter in the string pointed to by the `MbString` parameter. The `MbCharacter` parameter is a multibyte character represented as an integer. The terminating null character is considered to be part of the string.
Parameters

MbString
Points to a multibyte character string.
MbCharacter
Specifies a multibyte character represented as an integer.

Return Values

The mbstrchr subroutine returns a pointer to the MbCharacter parameter within the multibyte character string. It returns a null pointer if MbCharacter does not occur in the string.

Related Information

The "mbschr Subroutine" on page 787, "mbspbrk Subroutine" on page 791, "mbstomb Subroutine" on page 794, wcsrchr subroutine. Subroutines, Example Programs, and Libraries in AIX 5L Version 5.3 General Programming Concepts: Writing and Debugging Programs.


mbsrtowcs Subroutine

Purpose
Convert a character string to a wide-character string (restartable).

Library
Standard Library (libc.a)

Syntax

```c
#include <wchar.h>
size_t mbsrtowcs (wchar_t * dst, const char ** src, size_t len, mbstate_t * ps);
```

Description

The mbsrtowcs function converts a sequence of characters, beginning in the conversion state described by the object pointed to by ps, from the array indirectly pointed to by src into a sequence of corresponding wide-characters. If dst is not a null pointer, the converted characters are stored into the array pointed to by dst. Conversion continues up to and including a terminating null character, which is also stored. Conversion stops early in either of the following cases:

- When a sequence of bytes is encountered that does not form a valid character.
- When len codes have been stored into the array pointed to by dst (and dst is not a null pointer).

Each conversion takes place as if by a call to the mbtowc function.

If dst is not a null pointer, the pointer object pointed to by src is assigned either a null pointer (if conversion stopped due to reaching a terminating null character) or the address just past the last character converted (if any). If conversion stopped due to reaching a terminating null character, and if dst is not a null pointer, the resulting state described is the initial conversion state.

If ps is a null pointer, the mbsrtowcs function uses its own internal mbstate_t object, which is initialised at program startup to the initial conversion state. Otherwise, the mbstate_t object pointed to by ps is used to completely describe the current conversion state of the associated character sequence. The implementation will behave as if no function defined in this specification calls mbsrtowcs.
The behavior of this function is affected by the LC_CTYPE category of the current locale.

**Return Values**

If the input conversion encounters a sequence of bytes that do not form a valid character, an encoding error occurs. In this case, the `mbsrtowcs` function stores the value of the macro EILSEQ in errno and returns `(size_t)-1`; the conversion state is undefined. Otherwise, it returns the number of characters successfully converted, not including the terminating null (if any).

**Error Codes**

The `mbsrtowcs` function may fail if:

- `EINVAL` ps points to an object that contains an invalid conversion state.
- `EILSEQ` Invalid character sequence is detected.

**Related Information**

The "mbsinit Subroutine" on page 788, "mbrtowc Subroutine" on page 784.

---

**mbstomb Subroutine**

**Purpose**

Extracts a multibyte character from a multibyte character string.

**Note:** The `mbstomb` subroutine is specific to the manufacturer. It is not defined in the POSIX, ANSI, or X/Open standards. Use of this subroutine may affect portability.

**Library**

Standard C Library (libc.a)

**Syntax**

```c
#include <mbstr.h>

mbchar_t mbstomb (MbString)
const char *MbString;
```

**Description**

The `mbstomb` subroutine extracts the multibyte character pointed to by the `MbString` parameter from the multibyte character string. The LC_CTYPE category affects the behavior of the `mbstomb` subroutine.

**Parameters**

- `MbString` Contains a multibyte character string.

**Return Values**

The `mbstomb` subroutine returns the code point of the multibyte character as a `mbchar_t` data type. If an unusable multibyte character is encountered, a value of 0 is returned.

**Related Information**

The "mbschr Subroutine" on page 787, "mbspbrk Subroutine" on page 791, "mbsrchr Subroutine" on page 792.
mbstowcs Subroutine

Purpose
Converts a multibyte character string to a wide character string.

Library
Standard C Library (libc.a)

Syntax
```
#include <stdlib.h>

size_t mbstowcs(WcString, String, Number);

wchar_t * WcString;
const char * String;
size_t Number;
```

Description
The mbstowcs subroutine converts the sequence of multibyte characters pointed to by the String parameter to wide characters and places the results in the buffer pointed to by the WcString parameter. The multibyte characters are converted until a null character is reached or until the number of wide characters specified by the Number parameter have been processed.

Parameters

- **WcString**
  Points to the area where the result of the conversion is stored.
- **String**
  Points to a multibyte character string.
- **Number**
  Specifies the maximum number of wide characters to be converted.

Return Values
The mbstowcs subroutine returns the number of wide characters converted, not including a null terminator, if any. If an invalid multibyte character is encountered, a value of -1 is returned. The WcString parameter does not include a null terminator if the value Number is returned.

If WcString is a null wide character pointer, the mbstowcs subroutine returns the number of elements required to store the wide character codes in an array.

Error Codes
The mbstowcs subroutine fails if the following occurs:

- **EILSEQ**
  Invalid byte sequence is detected.

Related Information
The mblen Subroutine on page 782, mbslen Subroutine on page 789, mbtowc Subroutine on page 796, wcstombs subroutine, wctomb subroutine.
mbswidth Subroutine

Purpose
Determines the number of multibyte character string display columns.

Note: The mbswidth subroutine is specific to this manufacturer. It is not defined in the POSIX, ANSI, or X/Open standards. Use of this subroutine may affect portability.

Library
Standard C Library (libc.a)

Syntax
#include <mbstr.h>

int mbswidth (MbString, Number)
const char *MbString;
size_t Number;

Description
The mbswidth subroutine determines the number of display columns required for a multibyte character string.

Parameters
MbString Contains a multibyte character string.
Number Specifies the number of bytes to read from the s parameter.

Return Values
The mbswidth subroutine returns the number of display columns that will be occupied by the MbString parameter if the number of bytes (specified by the Number parameter) read from the MbString parameter form valid multibyte characters. If the MbString parameter points to a null character, a value of 0 is returned. If the MbString parameter does not point to valid multibyte characters, -1 is returned. If the MbString parameter is a null pointer, the behavior of the mbswidth subroutine is unspecified.

Related Information
The wcswidth subroutine, wcwidth subroutine.

mbtowc Subroutine

Purpose
Converts a multibyte character to a wide character.
Library
Standard C Library (libc.a)

Syntax
```c
#include <stdlib.h>

int mbtowc (WideCharacter, String, Number);
wchar_t *WideCharacter;
const char *String;
size_t Number;
```

Description
The `mbtowc` subroutine converts a multibyte character to a wide character and returns the number of bytes of the multibyte character.

The `mbtowc` subroutine determines the number of bytes that comprise the multibyte character pointed to by the `String` parameter. It then converts the multibyte character to a corresponding wide character and, if the `WideCharacter` parameter is not a null pointer, places it in the location pointed to by the `WideCharacter` parameter. If the `WideCharacter` parameter is a null pointer, the `mbtowc` subroutine returns the number of converted bytes but does not change the `WideCharacter` parameter value. If the `WideCharacter` parameter returns a null value, the multibyte character is not converted.

Parameters
- **WideCharacter**: Specifies the location where a wide character is to be placed.
- **String**: Specifies a multibyte character.
- **Number**: Specifies the maximum number of bytes of a multibyte character.

Return Values
The `mbtowc` subroutine returns a value of 0 if the `String` parameter is a null pointer. The subroutine returns a value of -1 if the bytes pointed to by the `String` parameter do not form a valid multibyte character before the number of bytes specified by the `Number` parameter (or fewer) have been processed. It then sets the `errno` global variable to indicate the error. Otherwise, the number of bytes comprising the multibyte character is returned.

Error Codes
The `mbtowc` subroutine fails if the following occurs:
- **EILSEQ**: Invalid byte sequence is detected.

Related Information
The `mblen` subroutine on page 782, `mbslen` subroutine on page 789, `mbstowcs` subroutine on page 795, `wcstombs` subroutine, `wctomb` subroutine.

Subroutines, Example Programs, and Libraries in AIX 5L Version 5.3 General Programming Concepts: Writing and Debugging Programs.

memccpy, memchr, memcmp, memcpy, memset or memmove
Subroutine

Purpose
Performs memory operations.

Library
Standard C Library (libc.a)

Syntax
#include <memory.h>

void *memccpy (Target, Source, C, N)
void *Target;
const void *Source;
int C;
size_t N;

void *memchr (S, C, N)
const void *S;
int C;
size_t N;

int memcmp (Target, Source, N)
const void *Target, *Source;
size_t N;

void *memcpy (Target, Source, N)
void *Target;
const void *Source;
size_t N;

void *memset (S, C, N)
void *S;
int C;
size_t N;

void *memmove (Target, Source, N)
void *Source;
const void *Target;
size_t N;

Description
The memory subroutines operate on memory areas. A memory area is an array of characters bounded by a count. The memory subroutines do not check for the overflow of any receiving memory area. All of the memory subroutines are declared in the memory.h file.

The memccpy subroutine copies characters from the memory area specified by the Source parameter into the memory area specified by the Target parameter. The memccpy subroutine stops after the first character specified by the C parameter (converted to the unsigned char data type) is copied, or after N characters are copied, whichever comes first. If copying takes place between objects that overlap, the behavior is undefined.

The memcmp subroutine compares the first N characters as the unsigned char data type in the memory area specified by the Target parameter to the first N characters as the unsigned char data type in the memory area specified by the Source parameter.

The memcpy subroutine copies N characters from the memory area specified by the Source parameter to the area specified by the Target parameter and then returns the value of the Target parameter.
The `memset` subroutine sets the first $N$ characters in the memory area specified by the $S$ parameter to the value of character $C$ and then returns the value of the $S$ parameter.

Like the `memcpy` subroutine, the `memmove` subroutine copies $N$ characters from the memory area specified by the `Source` parameter to the area specified by the `Target` parameter. However, if the areas of the `Source` and `Target` parameters overlap, the move is performed nondestructively, proceeding from right to left.

The `memccpy` subroutine is not in the ANSI C library.

**Parameters**

- **Target**: Points to the start of a memory area.
- **Source**: Points to the start of a memory area.
- **C**: Specifies a character to search.
- **N**: Specifies the number of characters to search.
- **S**: Points to the start of a memory area.

**Return Values**

The `memccpy` subroutine returns a pointer to character $C$ after it is copied into the area specified by the `Target` parameter, or a null pointer if the $C$ character is not found in the first $N$ characters of the area specified by the `Source` parameter.

The `memchr` subroutine returns a pointer to the first occurrence of the $C$ character in the first $N$ characters of the memory area specified by the `S` parameter, or a null pointer if the $C$ character is not found.

The `memcmp` subroutine returns the following values:

- **Less than 0**: If the value of the `Target` parameter is less than the values of the `Source` parameter.
- **Equal to 0**: If the value of the `Target` parameter equals the value of the `Source` parameter.
- **Greater than 0**: If the value of the `Target` parameter is greater than the value of the `Source` parameter.

**Related Information**

The `swab` subroutine.

See `Subroutines Overview` in *AIX 5L Version 5.3 General Programming Concepts: Writing and Debugging Programs*.

---

**mincore Subroutine**

**Purpose**

Determines residency of memory pages.

**Library**

Standard C Library (`libc.a`).
Syntax

```c
int mincore (addr, len, *vec)
caddr_t addr;
size_t len;
char *vec;
```

Description

The `mincore` subroutine returns the primary-memory residency status for regions created from calls made to the `mmap` subroutine. The status is returned as a character for each memory page in the range specified by the `addr` and `len` parameters. The least significant bit of each character returned is set to 1 if the referenced page is in primary memory. Otherwise, the bit is set to 0. The settings of the other bits in each character are undefined.

Parameters

- `addr` Specifies the starting address of the memory pages whose residency is to be determined. Must be a multiple of the page size returned by the `sysconf` subroutine using the `_SC_PAGE_SIZE` value for the `Name` parameter.
- `len` Specifies the length, in bytes, of the memory region whose residency is to be determined. If the `len` value is not a multiple of the page size as returned by the `sysconf` subroutine using the `_SC_PAGE_SIZE` value for the `Name` parameter, the length of the region is rounded up to the next multiple of the page size.
- `vec` Specifies the character array where the residency status is returned. The system assumes that the character array specified by the `vec` parameter is large enough to encompass a returned character for each page specified.

Return Values

When successful, the `mincore` subroutine returns 0. Otherwise, it returns -1 and sets the `errno` global variable to indicate the error.

Error Codes

If the `mincore` subroutine is unsuccessful, the `errno` global variable is set to one of the following values:

- `EFAULT` A part of the buffer pointed to by the `vec` parameter is out of range or otherwise inaccessible.
- `EINVAL` The `addr` parameter is not a multiple of the page size as returned by the `sysconf` subroutine using the `_SC_PAGE_SIZE` value for the `Name` parameter.
- `ENOMEM` Addresses in the `(addr, addr + len)` range are invalid for the address space of the process, or specify one or more pages that are not mapped.

Related Information

The `mmap` subroutine, `sysconf` subroutine.

[List of Memory Manipulation Services](#) in *AIX 5L Version 5.3 General Programming Concepts: Writing and Debugging Programs.*

MIO_aio_read64 Subroutine

Purpose

Read asynchronously from a file through MIO library.

Library

Modular I/O Library (libmio.a)
Syntax
#include <libmio.h>

int MIO_aio_read64( FileDescriptor, aiocbp )
int FileDescriptor;
struct aiocb64 *aiocbp;

Description
This subroutine is an entry point of the MIO library for the Legacy AIO aio_read64 subroutine. Use this subroutine to instrument your application with the MIO library. You can replace the Legacy AIO aio_read64 kernel I/O subroutine with this equivalent MIO subroutine. See Modular I/O in Performance management for MIO library implementation.

Use this subroutine to read asynchronously from an open file specified by the FileDescriptor parameter. The FileDescriptor parameter results from an MIO_open64 subroutine.

Parameters
The parameters are those of the corresponding standard POSIX system call aio_read64.

Return Values
The return values are those of the corresponding standard POSIX system call aio_read64.

Error Codes
The error codes are those of the corresponding standard POSIX system call aio_read64.

Location
/usr/lib/libmio.a

Related Information
The Modular I/O in Performance management.

The MIO_aio_suspend64 Subroutine

Purpose
Suspend the calling process until one or more asynchronous I/O requests are completed.

Library
Modular I/O Library (libmio.a)

Syntax
#include <libmio.h>

int MIO_aio_suspend64( Count, aiocbplist )
int Count;
struct aiocb64 **aiocbplist;
Description
This subroutine is an entry point of the MIO library for the Legacy AIO aio_suspend64 subroutine. Use this subroutine to instrument your application with the MIO library. You can replace the Legacy AIO aio_suspend64 kernel I/O subroutine with this equivalent MIO subroutine. See Modular I/O in Performance management for the MIO library implementation.

The aio_suspend64 subroutine suspends the calling process until one or more of the Count parameter asynchronous I/O requests are completed or a signal interrupts the subroutine. Specifically, the aio_suspend64 subroutine handles requests associated with the aio control block (aiocb) structures pointed to by the aiocbpalist parameter.

Parameters
The parameters are those of the corresponding standard POSIX system call aio_suspend64.

Return Values
The return values are those of the corresponding standard POSIX system call aio_suspend64.

Error Codes
The error codes are those of the corresponding standard POSIX system call aio_suspend64.

Location
/usr/lib/libmio.a

Related Information
The Modular I/O in Performance management.

MIO_aio_write64 Subroutine

Purpose
Write asynchronously to a file through the MIO library.

Library
Modular I/O library (libmio.a)

Syntax
#include <libmio.h>

int MIO_aio_write64( FileDescriptor, aiocbp )
int FileDescriptor; struct aiocb64 *aiocbp;
struct aiocb64 *aiocbp;

Description
This subroutine is an entry point of the MIO library for the Legacy AIO aio_write64 subroutine. Use this subroutine to instrument your application with the MIO library. You can replace the Legacy AIO aio_write64 kernel I/O subroutine with this equivalent MIO subroutine. See Modular I/O in Performance management for the MIO library implementation.
Use this subroutine to write asynchronously to an open file specified by the FileDescriptor parameter. The FileDescriptor parameter results from an MIO_open64 subroutine.

Parameters
The parameters are those of the corresponding standard POSIX system call aio_write64.

Return Values
The return values are those of the corresponding standard POSIX system call aio_write64.

Error Codes
The error codes are those of the corresponding standard POSIX system call aio_write64.

Location
/usr/lib/libmio.a

Related Information
Modular I/O in Performance management.

The aio_write64, MIO_open64, MIO_close, MIO_lseek64, MIO_write, MIO_ftruncate64, MIO_fstat64, MIO_fcntl, MIO_ffinfo, and MIO_fsync subroutines.

MIO_close Subroutine

Purpose
Close a file descriptor through the MIO library.

Library
Modular I/O library (libmio.a)

Syntax
#include <libmio.h>

int MIO_close (FileDescriptor)

int FileDescriptor;

Description
This subroutine is an entry point of the MIO library. Use this subroutine to instrument your application with the MIO library. You can replace the close kernel I/O subroutine with this equivalent MIO subroutine. See the Modular I/O in Performance management for the MIO library implementation.

Use this subroutine to close a file with the FileDescriptor parameter through the Modular I/O (MIO) library. The FileDescriptor parameter results from the MIO_open64 subroutine.

Parameters
The parameters are those of the corresponding standard POSIX system call close.

Return Values
The return values are those of the corresponding standard POSIX system call close.
Error Codes

The error codes are those of the corresponding standard POSIX system call close.

Standard Output

MIO library outputs are flushed on the **MIO_close** subroutine call in the stats file.

The following is the information found in the diagnostic output file. It contains debug information:

- If you set the stats option of the trace module (trace/stats), it runs diagnostics from the trace module.
- If you set the stats option of the pf module (pf/stats), it runs diagnostics from the pf module.
- If you set the stats option of the recov module (recov/stats), it runs diagnostics from the recovery trace.
- If you set the nostats option of the trace or the pf module, these diagnostics are suppressed.

The diagnostic file name is defined in the MIO_STATS environment variable if the stats option is set to the default value of *mioout*.

To separate the trace, pf or recov module diagnostics from other outputs, set the stats options to the following other file names:

- trace/stats=<*tracefile*>
- pf/stats=<*pffile*>
- recov/stats=<*recovfile*>

The *tracefile*, *pffile* and *recovfile* are templates for the file names of module diagnostics output. You can give file names for the output of the trace, pf or recov module diagnostics.

Standard output includes the following information:

**Header**, which contains the following information:

- Date
- Hostname
- Enabled or disabled AIO
- Program name
- MIO library version
- Environment variables

**Debug**, which contains the following information:

- List of all the debug options
- Table of all of the modules’ definitions if the DEF debug option is set
- Open request made to the **MIO_open64** subroutine if the OPEN debug option is set
- Modules invoked if the MODULES debug option is set

**Trace module diagnostic**, which contains the following information:

- Time if the TIMESTAMP debug option is set
- Trace on close or on intermediate interrupt
- Trace module position in module_list
- Processed file name
- Rate, which is the amount of data divided by the total time. The total time here means the cumulative amount of time spent beneath the trace module
- Demand rate, which is the amount of data divided by the length of time when the file is opened (including the time of opening and closing the file)
- The current (when tracing) file size and the maximum size of the file during this file processing
- File system information: file type and sector size
- Open mode and flags of the file
- For each subroutine, the following information is displayed:
  - name of the subroutine
  - count of calling of this subroutine
  - time of processing for this subroutine
- For read or write subroutines, the following information is displayed:
  - requested (requested size to read or write)
  - total (real size read or write: returned by AIX system call)
  - min (minimum size to read or write)
  - max (maximum size to read or write)
- For the seek subroutine, the following information is displayed:
  - the average seek delta (total seek delta/seek count)
- For the aread or awrite subroutine:
  - count, time and rate of transfer time including suspend, and read or write time
- For the fcntl subroutine, the number of pages is returned.

The following is an example of a trace diagnostic:

```
date
Trace on close or intermediate:
previous module or calling program<->next module: file name:
(total transferred bytes/total time)=rate
  demand rate=rate/s=(total transferred bytes/(close time-open time))
  current size=actual size of the file
max_size=max size of the file
mode=file open mode
FileSystemType=file system type given by fststat(stat_b.f_vfstype)
sector size=Minimum direct i/o transfer size
oflags=file open flags
open open count open time
fcntl fcntl count fcntl time
read read count read time requested size total size minimum maximum
aread aread count aread time requested size total size minimum maximum
suspend count time rate
write write count write time requested size total size minimum maximum
seek seek count seek time average seek delta
size
page fcntl page info count
```

The following is a sample of a trace diagnostic:

```
MIO statistics file: Tue May 10 14:14:08 2005
hostname=host1 : with Legacy aio available
Program=example
MIO library libmio.a 3.0.0.60  AIX 5.1 32 bit addressing built
Apr 19 2005 15:08:17
MIO_INSTALL_PATH=
MIO_STATS =example.stats
MIO_DEBUG =OPEN
MIO_FILES =*.dat [trace/stats]
MIO_DEFAULTS = trace/kbytes

MIO_DEBUG OPEN =T
```
Opening file file.dat
modules[11]=trace/stats

Trace close : program <-> aix : file.dat : (4800/0.04)=111538.02 kbytes/s
demand rate=42280.91 kbytes/s=4800/(0.12-0.01))
current size=0  max size=1600
mode =0640  FileSystemType=JFS  sector size=4096
oflags =0x302=RDWR

open
  1  0.00
write
  100  0.02  1600  1600  16384  16384
read
  200  0.02  3200  3200  16384  16384
seek
  101  0.01 average seek delta=48503
fcntl
  1  0.00
trunc
  1  0.01
close
  1  0.00

size
  100

The following is a template of the pf module diagnostic:

pf close for <name of the file in the cache>
pf close for <global or private cache> <global cache number>
<nb_pg_compute>page of <page size> <sector size> bytes per sector
<nb_real_pg_not_pf>/<nb_pg_not_pf> pages not preread for write
<nb_unused_pf> unused prefetches out of <nb_start_pf>
prefetch= <nb_pg_to_pf>
<number> of write behind
<number> of page syncs forced by ill formed writes
<number> of pages retained over close
<unit> transferred / Number of requests
program --> <bytes written into the cache by parent>/
<number of write from parent> --> pf -->
<written out of the cache from the child>/ <number of partial page written>
program --> <bytes read out of the cache by parent>/
<number of read from parent> --> pf -->
<bytes read in from child of the cache>/ <number of page read from child>

The following is explanation of the terms in the pf module template:

- nb_pg_compute= number of page compute by cache_size/ page size
- nb_real_pg_not_pf= real number page not prefetch because of pffw option (suppress number of page prefetch because sector not valid)
- nb_pg_not_pf= page of unused prefetch
- nb_unused_pf= number of started prefetch
- nb_pg_to_pf= number of page to prefetch

The following is a sample of the pf module diagnostic:

pf close for /home/user1/pthread/258/SM20182_0.SCR300
50 pages of 2097152 bytes 131072 bytes per sector
133/133 pages not preread for write
23 unused prefetches out of 242 : prefetch=2
95 write backs
mbytes transferred / Number of requests
program --> 257/257 --> pf --> 257/131 --> aix
program --> 269/269 --> pf --> 265/133 --> aix

The following is the recov module output:
If open or write routine failed, the **reco**v module, if set, is called. The **reco**v module adds the following comments in the output file:

- The value of the **open_command** option
- The value of the **command** option
- The **errno**
- The index of retry

The following is a sample of the **reco**v module:

```plaintext
15:30:00
  reco : command=ls -l file=file.dat errno=28 try=0
  reco : failure : new_ret=-1
```

**Location**

`/usr/lib/libmio.a`

**Related Information**

The Modular I/O in *Performance management*.


**MIO_fcntl Subroutine**

**Purpose**
Control open file descriptors through the MIO library.

**Library**
Modular I/O library (`libmio.a`)

**Syntax**
```
#include <libmio.h>

int MIO_fcntl ( FileDescriptor, Command, Argument )

int FileDescriptor, Command, Argument;
```

**Description**
This subroutine is an entry point of the MIO library, offering the same features as the `fcntl` subroutine. Use this subroutine to instrument your application with the MIO library. You can replace the `fcntl` kernel I/O subroutine with this equivalent MIO subroutine. See Modular I/O in *Performance management* for the MIO library implementation.

Use this subroutine to perform controlling operations on the open file specified by the `FileDescriptor` parameter. The `FileDescriptor` parameter results from the **MIO_open64** subroutine

**Parameters**
The parameters are those of the corresponding standard POSIX system call `fcntl`.

**Return Values**
The return values are those of the corresponding standard POSIX system call `fcntl`.
Error Codes
The error codes are those of the corresponding standard POSIX system call fcntl.

Location
/usr/lib/libmio.a

Related Information
The Modular I/O in Performance management.

The fcntl, MIO_open64, MIO_close, MIO_lseek64, MIO_write, MIO_ftruncate64, MIO_fstat64, MIO_ffinfo, and MIO_fsync subroutines.

MIO_ffinfo Subroutine

Purpose
Return file information through the MIO library.

Library
Modular I/O library (libmio.a)

Syntax
#include <libmio.h>

int MIO_ffinfo (FileDescriptor, Command, Buffer, Length)

int FileDescriptor;

int Command;

struct diocapbuf *Buffer;

int Length;

Description
This subroutine is an entry point of the MIO library. Use this subroutine to instrument your application with the MIO library. You can replace the ffinfo kernel I/O subroutine with this equivalent MIO subroutine. See the Modular I/O in Performance management for MIO library implementation.

Use this subroutine to obtain specific file information for the open file referenced by the FileDescriptor parameter. The FileDescriptor parameter results from the MIO_open64 subroutine.

Parameters
The parameters are those of the corresponding standard POSIX system call ffinfo.

Return Values
The return values are those of the corresponding standard POSIX system call ffinfo.

Error Codes
The error codes are those of the corresponding standard POSIX system call ffinfo.

Location
/usr/lib/libmio.a
MIO_fstat64 Subroutine

Purpose
Provide information about a file through the MIO library.

Library
Modular I/O library (libmio.a)

Syntax
#include <libmio.h>

int MIO_fstat64 (Filedescriptor, Buffer)
int FileDescriptor;
struct stat64 *Buffer;

Description
This subroutine is an entry point of the MIO library. Use this subroutine to instrument your application with the MIO library. You can replace the fstat64 kernel I/O subroutine with this equivalent MIO subroutine. See the Modular I/O in Performance management for the MIO library implementation.

Use this subroutine to obtain information about the open file referenced by FileDescriptor parameter. The FileDescriptor parameter results from the MIO_open64 subroutine.

Parameters
The parameters are those of the corresponding standard POSIX system call fstat64.

Return Values
The return values are those of the corresponding standard POSIX system call fstat64.

Error Codes
The error codes are those of the corresponding standard POSIX system call fstat64.

Location
/usr/lib/libmio.a

Related Information
The Modular I/O in Performance management.

The ffinfo, MIO_open64, MIO_close, MIO_lseek64, MIO_write, MIO_ftruncate64, MIO_fstat64
MIO_fcntl, and MIO_fsync subroutines.
MIO_fsync Subroutine

Purpose
Save changes in a file to permanent storage through the MIO library.

Library
Modular I/O library (libmio.a)

Syntax
#include <libmio.h>
int MIO_fsync (FileDescriptor)
int FileDescriptor;

Description
This subroutine is an entry point of the MIO library. Use this subroutine to instrument your application with the MIO library. You can replace the fsync kernel I/O subroutine with this equivalent MIO subroutine. See the Modular I/O in Performance management for the MIO library implementation.

Use this subroutine to save to permanent storage all modified data in the specified range of the open file specified by the FileDescriptor parameter. The FileDescriptor parameter results from the MIO_open64 subroutine.

Parameters
The parameters are those of the corresponding standard POSIX system call fsync.

Return Values
The return values are those of the corresponding standard POSIX system call fsync.

Error Codes
The error codes are those of the corresponding standard POSIX system call fsync.

Location
/usr/lib/libmio.a

Related Information
The Modular I/O in Performance management.

The fsync, MIO_open64, MIO_close, MIO_lseek64, MIO_write, MIO_ftruncate64, MIO_fstats64, MIO_fcntl, and MIO_ffinfo subroutines.

MIO_ftruncate64 Subroutine

Purpose
Change the length of regular files through the MIO library.

Library
Modular I/O library (libmio.a)
Syntax
#include <libmio.h>

int MIO_ftruncate64 (FileDescriptor, Length)
int FileDescriptor;
int64 Length;

Description
This subroutine is an entry point of the MIO library. Use this subroutine to instrument your application with the MIO library. You can replace the ftruncate64 kernel I/O subroutine with this equivalent MIO subroutine. See the Modular I/O in Performance management for the MIO library implementation.

Use this subroutine to change the length of the open file specified by the FileDescriptor parameter through Modular I/O (MIO) library. The FileDescriptor parameter results from the MIO_open64 subroutine.

Parameters
The parameters are those of the corresponding standard POSIX system call ftruncate64.

Return Values
The return values are those of the corresponding standard POSIX system call ftruncate64.

Error Codes
The error codes are those of the corresponding standard POSIX system call ftruncate64.

Location
/usr/lib/libmio.a

Related Information
The Modular I/O in Performance management.
The ftruncate64, MIO_open64, MIO_close, MIO_lseek64, MIO_write, MIO_fstat64, MIO_fcntl, MIO_finfo, and MIO_fsync subroutines.

MIO_lio_listio64 Subroutine

Purpose
Initiate a list of asynchronous I/O requests with a single call.

Library
Modular I/O library (libmio.a)

Syntax
#include <libmio.h>

int MIO_lio_listio64 (Command, List, Nent, Eventp)
int Command;
struct liocb64 *List;
int Nent;
struct event *Eventp;
Description
This subroutine is an entry point of the MIO library for the Legacy AIO lio_listio64 Subroutine. Use this subroutine to instrument your application with MIO library. You can replace the Legacy AIO lio_listio64 kernel I/O subroutine with this equivalent MIO subroutine. See the Modular I/O in Performance management for the MIO library implementation.

The lio_listio64 subroutine allows the calling process to initiate the Nent parameter asynchronous I/O requests. These requests are specified in the liocb structures pointed to by the elements of the List array. The call may block or return immediately depending on the Command parameter. If the Command parameter requests that I/O completion be asynchronously notified, a SIGIO signal is delivered when all of the I/O operations are completed.

Parameters
The parameters are those of the corresponding standard POSIX system call lio_listio64.

Return Values
The return values are those of the corresponding standard POSIX system call lio_listio64.

Error Codes
The error codes are those of the corresponding standard POSIX system call lio_listio64.

Location
/usr/lib/libmio.a

Related Information
The Modular I/O in Performance management.

MIO_lseek64 Subroutine

Purpose
Move the read-write file pointer through the MIO library.

Library
Modular I/O library (libmio.a)

Syntax
#include <libmio.h>

int64 MIO_lseek64 (FileDescriptor, Offset, Whence) 
int FileDescriptor; 
int64 Offset; 
int Whence;
**Description**

This subroutine is an entry point of the MIO library. Use this subroutine to instrument your application with the MIO library. You can replace the fseek64 kernel I/O subroutine with this equivalent MIO subroutine. See the Modular I/O in Performance management for the MIO library implementation.

Use this subroutine to set the read-write file pointer for the open file specified by the FileDescriptor parameter through the Modular I/O (MIO) library. The FileDescriptor parameter results from the MIO_open64 subroutine.

**Parameters**

The parameters are those of the corresponding standard POSIX system call lseek64.

**Return Values**

The return values are those of the corresponding standard POSIX system call lseek64.

**Error Codes**

The error codes are those of the corresponding standard POSIX system call lseek64.

**Location**

/usr/lib/libmio.a

**Related Information**

The Modular I/O in Performance management.

The lseek64, MIO_open64, MIO_close, MIO_write, MIO_ftruncate64, MIO_fstat64, MIO_fcntl, MIO_ffinfo, and MIO_fsync subroutines.

---

**MIO_open64 Subroutine**

**Purpose**

Open a file for reading or writing through the MIO library.

**Library**

Modular I/O library (libmio.a)

**Syntax**

```c
#include <libmio.h>

int MIO_open64 (Path, OFlag, Mode, Extra)
char *Path;
int OFlag;
int Mode;
struct mio_extra *Extra;
```

**Description**

This subroutine is an entry point of the MIO library. Use this subroutine to instrument your application with the MIO library. You can replace the open64 kernel I/O subroutine with this equivalent MIO subroutine. See the Modular I/O in Performance management for the MIO library implementation.

Use this subroutine to open a file through the Modular I/O (MIO) library. This library creates the context for this open file, according to the configuration set in MIO environment variables, or in the Extra parameter.
To analyze your application I/O and tune the I/O, use the MIO subroutines in the place of the standard I/O subroutines.

The MIO subroutines are:

- MIO_close
- MIO_lseek64
- MIO_read
- MIO_write
- MIO_ftruncate64
- MIO_fstat64
- MIO_fcntl
- MIO_ffinfo
- MIO_fsync

The standard I/O subroutines are:

- close
- lseek64
- read
- write
- truncate64
- fstat64
- fcntl
- ffinfo
- fsync

Parameters
The parameters are those of the corresponding standard POSIX system call open64, except the Extra parameter.

Extra
Specifies some extra arguments for the MIO library. The simplest implementation is for any application to pass a zero pointer as the fourth argument. The fourth argument is a pointer to the mio_extra structure, you can usually pass a zero pointer, or can pass a mio_extra pointer (only for very advanced use).
The mio_extra structure is defined as follows:

```c
struct mio_extra {
    int    cookie;
    /* Default value: MIO_EXTRA_COOKIE/ */
    int    taskid;
    /* for later */
    int64  bufsiz;
    /* if > 1: force the prefetch for write pffw */
    char *modules;
    /* explicit module name,
    if any modules returns from MIO_FILES environment variable match */
    char *logical_name;
    /* logical file name to open
    if file name don't match with MIO_FILES regexp */
    int    flags;
    /* if MIO_EXTRA_SKIP_MIO_FILES_FLAG:
    don't use MIO_FILES env variable, but use extra->modules */
};
```

**Note:** For applications that would not use the environment variable interface to apply the MIO modules to a file, the mio_extra hook provides an easy way to do that.

**Environment variables**

MIO is controlled by the following environment variables, which define the MIO features and are processed by the `MIO_open64` subroutine:

The `MIO_STATS` variable is used to indicate a file that will be used as a repository for diagnostic messages and for output requested from the MIO modules. It is interpreted as a file name with two special cases. If the file is either standard `stderr` or `stdout` output, the output will be directed towards the appropriate file stream. If the first character of the `MIO_STATS` variable is a plus sign (+), the file name to be used is the string following the plus sign (+), and the file will be opened for appending. Without the preceding plus sign (+), the file is overwritten.

The `MIO_FILES` variable is the key to determine which modules are to be invoked for a given file when the `MIO_open64` subroutine is called. The format for the `MIO_FILES` variable is the following:

```text
first_file_name_list [ module list ] second_file_name_list [ module list ] ...
```

When the `MIO_open64` subroutine is called, MIO checks for the existence of the `MIO_FILES` variable and parses it as follows:

The `MIO_FILES` variable is parsed left to right. All characters up to the next occurrence of the bracket ([]) are taken as a `file_name_list`. A `file_name_list` is a colon(:) separated list of `file_name_templates`. A `file_name_templates` is used to match the name of the file opened by MIO and can use the following wildcard characters:

- `*` Matches zero or more characters of a directory or file name.
- `?` Matches one character of a directory or file name.
- `**` Matches all remaining characters of a full path name.
If the \textit{file\_name\_templates} does not contain a forward slash (/), then all of the path directory information in the file name passed to the \texttt{MIO\_open64} subroutine is ignored and matching is applied only to the file name of the file being opened.

If the name of the file being opened is matched by one of the \textit{file\_name\_templates} in the \textit{file\_name\_list} then the module list to be invoked is taken as the string between brackets (\{ \}). If the name of the file match two or more \textit{file\_name\_templates}, the first match is taken into account. If the name of the file being opened does not match any of the \textit{file\_name\_templates} in any of the \textit{file\_name\_lists} then the file is opened with a default invocation of the AIX module.

If a match has occurred, the modules to be invoked are taken from the associated module list in the \texttt{MIO\_FILES} variable. The modules are invoked left to right, with the left-most being closest to the user program and the right-most being closest to the operating system. If the module list does not start with the MIO module, a default invocation of the MIO module is added as a prefix. If the AIX module is not specified, a default invocation of the AIX module is appended.

The following is an example of the \texttt{MIO\_FILES} variable:

\begin{verbatim}
setenv MIO_FILES " *.dat [ trace/stats ]" 
\end{verbatim}

Assume the \texttt{MIO\_FILES} variable is set as follows:

\begin{verbatim}
MIO\_FILES= *.dat:*.*.scr [ trace ] *.f01:*.*.f02:*.*.f03 [ trace | pf | trace ]
\end{verbatim}

If the \texttt{test.dat} file is opened by the \texttt{MIO\_open64} subroutine, the \texttt{test.dat} file name matches \texttt{*.dat} and the following modules are invoked:

\begin{verbatim}
mio | trace | aix
\end{verbatim}

If the \texttt{test.f02} file is opened by the \texttt{MIO\_open64} subroutine, the \texttt{test.f02} file name matches the second \textit{file\_name\_templates} in the second \textit{file\_name\_list} and the following modules are invoked:

\begin{verbatim}
mio | trace | pf | trace | aix
\end{verbatim}

Each module has its own hardcoded default options for a default invocation. You can override the default options by specifying them in the associated \texttt{MIO\_FILES} module list. The following example turns on the \texttt{stats} option for the \texttt{trace} module and requests that the output be directed to the \texttt{my.stats} file:

\begin{verbatim}
MIO\_FILES= *.dat:*.*.scr [ trace/stats=my.stats ]
\end{verbatim}

The options for a module are delimited with a forward slash (/). Some options require an associated string value and others might require an integer value. For those requiring a string value, if the string includes a forward slash (/), enclose the string in braces (\{ \}).

For those options requiring an integer value, append the integer value with a \texttt{k}, \texttt{m}, \texttt{g}, or \texttt{t} to represent kilo, mega, giga, or tera. You might also input integer values in base 10, 8, or 16. If you add a \texttt{0x} prefix to the integer value, the integer is interpreted as base 16. If you add a \texttt{0} prefix to the integer value, the integer is interpreted as base 8. If you add neither a \texttt{0x} prefix nor a \texttt{0} prefix to the integer value, the integer is interpreted as base 10.

The \texttt{MIO\_DEFAULTS} variable is intended as a way to keep the \texttt{MIO\_FILES} variable more readable. If the user is specifying several modules for multiple \textit{file\_name\_list} and \textit{module\_list} pairs, then the \texttt{MIO\_FILES} variable might become quite long. To repeatedly override the hardcoded defaults in the same manner, you can specify new defaults for a module by specifying such defaults in the \texttt{MIO\_DEFAULTS} variable. The \texttt{MIO\_DEFAULTS} variable is a comma separated list of modules with their new defaults.

The following is an example of the \texttt{MIO\_DEFAULTS} variable:

\begin{verbatim}
setenv MIO\_DEFAULTS " trace/kbytes "
\end{verbatim}

Assume that \texttt{MIO\_DEFAULTS} variable is set as follows:

\begin{verbatim}
816 Technical Reference, Volume 1: Base Operating System and Extensions
MIO_DEFAULTS = trace/events=prob.events, aix/debug

Any default invocation of the trace module will have binary event tracing enabled and directed towards the prob.events file and any default invocation of the AIX module will have debug enabled.

The MIO_DEBUG variable is intended as an aid in debugging the use of MIO. MIO searches the MIO_DEFAULTS variable for keywords and provides debugging output for the option. The available keywords are the following:

ALL    Turns on all of the MIO_DEBUG variable keywords.
ENV    Outputs environment variable matching requests.
OPEN    Outputs open requests made to the MIO_open64 subroutine.
MODULES    Outputs modules invoked for each call to the MIO_open64 subroutine.
TIMESTAMP    Places a timestamp preceding each entry into a stats file.
DEF    Outputs the definition table of each module. When the file opens, the outputs of all of the MIO library's definitions are processed for all the MIO library modules.

Return Values

The return values are those of the corresponding standard POSIX system call open64.

Error Codes

The error codes are those of the corresponding standard POSIX system call open64.

Standard Output

There is no MIO library output for the MIO_open64 subroutine.

Note: MIO library output statistics are written in the MIO_close subroutine. This output filename is configurable with the MIO_STATS environment variable.

In the example.stats MIO output file, the module trace is set and reported, and the open requests are output. All of the values are in kilobytes.

Examples

The following example.c file issues 100 writes of 16 KB, seeks to the beginning of the file, issues 100 reads of 16 KB, and then seeks backward through the file reading 16 KB records. At the end the file is truncated to 0 bytes in length.

The filename argument to the following example is the file to be created, written to and read forwards and backwards:

```c
#define _LARGE_FILES
#include <fcntl.h>
#include <stdio.h>
#include <errno.h>
#include "libmio.h"

/* Define open64, lseek64 and ftruncate64, not
* open, lseek, and ftruncate that are used in the code. This is
* because libmio.h defines _LARGE_FILES which forces <fcntl.h> to
* redefine open, lseek, and ftruncate as open64, lseek64, and
* ftruncate64
*/
```
#define open64(a,b,c) MIO_open64(a,b,c,0)
#define close MIO_close
#define lseek64 MIO_lseek64
#define write MIO_write
#define read MIO_read
#define ftruncate64 MIO_ftruncate64
#define RECSIZE 16384
#define NREC 100

main(int argc, char **argv)
{
    int i, fd, status;
    char *name;
    char *buffer;
    int64 ret64;
    
    if( argc < 2 ){
        fprintf(stderr,"Usage : example file_name\n");
        exit(-1);
    }
    name = argv[1];
    buffer = (char *)malloc(RECSIZE);
    memset( buffer, 0, RECSIZE );

    fd = open(name, O_RDWR|O_TRUNC|O_CREAT, 0640 );
    if( fd < 0 ){
        fprintf(stderr,"Unable to open file %s errno=%d\n",name,errno);
        exit(-1);
    }

    /* write the file */
    for(i=0;i<NREC;i++){
        status = write( fd, buffer, RECSIZE );
    }

    /* read the file forwards */
    ret64 = lseek(fd, 0, SEEK_SET );
    for(i=0;i<NREC;i++){
        status = read( fd, buffer, RECSIZE );
    }

    /* read the file backwards */
    for(i=0;i<NREC;i++){
        ret64 = lseek(fd, (NREC-i-1)*RECSIZE, SEEK_SET );
        status = read( fd, buffer, RECSIZE );
    }

    /* truncate the file back to 0 bytes*/
    status = ftruncate( fd, 0 );
    free(buffer);

    /* close the file */
    status = close(fd);
}

Both the (example.c) example and a script that sets the environment variables, compiles and calls the application are delivered and installed with the libmio, as follows:
cc -o example example.c -lmio
./example file.dat
The following environment variables are set to configure MIO:

setenv MIO_STATS example.stats
setenv MIO_FILES "*.dat [trace/stats]"
setenv MIO_DEFAULTS "trace/kbytes"
setenv MIO_DEBUG OPEN

See the /usr/samples/libmio/README file and sample files for details.

**Location**
/usr/lib/libmio.a

**Related Information**
The Modular I/O in Performance management.

The open, MIO_close, MIO_lseek64, MIO_read, MIO_write, MIO_ftruncate64, MIO_fstat64, MIO_fcntl, MIO_ffinfo, and MIO_fsync subroutines.

---

**MIO_open Subroutine**

**Purpose**
Open a file for reading or writing through the MIO library.

**Library**
Modular I/O library (libmio.a)

**Syntax**

```c
#include <libmio.h>

int MIO_open (Path, OFlag, Mode, Extra)
char *Path;
int OFlag;
int Mode;
struct mio_extra *Extra;
```

**Description**
This subroutine, which is a redirection to the MIO_open64 subroutine, is an entry point of the MIO library. To use the MIO library, the files have to be opened with the O_LARGEFILE flag. For more details on the O_LARGEFILE flag, see the fcntl.h file.

Use this subroutine to instrument your application with the MIO library. You can replace the open kernel I/O subroutine with this equivalent MIO subroutine. See the Modular I/O in Performance management for the MIO library implementation.

Use this subroutine to open a file through the Modular I/O (MIO) library. This library creates the context for this open file, according to the configuration set in the MIO environment variables, or in the Extra parameter.

To analyze your application I/O and tune the I/O, use the MIO subroutines in the place of the standard I/O subroutines.

The MIO subroutines are:
• MIO_close
The standard I/O subroutines are:

- close
- lseek64
- read
- write
- ftruncate64
- fstat64
- fcntl
- ffinfo
- fsync

The parameters are those of the corresponding standard POSIX system call open64, except the *Extra* parameter.

*Extra* specifies some extra arguments for the MIO library. The simplest implementation is to pass a zero pointer as the fourth argument. The fourth argument is a pointer to the mio_extra structure, you can usually pass a zero pointer, or can pass a mio_extra pointer (only for very advanced use).

The mio_extra structure is defined as follows:

```c
struct mio_extra {
    int cookie;
    /* Default value: MIO_EXTRA_COOKIE/

    int taskid;
    /* for later */

    int64 bufsiz;
    /* if > 1 : force the prefetch for write pffw */

    char *modules;
    /* explicit module name,
    if any modules returns from MIO_FILES environment variable match */

    char *logical_name;
    /* logical file name to open
    if file name don't match with MIO_FILES regexp */

    int flags;
    /* if MIO_EXTRA_SKIP_MIO_FILES_FLAG :
    don't use MIO_FILES env variable, but use extra->modules */
};
```

**Note:** For applications that would not use the environment variable interface to apply MIO modules to a file, the mio_extra hook provides an easy way to do that.
Environment variables

MIO is controlled through the following four environment variables. These environment variables, which define the MIO features, are processed by the MIO_open64 subroutine.

The MIO_STATS variable is used to indicate a file that will be used as a repository for diagnostic messages and for output requested from the MIO modules. It is interpreted as a file name with two special cases. If the file is either the stderr or stdout output, the output will be directed towards the appropriate file stream. If the first character of the MIO_STATS variable is a plus sign (+), the file name to be used is the string following the plus sign (+), and the file will be opened for appending. Without the preceding plus sign (+), the file is overwritten.

The MIO_FILES variable is the key to determine which modules are to be invoked for a given file when the MIO_open64 subroutine is called. The format for the MIO_FILES variable is the following:

\[
\text{first_file_name_list}\ [\text{module list}] \text{second_file_name_list}\ [\text{module list}] \ldots
\]

When the MIO_open64 subroutine is called, MIO checks for the existence of the MIO_FILES variable and parses it as follows:

The MIO_FILES variable is parsed left to right. All characters up to the next occurrence of the bracket (]) are taken as a file_name_list. A file_name_list is a colon(:) separated list of file_name_templates. A file_name_templates is used to match the name of the file opened by MIO and can use the following wildcard characters:

- `*` Matches zero or more characters of a directory or file name.
- `?` Matches one character of a directory or file name.
- `**` Matches all remaining characters of a full path name.

If the file_name_templates does not contain a forward slash (/), then all of the path directory information in the file name passed to the MIO_open64 subroutine is ignored and matching is applied only to the file name of the file being opened.

If the name of the file being opened is matched by one of the file_name_templates in the file_name_list then the module list to be invoked is taken as the string between brackets ([ ]). If the name of the file match two or more file_name_templates, the first match is taken into account. If the name of the file being opened does not match any of the file_name_templates in any of the file_name_lists then the file is opened with a default invocation of the AIX module.

If a match has occurred, the modules to be invoked are taken from the associated module list in the MIO_FILES variable. The modules are invoked left to right, with the left-most being closest to the user program and the right-most being closest to the operating system. If the module list does not start with the MIO module, a default invocation of the MIO module is added as a prefix. If the AIX module is not specified, a default invocation of the AIX module is appended.

The following is an example of the MIO_FILES variable:

```
setenv MIO_FILES " *.dat [ trace/stats ]"
```

Assume the MIO_FILES variable is set as follows:

```
MIO_FILES= *.dat:*scr [ trace ] *.f01:*f02:*f03 [ trace | pf | trace ]
```

If the test.dat file is opened by the MIO_open64 subroutine, the test.dat file name matches *.dat and the following modules are invoked:

```
mio | trace | aix
```
If the *test.f02* file is opened by the MIO_open64 subroutine, the *test.f02* file name matches the second file_name_templates in the second file_name_list and the following modules are invoked:

mio | trace | pf | trace | aix

Each module has its own hardcoded default options for a default invocation. You can override the default options by specifying them in the associated MIO_FILES module list. The following example turns on the stats option for the trace module and requests that the output be directed to the *my.stats* file:

MIO_FILES= *.dat : *.scr [ trace/stats=my.stats ]

The options for a module are delimited with a forward slash (/). Some options require an associated string value and others might require an integer value. For those requiring a string value, if the string includes a forward slash (/), enclose the string in braces ({ }).

For those options requiring an integer value, append the integer value with a k, m, g, or t to represent kilo, mega, giga, or tera. You might also input integer values in base 10, 8, or 16. If you add a 0x prefix to the integer value, the integer is interpreted as base 16. If you add a 0 prefix to the integer value, the integer is interpreted as base 8. If you add neither a 0x prefix nor a 0 prefix to the integer value, the integer is interpreted as base 10.

The MIO_DEFAULTS variable is intended as a way to keep the MIO_FILES variable more readable. If the user is specifying several modules for multiple file_name_list and module_list pairs, then the MIO_FILES variable might become quite long. To repeatedly override the hardcoded defaults in the same manner, you can specify new defaults for a module by specifying such defaults in the MIO_DEFAULTS variable. The MIO_DEFAULTS variable is a comma separated list of modules with their new defaults.

The following is an example of the MIO_DEFAULTS variable:

setenv MIO_DEFAULTS " trace/kbytes "

Assume that MIO_DEFAULTS variable is set as follows:

MIO_DEFAULTS = trace/events=prob.events , aix/debug

Any default invocation of the trace module will have binary event tracing enabled and directed towards the prob.events file and any default invocation of the AIX module will have debug enabled.

The MIO_DEBUG variable is intended as an aid in debugging the use of MIO. MIO searches the MIO_DEFAULTS variable for keywords and provides debugging output for the option. The available keywords are the following:

**ALL**
Turns on all of the MIO_DEBUG variable keywords.

**ENV**
Outputs environment variable matching requests.

**OPEN**
Outputs open requests made to the MIO_open64 subroutine.

**MODULES**
Outputs modules invoked for each call to the MIO_open64 subroutine.

**TIMESTAMP**
Places a timestamp preceding each entry into a stats file.

**DEF**
Outputs the definition table of each module. When the file opens, the outputs of all of the MIO library’s definitions are processed for all the MIO library modules.

**Return values**
The return values are those of the corresponding standard POSIX system call open64.

**Error codes**
The error codes are those of the corresponding standard POSIX system call open64.
Standard output

There is no MIO library output for the `MIO_open64` subroutine.

Note: MIO library output statistics are written in the `MIO_close` subroutine. This output filename is configurable with the `MIO_STATS` environment variable.

In the `example.stats`. MIO output file, the module trace is set and reported, and the open requests are output. All the values are in kilobytes.

Examples

The following `example.c` file issues 100 writes of 16 KB, seeks to the beginning of the file, issues 100 reads of 16 KB, and then seeks backward through the file reading 16 KB records. At the end the file is truncated to 0 bytes in length.

The `filename` argument to the following example is the file to be created, written to and read forwards and backwards:

```c
#include "libmio.h"

#define open64(a,b,c) MIO_open64(a,b,c,0)
#define close MIO_close
#define lseek64 MIO_lseek64
#define write MIO_write
#define read MIO_read
#define ftruncate64 MIO_ftruncate64

#define RECSIZE 16384
#define NREC 100

main(int argc, char **argv)
{
    int i, fd, status ;
    char *name ;
    char *buffer ;
    int64 ret64 ;
    
    if( argc < 2 ){
        fprintf(stderr,"Usage : example file_name\n");
        exit(-1);
    }
    name = argv[1] ;
    buffer = (char *)malloc(RECSIZE);
    memset( buffer, 0, RECSIZE ) ;
    fd = open(name, O_RDWR|O_TRUNC|O_CREAT, 0640 ) ;
    if( fd < 0 ){
        fprintf(stderr,"Unable to open file %s errno=%d\n",name,errno);
        exit(-1);
    }
    
    /* Define open64, lseek64 and ftruncate64, not
    * open, lseek, and ftruncate that are used in the code. This is
    * because libmio.h defines _LARGE_FILES which forces <fcntl.h> to
    * redefine open, lseek, and ftruncate as open64, lseek64, and
    * ftruncate64
    */
    
    
    return 0 ;
}
```

/* write the file */
for(i=0;i<NREC;i++){
    status = write( fd, buffer, RECSIZE ) ;
}

/* read the file forwards */
ret64 = lseek(fd, 0, SEEK_SET) ;
for(i=0;i<NREC;i++){
    status = read( fd, buffer, RECSIZE ) ;
}
/* read the file backwards */
for(i=0;i<NREC;i++){
    ret64 = lseek(fd, (NREC-i-1)*RECSIZE, SEEK_SET) ;
    status = read( fd, buffer, RECSIZE ) ;
}
/* truncate the file back to 0 bytes*/
status = ftruncate( fd, 0 ) ;
free(buffer);
/* close the file */
status = close(fd);

--------------------------------------------------------------------------------

Both the (example.c) example and a script that sets the environment variables, compiles and calls the application are delivered and installed with the libmio, as follows:
cc -o example example.c -lmio
./example file.dat

The following environment variables are set to configure MIO:

setenv MIO_STATS example.stats
setenv MIO_FILES " *.dat [ trace/stats ] "
setenv MIO_DEFAULTS " trace/kbytes "
setenv MIO_DEBUG OPEN

See the /usr/samples/libmio/README and sample files for details.

Location
/usr/lib/libmio.a

Related Information
The Modular I/O in Performance management.

MIO_read Subroutine

Purpose
Read from a file through the MIO library.
Library
Modular I/O library (libmio.a)

Syntax
#include <libmio.h>

int MIO_read(FileDescriptor, Buffer, NBytes)
int FileDescriptor;
void * Buffer;
int NBytes;

Description
This subroutine is an entry point of the MIO library. Use this subroutine to instrument your application with the MIO library. You can replace the read kernel I/O subroutine with this equivalent MIO subroutine. See the Modular I/O in Performance management for the MIO library implementation.

Use this subroutine to read to the number of bytes of data specified by the NBytes parameter from the file associated with the FileDescriptor parameter into the buffer, through the Modular I/O (MIO) library. The Buffer parameter points to the buffer. The FileDescriptor parameter results from the MIO_open64 subroutine.

Parameters
The parameters are those of the corresponding standard POSIX system call read.

Return Values
The return values are those of the corresponding standard POSIX system call read.

Error Codes
The error codes are those of the corresponding standard POSIX system call read.

Location
/usr/lib/libmio.a

Related Information
The Modular I/O in Performance management.

The read, MIO_open64, MIO_close, MIO_lseek64, MIO_write, MIO_ftruncate64, MIO_fstat64, MIO_fcntl, MIO_ffinfo, and MIO_fsync subroutines.

MIO_write Subroutine

Purpose
Write to a file through the MIO library.

Library
Modular I/O library (libmio.a)
Syntax
#include <libmio.h>

int MIO_write(FileDescriptor, Buffer, NBytes)
int FileDescriptor;
void * Buffer;
int NBytes;

Description
This subroutine is an entry point of the MIO library. Use this subroutine to instrument your application with the MIO library. You can replace the write kernel I/O subroutine with this equivalent MIO subroutine. See the Modular I/O in Performance management for the MIO library implementation.

Use this subroutine to write the number of bytes of data specified by the NBytes parameter from the buffer to the file associated with the FileDescriptor parameter through the Modular I/O (MIO) library. The Buffer parameter points to the buffer. The FileDescriptor parameter results from the MIO_open64 subroutine.

Parameters
The parameters are those of the corresponding standard POSIX system call write.

Return Values
The return values are those of the corresponding standard POSIX system call write.

Error Codes
The error codes are those of the corresponding standard POSIX system call write.

Location
/usr/lib/libmio.a

Related Information
The Modular I/O in Performance management.
The write, MIO_open64, MIO_close, MIO_lseek64, MIO_ftruncate64, MIO_fstat64, MIO_fcntl, MIO_finfo, and MIO_fsync subroutines.

mkdir Subroutine

Purpose
Creates a directory.

Library
Standard C Library (libc.a)

Syntax
#include <sys/stat.h>

int mkdir (Path, Mode)
const char * Path;
mode_t Mode;
Description
The `mkdir` subroutine creates a new directory.

The new directory has the following:
- The owner ID is set to the process-effective user ID.
- If the parent directory has the `SetFileGroupID` (`S_ISGID`) attribute set, the new directory inherits the group ID of the parent directory. Otherwise, the group ID of the new directory is set to the effective group ID of the calling process.
- Permission and attribute bits are set according to the value of the `Mode` parameter, with the following modifications:
  - All bits set in the process-file mode-creation mask are cleared.
  - The `SetFileUserID` and `Sticky` (`S_ISVTX`) attributes are cleared.
- If the `Path` variable names a symbolic link, the link is followed. The new directory is created where the variable pointed.

Parameters

Path
Specifies the name of the new directory. If Network File System (NFS) is installed on your system, this path can cross into another node. In this case, the new directory is created at that node.

To execute the `mkdir` subroutine, a process must have search permission to get to the parent directory of the `Path` parameter as well as write permission in the parent directory itself.

Mode
Specifies the mask for the read, write, and execute flags for owner, group, and others. The `Mode` parameter specifies directory permissions and attributes. This parameter is constructed by logically ORing values described in the `sys/mode.h` file.

Return Values
Upon successful completion, the `mkdir` subroutine returns a value of 0. Otherwise, a value of -1 is returned, and the `errno` global variable is set to indicate the error.

Error Codes
The `mkdir` subroutine is unsuccessful and the directory is not created if one or more of the following are true:

EACCES Creating the requested directory requires writing in a directory with a mode that denies write permission.

EEXIST The named file already exists.

EROFS The named file resides on a read-only file system.

ENOSPC The file system does not contain enough space to hold the contents of the new directory or to extend the parent directory of the new directory.

EMLINK The link count of the parent directory exceeds the maximum (`LINK_MAX`) number. (`LINK_MAX`) is defined in `limits.h` file.

ENAMETOOLONG The `Path` parameter or a path component is too long and cannot be truncated.

ENOENT A component of the path prefix does not exist or the `Path` parameter points to an empty string.

ENOTDIR A component of the path prefix is not a directory.

EDQUOT The directory in which the entry for the new directory is being placed cannot be extended, or an i-node or disk blocks could not be allocated for the new directory because the user’s or group’s quota of disk blocks or i-nodes on the file system containing the directory is exhausted.
The `mkdir` subroutine can be unsuccessful for other reasons. See “Appendix A. Base Operating System Error Codes for Services That Require Path-Name Resolution” for a list of additional errors.

If NFS is installed on the system, the `mkdir` subroutine is also unsuccessful if the following is true:

**ETIMEDOUT** The connection timed out.

### Related Information

The `chmod` subroutine, `mknod` subroutine, `rmdir` subroutine, `umask` subroutine.

The `chmod` command, `mkdir` command, `mknod` command.

Files, Directories, and File Systems for Programmers in AIX 5L Version 5.3 General Programming Concepts: Writing and Debugging Programs.

### mknod or mkfifo Subroutine

#### Purpose

Creates an ordinary file, first-in-first-out (FIFO), or special file.

#### Library

Standard C Library (`libc.a`)

#### Syntax

```
#include <sys/stat.h>

int mknod (const char *Path, mode_t Mode, dev_t Device)
char *Path;
int Mode;
dev_t Device;

int mkfifo (const char *Path, mode_t Mode)
const char *Path;
int Mode;
```

#### Description

The `mknod` subroutine creates a new regular file, special file, or FIFO file. Using the `mknod` subroutine to create file types (other than FIFO or special files) requires root user authority.

For the `mknod` subroutine to complete successfully, a process must have both search and write permission in the parent directory of the `Path` parameter.

The `mkfifo` subroutine is an interface to the `mknod` subroutine, where the new file to be created is a FIFO or special file. No special system privileges are required.

The new file has the following characteristics:

- File type is specified by the `Mode` parameter.
- Owner ID is set to the effective user ID of the process.
- Group ID of the file is set to the group ID of the parent directory if the `SetGroupID` attribute (`S_ISGID`) of the parent directory is set. Otherwise, the group ID of the file is set to the effective group ID of the calling process.
Permission and attribute bits are set according to the value of the *Mode* parameter. All bits set in the file-mode creation mask of the process are cleared.

Upon successful completion, the `mkfifo` subroutine marks for update the `st_atime`, `st_ctime`, and `st_mtime` fields of the file. It also marks for update the `st_ctime` and `st_mtime` fields of the directory that contains the new entry.

If the new file is a character special file having the *S_IMP X* attribute (multiplexed character special file), when the file is used, additional path-name components can appear after the path name as if it were a directory. The additional part of the path name is available to the device driver of the file for interpretation. This feature provides a multiplexed interface to the device driver.

**Parameters**

**Path** Names the new file. If Network File System (NFS) is installed on your system, this path can cross into another node.

**Mode** Specifies the file type, attributes, and access permissions. This parameter is constructed by logically ORing values described in the `sys/mode.h` file.

**Device** Specifies the ID of the device, which corresponds to the `st_rdev` member of the structure returned by the `statx` subroutine. This parameter is configuration-dependent and used only if the *Mode* parameter specifies a block or character special file. If the file you specify is a remote file, the value of the *Device* parameter must be meaningful on the node where the file resides.

**Return Values**

Upon successful completion, a value of 0 is returned. Otherwise, a value of -1 is returned and the *errno* global variable is set to indicate the error.

**Error Codes**

The `mknod` subroutine fails and the new file is not created if one or more of the following are true:

- **EEXIST** The named file exists.
- **EDQUOT** The directory in which the entry for the new file is being placed cannot be extended, or an i-node could not be allocated for the file because the user's or group's quota of disk blocks or i-nodes on the file system is exhausted.
- **EISDIR** The *Mode* parameter specifies a directory. Use the `mkdir` subroutine instead.
- **ENOSPC** The directory that would contain the new file cannot be extended, or the file system is out of file-allocation resources.
- **EPERM** The *Mode* parameter specifies a file type other than *S_IFIFO*, and the calling process does not have root user authority.
- **EROFS** The directory in which the file is to be created is located on a read-only file system.

The `mknod` and `mkfifo` subroutine can be unsuccessful for other reasons. See “Appendix. A Base Operating System Error Codes for Services That Require Path-Name Resolution” for list of additional errors.

If NFS is installed on the system, the `mknod` subroutine can also fail if the following is true:

- **ETIMEDOUT** The connection timed out.
Related Information
The `chmod` subroutine, `mkdir` subroutine, `open`, `openx`, or `creat` subroutine, `statx` subroutine, `umask` subroutine.

The `chmod` command, `mkdir` command, `mknod` command.

The `mode.h` file, `types.h` file.

Files, Directories, and File Systems for Programmers in AIX 5L Version 5.3 General Programming Concepts: Writing and Debugging Programs.

mktemp or mkstemp Subroutine

Purpose
Constructs a unique file name.

Libraries
Standard C Library (`libc.a`)

Berkeley Compatibility Library (libbsd.a)

Syntax
```c
#include <stdlib.h>

char *mktemp (Template);
char *Template;

int mkstemp (Template);
char *Template;
```

Description
The `mktemp` subroutine replaces the contents of the string pointed to by the `Template` parameter with a unique file name.

**Note:** The `mktemp` subroutine creates a filename and checks to see if the file exist. If that file does not exist, the name is returned. If the user calls `mktemp` twice without creating a file using the name returned by the first call to `mktemp`, then the second `mktemp` call may return the same name as the first `mktemp` call since the name does not exist.

To avoid this, either create the file after calling `mktemp` or use the `mkstemp` subroutine. The `mkstemp` subroutine creates the file for you.

To get the BSD version of this subroutine, compile with Berkeley Compatibility Library (libbsd.a).

The `mkstemp` subroutine performs the same substitution to the template name and also opens the file for reading and writing.

In BSD systems, the `mkstemp` subroutine was intended to avoid a race condition between generating a temporary name and creating the file. Because the name generation in the operating system is more random, this race condition is less likely. BSD returns a file name of / (slash).

Former implementations created a unique name by replacing X’s with the process ID and a unique letter.
Parameters

Template  Points to a string to be replaced with a unique file name. The string in the Template parameter is a file name with up to six trailing X’s. Since the system randomly generates a six-character string to replace the X’s, it is recommended that six trailing X’s be used.

Return Values

Upon successful completion, the 

mktemp subroutine returns the address of the string pointed to by the Template parameter.

If the string pointed to by the Template parameter contains no X’s, and if it is an existing file name, the Template parameter is set to a null character, and a null pointer is returned; if the string does not match any existing file name, the exact string is returned.

Upon successful completion, the 

mkstemp subroutine returns an open file descriptor. If the mkstemp subroutine fails, it returns a value of -1.

Related Information

The getpid ("getpid, getpgrp, or getppid Subroutine" on page 402) subroutine, 

TMPFILE subroutine, 

tmpnam or tempnam subroutine.

Files, Directories, and File Systems for Programmers in AIX 5L Version 5.3 General Programming Concepts: Writing and Debugging Programs.

mlock and munlock Subroutine

Purpose

Locks or unlocks a range of process address space.

Library

Standard C Library (libc.a)

Syntax

```
#include <sys/mman.h>

int mlock (addr, len)
const void *addr;
size_t len;
```

```
int munlock (addr, len)
const void *addr;
size_t len;
```

Description

The mlock subroutine causes those whole pages containing any part of the address space of the process starting at address addr and continuing for len bytes to be memory-resident until unlocked or until the process exits or executes another process image. If the starting address addr is not a multiple of PAGESIZE, it is rounded down to the lowest page boundary. The len is rounded up to a multiple of PAGESIZE.

The munlock subroutine unlocks those whole pages containing any part of the address space of the process starting at address addr and continuing for len bytes, regardless of how many times mlock has been called by the process for any of the pages in the specified range.
If any of the pages in the range specified in a call to the munlock subroutine are also mapped into the address spaces of other processes, any locks established on those pages by another process are unaffected by the call of this process to the munlock subroutine. If any of the pages in the range specified by a call to the munlock subroutine are also mapped into other portions of the address space of the calling process outside the range specified, any locks established on those pages through other mappings are also unaffected by this call.

Upon successful return from mlock, pages in the specified range are locked and memory-resident. Upon successful return from munlock, pages in the specified range are unlocked with respect to the address space of the process.

The calling process must have the root user authority to use this subroutine.

**Parameters**

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>addr</td>
<td>Specifies the address space of the process to be locked or unlocked.</td>
</tr>
<tr>
<td>len</td>
<td>Specifies the length in bytes of the address space.</td>
</tr>
</tbody>
</table>

**Return Values**

Upon successful completion, the mlock and munlock subroutines return zero. Otherwise, no change is made to any locks in the address space of the process, the subroutines return -1 and set errno to indicate the error.

**Error Codes**

The mlock and munlock subroutines fail if:

- **ENOMEM** Some or all of the address range specified by the addr and len parameters does not correspond to valid mapped pages in the address space of the process.
- **EINVAL** The process has already some plocked memory or the len parameter is negative.
- **EPERM** The calling process does not have the appropriate privilege to perform the requested operation.

The mlock subroutine might fail if:

- **ENOMEM** Locking the pages mapped by the specified range would exceed the limit on the amount of memory the process may lock.

**Related Information**

- ["exec: execl, execlp, execv, execvp, or exec Subroutine" on page 235](#)
- ["exit, atexit, unatexit, exit, or _Exit Subroutine" on page 242](#)
- ["fork, _fork, or vfork Subroutine" on page 287](#)
- ["mlockall and munlockall Subroutine" and "munmap Subroutine" on page 884](#)

**mlockall and munlockall Subroutine**

**Purpose**

Locks or unlocks the address space of a process.

**Library**

Standard C Library (libc.a)
Syntax

```c
#include <sys/mman.h>

int mlockall (flags);
int flags;

int munlockall (void);
```

Description

The `mlockall` subroutine causes all of the pages mapped by the address space of a process to be memory-resident until unlocked or until the process exits or executes another process image. The `flags` parameter determines whether the pages to be locked are those currently mapped by the address space of the process, those that are mapped in the future, or both. The `flags` parameter is constructed from the bitwise-inclusive OR of one or more of the following symbolic constants, defined in the `sys/mman.h` header file:

- **MCL_CURRENT**
  - Lock all of the pages currently mapped into the address space of the process.

- **MCL_FUTURE**
  - Lock all of the pages that become mapped into the address space of the process in the future, when those mappings are established.

When **MCL_FUTURE** is specified, the future mapping functions might fail if the system is not able to lock this amount of memory because of lack of resources, for example.

The `munlockall` subroutine unlocks all currently mapped pages of the address space of the process. Any pages that become mapped into the address space of the process after a call to the `munlockall` subroutine are not locked, unless there is an intervening call to the `mlockall` subroutine specifying **MCL_FUTURE** or a subsequent call to the `mlockall` subroutine specifying **MCL_CURRENT**. If pages mapped into the address space of the process are also mapped into the address spaces of other processes and are locked by those processes, the locks established by the other processes are unaffected by a call to the `munlockall` subroutine.

Regarding libraries that are pinned, a distinction has been made internally between a user referencing memory to perform a task related to the application and the system referencing memory on behalf of the application. The former is pinned, and the latter is not. The user-addressable loader data that remains unlocked includes:

- loader entries
- user loader entries
- page-descriptor segment
- usla heap segment
- usla text segment
- all the global segments related to the 64-bit shared library loadlist (shlib heap segment, shlib le segment, shlib text and data heap segments).

This limit affects implementation only, and it does not cause the API to fail.

Upon successful return from a `mlockall` subroutine that specifies **MCL_CURRENT**, all currently mapped pages of the process’ address space are memory-resident and locked. Upon return from the `munlockall` subroutine, all currently mapped pages of the process’ address space are unlocked with respect to the process’ address space.

The calling process must have the root user authority to use this subroutine.
Parameters

flags   Determines whether the pages to be locked are those currently mapped by the address space of the process, those that are mapped in the future, or both.

Return Values

Upon successful completion, the `mlockall` subroutine returns 0. Otherwise, no additional memory is locked, and the subroutine returns -1 and sets `errno` to indicate the error.

Upon successful completion, the `munlockall` subroutine returns 0. Otherwise, no additional memory is unlocked, and the subroutine returns -1 and sets `errno` to indicate the error.

Error Codes

The `mlockall` subroutine fails if:

- **EINVAL** The `flags` parameter is 0, or includes unimplemented flags or the process has already some plocked memory.
- **ENOMEM** Locking all of the pages currently mapped into the address space of the process would exceed the limit on the amount of memory that the process may lock.
- **EPERM** The calling process does not have the appropriate authority to perform the requested operation.

The `munlockall` subroutine fails if:

- **EINVAL** The process has already some plocked memory
- **EPERM** The calling process does not have the appropriate privilege to perform the requested operation

Error Codes

The `mlockall` subroutine fails if:

- **EINVAL** The `flags` parameter is 0, or includes unimplemented flags or the process has already some plocked memory.
- **ENOMEM** Locking all of the pages currently mapped into the address space of the process would exceed the limit on the amount of memory that the process may lock.
- **EPERM** The calling process does not have the appropriate authority to perform the requested operation.

The `munlockall` subroutine fails if:

- **EINVAL** The process has already some plocked memory
- **EPERM** The calling process does not have the appropriate privilege to perform the requested operation

Related Information

"exec: execl, exect, execle, execvp, or exect Subroutine" on page 235, "exit, atexit, unatexit, _exit, or _Exit Subroutine" on page 242, "fork, _fork, or vfork Subroutine" on page 287, "mlock and munlock Subroutine" on page 831, and "munmap Subroutine" on page 884.

mmap or mmap64 Subroutine

Purpose
Maps a file-system object into virtual memory.

Library
Standard C library (libc.a)

Syntax

```c
#include <sys/types.h>
#include <sys/mman.h>

void *mmap (addr, len, prot, flags, fildes, off)
void *addr;
size_t len;
int prot, flags, fildes;
off_t off;
```
**Description**

**Attention:** A file-system object should not be simultaneously mapped using both the `mmap` and `shmat` subroutines. Unexpected results may occur when references are made beyond the end of the object.

The `mmap` subroutine creates a new mapped file or anonymous memory region by establishing a mapping between a process-address space and a file-system object. Care needs to be taken when using the `mmap` subroutine if the program attempts to map itself. If the page containing executing instructions is currently referenced as data through an mmap mapping, the program will hang. Use the `-H4096` binder option, and that will put the executable text on page boundaries. Then reset the file that contains the executable material, and view via an `mmap` mapping.

A region created by the `mmap` subroutine cannot be used as the buffer for read or write operations that involve a device. Similarly, an `mmap` region cannot be used as the buffer for operations that require either a `pin` or `xmattach` operation on the buffer.

Modifications to a file-system object are seen consistently, whether accessed from a mapped file region or from the `read` or `write` subroutine.

Child processes inherit all mapped regions from the parent process when the `fork` subroutine is called. The child process also inherits the same sharing and protection attributes for these mapped regions. A successful call to any `exec` subroutine will unmap all mapped regions created with the `mmap` subroutine.

The `mmap64` subroutine is identical to the `mmap` subroutine except that the starting offset for the file mapping is specified as a 64-bit value. This permits file mappings which start beyond `OFF_MAX`.

In the large file enabled programming environment, `mmap` is redefined to be `mmap64`.

If the application has requested SPEC1170 compliant behavior then the `st_atime` field of the mapped file is marked for update upon successful completion of the `mmap` call.

If the application has requested SPEC1170 compliant behavior then the `st_ctime` and `st_mtime` fields of a file that is mapped with `MAP_SHARED` and `PROT_WRITE` are marked for update at the next call to `msync` subroutine or `munmap` subroutine if the file has been modified.

**Parameters**

- `addr` Specifies the starting address of the memory region to be mapped. When the `MAP_FIXED` flag is specified, this address must be a multiple of the page size returned by the `sysconf` subroutine using the `_SC_PAGE_SIZE` value for the `Name` parameter. A region is never placed at address zero, or at an address where it would overlap an existing region.

- `len` Specifies the length, in bytes, of the memory region to be mapped. The system performs mapping operations over whole pages only. If the `len` parameter is not a multiple of the page size, the system will include in any mapping operation the address range between the end of the region and the end of the page containing the end of the region.
Specifies the access permissions for the mapped region. The `sys/mman.h` file defines the following access options:

- **PROT_READ**
  Region can be read.
- **PROT_WRITE**
  Region can be written.
- **PROT_EXEC**
  Region can be executed.
- **PROT_NONE**
  Region cannot be accessed.

The `prot` parameter can be the `PROT_NONE` flag, or any combination of the `PROT_READ` flag, `PROT_WRITE` flag, and `PROT_EXEC` flag logically ORed together. If the `PROT_NONE` flag is not specified, access permissions may be granted to the region in addition to those explicitly requested. However, write access will not be granted unless the `PROT_WRITE` flag is specified.

**Note:** The operating system generates a `SIGSEGV` signal if a program attempts an access that exceeds the access permission given to a memory region. For example, if the `PROT_WRITE` flag is not specified and a program attempts a write access, a `SIGSEGV` signal results.

If the region is a mapped file that was mapped with the `MAP_SHARED` flag, the `mmap` subroutine grants read or execute access permission only if the file descriptor used to map the file was opened for reading. It grants write access permission only if the file descriptor was opened for writing.

If the region is a mapped file that was mapped with the `MAP_PRIVATE` flag, the `mmap` subroutine grants read, write, or execute access permission only if the file descriptor used to map the file was opened for reading. If the region is an anonymous memory region, the `mmap` subroutine grants all requested access permissions.

**fildes**

Specifies the file descriptor of the file-system object or of the shared memory object to be mapped. If the `MAP_ANONYMOUS` flag is set, the `fildes` parameter must be -1. After the successful completion of the `mmap` subroutine, the file or the shared memory object specified by the `fildes` parameter can be closed without affecting the mapped region or the contents of the mapped file. Each mapped region creates a file reference, similar to an open file descriptor, which prevents the file data from being deallocated.

**Note:** The `mmap` subroutine supports the mapping of shared memory object and regular files only. An `mmap` call that specifies a file descriptor for a special file fails, returning the `ENODEV` error code. An example of a file descriptor for a special file is one that might be used for mapping either I/O or device memory.

**off**

Specifies the file byte offset at which the mapping starts. This offset must be a multiple of the page size returned by the `sysconf` subroutine using the `_SC_PAGE_SIZE` value for the `Name` parameter.
flags Specifies attributes of the mapped region. Values for the flags parameter are constructed by a
bitwise-inclusive ORing of values from the following list of symbolic names defined in the sys/mman.h
file:

MAP_FILE
Specifies the creation of a new mapped file region by mapping the file associated with the
fildes file descriptor. The mapped region can extend beyond the end of the file, both at the time
when the mmap subroutine is called and while the mapping persists. This situation could occur
if a file with no contents was created just before the call to the mmap subroutine, or if a file
was later truncated. However, references to whole pages following the end of the file result in
the delivery of a SIGBUS signal. Only one of the MAP_FILE and MAP_ANONYMOUS flags
must be specified with the mmap subroutine.

MAP_ANONYMOUS
Specifies the creation of a new, anonymous memory region that is initialized to all zeros. This
memory region can be shared only with the descendants of the current process. When using
this flag, the fildes parameter must be -1. Only one of the MAP_FILE and MAP_ANONYMOUS
flags must be specified with the mmap subroutine.

MAP_VARIABLE
Specifies that the system select an address for the new memory region if the new memory
region cannot be mapped at the address specified by the addr parameter, or if the addr
parameter is null. Only one of the MAP_VARIABLE and MAP_FIXED flags must be specified
with the mmap subroutine.

MAP_FIXED
Specifies that the mapped region be placed exactly at the address specified by the addr
parameter. If the application has requested SPEC1170 compliant behavior and the mmap
request is successful, the mapping replaces any previous mappings for the process' pages in
the specified range. If the application has not requested SPEC1170 compliant behavior and a
previous mapping exists in the range then the request fails. Only one of the MAP_VARIABLE
and MAP_FIXED flags must be specified with the mmap subroutine.

MAP_SHARED
When the MAP_SHARED flag is set, modifications to the mapped memory region will be
visible to other processes that have mapped the same region using this flag. If the region is a
mapped file region, modifications to the region will be written to the file.

You can specify only one of the MAP_SHARED or MAP_PRIVATE flags with the mmap
subroutine. MAP_PRIVATE is the default setting when neither flag is specified unless you
request SPEC1170 compliant behavior. In this case, you must choose either MAP_SHARED or
MAP_PRIVATE.

MAP_PRIVATE
When the MAP_PRIVATE flag is specified, modifications to the mapped region by the calling
process are not visible to other processes that have mapped the same region. If the region is a
mapped file region, modifications to the region are not written to the file.

If this flag is specified, the initial write reference to an object page creates a private copy of
that page and redirects the mapping to the copy. Until then, modifications to the page by
processes that have mapped the same region with the MAP_SHARED flag are visible.

You can specify only one of the MAP_SHARED or MAP_PRIVATE flags with the mmap
subroutine. MAP_PRIVATE is the default setting when neither flag is specified unless you
request SPEC1170 compliant behavior. In this case, you must choose either MAP_SHARED or
MAP_PRIVATE.

Return Values
If successful, the mmap subroutine returns the address at which the mapping was placed. Otherwise, it
returns -1 and sets the errno global variable to indicate the error.
Error Codes

Under the following conditions, the mmap subroutine fails and sets the errno global variable to:

- **EACCES** The file referred to by the fildes parameter is not open for read access, or the file is not open for write access and the PROT_WRITE flag was specified for a MAP_SHARED mapping operation. Or, the file to be mapped has enforced locking enabled and the file is currently locked.

- **EAGAIN** The fildes parameter refers to a device that has already been mapped.

- **EBADF** The fildes parameter is not a valid file descriptor, or the MAP_ANONYMOUS flag was set and the fildes parameter is not -1.

- **EFILE** The mapping requested extends beyond the maximum file size associated with fildes.

- **EINVAL** The flags or prot parameter is invalid, or the addr parameter or off parameter is not a multiple of the page size returned by the sysconf subroutine using the _SC_PAGE_SIZE value for the Name parameter.

- **EINVAL** The application has requested SPEC1170 compliant behavior and the value of flags is invalid (neither MAP_PRIVATE nor MAP_SHARED is set).

- **EMFILE** The application has requested SPEC1170 compliant behavior and the number of mapped regions would exceed a implementation-dependent limit (per process or per system).

- **ENODEV** The fildes parameter refers to an object that cannot be mapped, such as a terminal.

- **ENOMEM** There is not enough address space to map len bytes, or the application has not requested Single UNIX Specification, Version 2 compliant behavior and the MAP_FIXED flag was set and part of the address-space range (addr, addr+len) is already allocated.

- **ENXIO** The addresses specified by the range (off, off+len) are invalid for the fildes parameter.

- **EOVERFLOW** The mapping requested extends beyond the offset maximum for the file description associated with fildes.

Related Information

The exec subroutine, fork subroutine, munmap subroutine, read subroutine, shm_open subroutine, shm_unlink subroutine, shmat subroutine, sysconf subroutine, write subroutine.

The pin kernel service, xattach kernel service.

### mntctl Subroutine

#### Purpose

Returns information about the mount status of the system.

#### Library

Standard C Library (libc.a)

#### Syntax

```c
#include <sys/types.h>
#include <sys/mntctl.h>
#include <sys/vmount.h>

int mntctl (Command, Size, Buffer)
int Command;
int Size;
char *Buffer;
```

838 Technical Reference, Volume 1: Base Operating System and Extensions
Description

The `mntctl` subroutine is used to query the status of virtual file systems (also known as mounted file systems).

Each virtual file system (VFS) is described by a `vmount` structure. This structure is supplied when the VFS is created by the `vmount` subroutine. The `vmount` structure is defined in the `sys/vmount.h` file.

Parameters

- **Command**: Specifies the operation to be performed. Valid commands are defined in the `sys/vmount.h` file. At present, the only command is:
  - `MCTL_QUERY` - Query mount information.

- **Buffer**: Points to a data area that will contain an array of `vmount` structures. This data area holds the information returned by the query command. Since the `vmount` structure is variable-length, it is necessary to reference the `vmt_length` field of each structure to determine where in the `Buffer` area the next structure begins.

- **Size**: Specifies the length, in bytes, of the buffer pointed to by the `Buffer` parameter.

Return Values

If the `mntctl` subroutine is successful, the number of `vmount` structures copied into the `Buffer` parameter is returned. If the `Size` parameter indicates the supplied buffer is too small to hold the `vmount` structures for all the current VFSs, the `mntctl` subroutine sets the first word of the `Buffer` parameter to the required size (in bytes) and returns the value 0. If the `mntctl` subroutine otherwise fails, a value of -1 is returned, and the `errno` global variable is set to indicate the error.

Error Codes

The `mntctl` subroutine fails and the requested operation is not performed if one or both of the following are true:

- `EINVAL` - The `Command` parameter is not `MCTL_QUERY`, or the `Size` parameter is not a positive value.
- `EFAULT` - The `Buffer` parameter points to a location outside of the allocated address space of the process.

Related Information

The `uvmount` or `umount` subroutine, `vmount` or `mount` subroutine.

**Files, Directories, and File Systems for Programmers** in *AIX 5L Version 5.3 General Programming Concepts: Writing and Debugging Programs.*

---

### modf, modff, or modfl Subroutine

**Purpose**

Decomposes a floating-point number.

**Syntax**

```c
#include <math.h>

float modff (x, iptr)
float x;
float *iptr;

double modf (x, iptr)
```

Base Operating System (BOS) Runtime Services (A-P) 839
double x, *iptr;
long double modfl (x, iptr)
long double x, *iptr;

Description
The **modff**, **modf**, and **modfl** subroutines break the **x** parameter into integral and fractional parts, each of which has the same sign as the argument. It stores the integral part as a floating-point value in the object pointed to by **iptr**.

Parameters

- **x** Specifies the value to be computed.
- **iptr** Points to the object where the integral part is stored.

Return Values

Upon successful completion, the **modff**, **modf**, and **modfl** subroutines return the signed fractional part of **x**.

If **x** is NaN, a NaN is returned, and *iptr is set to a NaN.

If **x** is ±Inf, ±0 is returned, and *iptr is set to ±Inf.

Related Information

- “class, _class, finite, isnan, or unordered Subroutines” on page 167 and “ldexp, ldexpf, or ldexpl Subroutine” on page 693
- **math.h** in AIX 5L Version 5.3 Files Reference.
- Subroutines Overview in AIX 5L Version 5.3 General Programming Concepts: Writing and Debugging Programs.
- 128-Bit long Double Floating-Point Format in AIX 5L Version 5.3 General Programming Concepts: Writing and Debugging Programs.

**moncontrol Subroutine**

**Purpose**

Starts and stops execution profiling after initialization by the **monitor** subroutine.

**Library**

Standard C Library (libc.a)

**Syntax**

```c
#include <mon.h>

int moncontrol ( Mode )
int Mode;
```

**Description**

The **moncontrol** subroutine starts and stops profiling after profiling has been initialized by the **monitor** subroutine. It may be used with either -p or -pg profiling. When **moncontrol** stops profiling, no output data
file is produced. When profiling has been started by the monitor subroutine and the exit subroutine is called, or when the monitor subroutine is called with a value of 0, then profiling is stopped, and an output file is produced, regardless of the state of profiling as set by the moncontrol subroutine.

The moncontrol subroutine examines global and parameter data in the following order:

1. When the _mondata.prof_type global variable is neither -1 (-p profiling defined) nor +1 (-pg profiling defined), no action is performed, 0 is returned, and the function is considered complete.
   
   The global variable is set to -1 in the mcrt0.o file and to +1 in the gcrt0.o file and defaults to 0 when the crt0.o file is used.

2. When the Mode parameter is 0, profiling is stopped. For any other value, profiling is started.
   
   The following global variables are used in a call to the profil subroutine:

   _mondata.ProfBuf
   _mondata.ProfBufSiz
   _mondata.ProfLoPC
   _mondata.ProfScale

   These variables are initialized by the monitor subroutine each time it is called to start profiling.

Parameters

Mode Specifies whether to start (resume) or stop profiling.

Return Values

The moncontrol subroutine returns the previous state of profiling. When the previous state was STOPPED, a 0 is returned. When the previous state was STARTED, a 1 is returned.

Error Codes

When the moncontrol subroutine detects an error from the call to the profil subroutine, a -1 is returned.

Related Information

The monitor subroutine, monstartup subroutine, profil subroutine.

List of Memory Manipulation Services in AIX 5L Version 5.3 General Programming Concepts: Writing and Debugging Programs.

monitor Subroutine

Purpose

Starts and stops execution profiling using data areas defined in the function parameters.

Library

Standard C Library (libc.a)
Syntax

```c
#include <mon.h>

int monitor (LowProgramCounter, HighProgramCounter, Buffer, BufferSize, NFunction)

OR

int monitor (NotZeroA, DoNotCareA, Buffer, -1, NFunction)

OR

int monitor ((caddr_t)0)
```

caddr_t LowProgramCounter, HighProgramCounter;
HISTCOUNTER *Buffer;
int BufferSize, NFunction;
caddr_t NotZeroA, DoNotCareA;

Description

The `monitor` subroutine initializes the buffer area and starts profiling, or else stops profiling and writes out the accumulated profiling data. Profiling, when started, causes periodic sampling and recording of the program location within the program address ranges specified. Profiling also accumulates function call count data compiled with the `-p` or `-pg` option.

Executable programs created with the `cc` `-p` or `cc` `-pg` command automatically include calls to the `monitor` subroutine (through the `monstartup` and `exit` subroutines) to profile the complete user program, including system libraries. In this case, you do not need to call the `monitor` subroutine.

The `monitor` subroutine is called by the `monstartup` subroutine to begin profiling and by the `exit` subroutine to end profiling. The `monitor` subroutine requires a global data variable to define which kind of profiling, `-p` or `-pg`, is in effect. The `monitor` subroutine initializes four global variables that are used as parameters to the `profil` subroutine by the `moncontrol` subroutine:

- The `monitor` subroutine calls the `moncontrol` subroutine to start the profiling data gathering.
- The `moncontrol` subroutine calls the `profil` subroutine to start the system timer-driven program address sampling.
- The `prof` command processes the data file produced by `-p` profiling.
- The `gprof` command processes the data file produced by `-pg` profiling.

The `monitor` subroutine examines the global data and parameter data in this order:

1. When the `_mondata.prof_type` global variable is neither -1 (`-p` profiling defined) nor +1 (`-pg` profiling defined), an error is returned, and the function is considered complete.
   The global variable is set to -1 in the `mcr0.o` file and to +1 in the `gcrt0.o` file, and defaults to 0 when the `crt0.o` file is used.
2. When the first parameter to the `monitor` subroutine is 0, profiling is stopped and the data file is written out.
   If `-p` profiling was in effect, then the file is named `mon.out`. If `-pg` profiling was in effect, the file is named `gmon.out`. The function is complete.
3. When the first parameter to the `monitor` subroutine is not , the `monitor` parameters and the profiling global variable, `_mondata.prof_type`, are examined to determine how to start profiling.
4. When the `BufferSize` parameter is not -1, a single program address range is defined for profiling, and the first `monitor` definition in the syntax is used to define the single program range.
5. When the *BufferSize* parameter is -1, multiple program address ranges are defined for profiling, and the second *monitor* definition in the syntax is used to define the multiple ranges. In this case, the *ProfileBuffer* value is the address of an array of *prof* structures. The size of the *prof* array is denoted by a zero value for the *HighProgramCounter* \((p\_high)\) field of the last element of the array. Each element in the array, except the last, defines a single programming address range to be profiled. Programming ranges must be in ascending order of the program addresses with ascending order of the *prof* array index. Program ranges may not overlap.

The buffer space defined by the *p_buff* and *p_bufsize* fields of all of the *prof* entries must define a single contiguous buffer area. Space for the function-count data is included in the first range buffer. Its size is defined by the *NFunction* parameter. The *p_scale* entry in the *prof* structure is ignored. The *prof* structure is defined in the `mon.h` file. It contains the following fields:

```c
    caddr_t p_low;        /* low sampling address */
    caddr_t p_high;       /* high sampling address */
    HISTCOUNTER *p_buff;  /* address of sampling buffer */
    int p_bufsize;        /* buffer size- monitor/HISTCOUNTERs,\n                         profil/bytes */
    uint p_scale;         /* scale factor */
```

### Parameters

**LowProgramCounter**

* (prof name: *p_low*)

Defines the lowest execution-time program address in the range to be profiled. The value of the *LowProgramCounter* parameter cannot be 0 when using the *monitor* subroutine to begin profiling.

**HighProgramCounter**

* (prof name: *p_high*)

Defines the next address after the highest-execution time program address in the range to be profiled.

The program address parameters may be defined by function names or address expressions. If defined by a function name, then a function name expression must be used to dereference the function pointer to get the address of the first instruction in the function. This is required because the function reference in this context produces the address of the function descriptor. The first field of the descriptor is the address of the function code. See the examples for typical expressions to use.

**Buffer**

* (prof name: *p_buff*)

Defines the beginning address of an array of *BufferSize* HISTCOUNTERs to be used for data collection. This buffer includes the space for the program address-sampling counters and the function-count data areas. In the case of a multiple range specification, the space for the function-count data area is included at the beginning of the first range in the *BufferSize* specification.

**BufferSize**

* (prof name: *p_bufsize*)

Defines the size of the buffer in number of HISTCOUNTERs. Each counter is of type HISTCOUNTER (defined as short in the `mon.h` file). When the buffer includes space for the function-count data area (single range specification and first range of a multi-range specification) the *NFunction* parameter defines the space to be used for the function count data, and the remainder is used for program-address sampling counters for the range defined. The scale for the *profil* call is calculated from the number of counters available for program address-sample counting and the address range defined by the *LowProgramCounter* and *HighProgramCounter* parameters. See the `mon.h` file.
**NFUnction**

Defines the size of the space to be used for the function-count data area. The space is included as part of the first (or only) range buffer.

When `-p` profiling is defined, the `NFUnction` parameter defines the maximum number of functions to be counted. The space required for each function is defined to be:

\[
\text{sizeof(struct poutcnt)}
\]

The `poutcnt` structure is defined in the `mon.h` file. The total function-count space required is:

\[
\text{NFUnction} \times \text{sizeof(struct poutcnt)}
\]

When `-pg` profiling is defined, the `NFUnction` parameter defines the size of the space (in bytes) available for the function-count data structures, as follows:

\[
\begin{align*}
\text{range} & = \text{HighProgramCounter} - \text{LowProgramCounter}; \\
\text{tonum} & = \text{TO_NUM_ELEMENTS( range )}; \\
\text{if} \ ( \text{tonum} < \text{MINARCS} ) \ \text{tonum} & = \text{MINARCS}; \\
\text{if} \ ( \text{tonum} > \text{TO_MAX}-1 ) \ \text{tonum} & = \text{TO_MAX}-1; \\
\text{tosize} & = \text{tonum} \times \text{sizeof( struct tostruct )}; \\
\text{fromsize} & = \text{FROM_STG_SIZE( range )}; \\
\text{rangesize} & = \text{tosize} + \text{fromsize} + \text{sizeof(struct gfctl)};
\end{align*}
\]

This is computed and summed for all defined ranges. In this expression, the functions and variables in capital letters as well as the structures are defined in the `mon.h` file.

**NotZeroA**

Specifies a value of parameter 1, which is any value except 0. Ignored when it is not zero.

**DoNotCareA**

Specifies a value of parameter 2, of any value, which is ignored.

**Return Values**

The `monitor` subroutine returns 0 upon successful completion.

**Error Codes**

If an error is found, the `monitor` subroutine sends an error message to `stderr` and returns -1.

**Examples**

1. This example shows how to profile the main load module of a program with `-p` profiling:

```c
#include <sys/types.h>
#include <mon.h>
main()
{
    extern caddr_t etext; /*system end of main module text symbol*/
    extern int start(); /*first function in main program*/
    extern struct monglobal _mondata; /*profiling global variables*/
    struct desc {
        /*function descriptor fields*/
        caddr_t begin; /*initial code address*/
        caddr_t toc; /*table of contents address*/
        caddr_t env; /*environment pointer*/
    };
    /*function descriptor structure*/
    struct desc *fd; /*pointer to function descriptor*/
    int rc; /*monitor return code*/
```
This example profiles the main program and the libc.a shared library with -p profiling. The range of addresses for the shared libc.a is assumed to be:

low = d0300000
high = d0312244

These two values can be determined from the loadquery subroutine at execution time, or by using a debugger to view the loaded programs’ execution addresses and the loader map.

```c
#include <sys/types.h>
#include <mon.h>

main()
{
    extern caddr_t etext; /*system end of text symbol*/
    extern int start(); /*first function in main program*/
    extern struct monglobal _mondata; /*profiling global variables*/
    struct prof pb[3]; /*prof array of 3 to define 2 ranges*/
    int rc; /*monitor return code*/
    int range; /*program address range for profiling*/
    int numfunc; /*number of functions to count (max)*/
    int numtics; /*number of sample counters*/
    int num4fcnt; /*number of HISTCOUNTERs used for func space*/
    int BufferSize1; /*first range BufferSize*/
    int BufferSize2; /*second range BufferSize*/
    caddr_t llibb0=0xd0300000; /*lib low address (example only)*/
    caddr_t llibb1=0xd0312244; /*lib high address (example only)*/
    numfunc = 400; /*arbitrary number for example*/
    num4fcnt = numfunc*sizeof(struct poutcnt)/HIST_COUNTER_SIZE;
    BufferSize1 = numtics + num4fcnt;
    /*compute second range buffer size*/
    range = llibb1-llibb0;
    BufferSize2 = range / 12; /*counter for every 12 inst bytes for a change*/
    /*allocate buffer space - note: must be single contiguous*/
    numtics = NUM_HIST_COUNTERS(range);
    numfunc = NUM_HIST_COUNTERS(range);
    etext = (caddr_t)etext+(numfunc*sizeof(struct poutcnt));
    rc = monitor((caddr_t)etext, BufferSize1, numfunc);
    if ( rc != 0 ) /*profiling did not start, do error recovery*/
        return(-1);
    rc = monitor((caddr_t)etext+BufferSize1);
    if ( rc != 0 ) /*did not stop correctly, do error recovery*/
        return(-1);
}
```

2. This example profiles the main program and the libc.a shared library with -p profiling. The range of addresses for the shared libc.a is assumed to be:

low = d0300000
high = d0312244

These two values can be determined from the loadquery subroutine at execution time, or by using a debugger to view the loaded programs’ execution addresses and the loader map.
buffer*/
pb[0].p_buff = (HISTCOUNTER *)malloc( (BufferSize1 +BufferSize2)\  
*HIST COUNTER_SIZE);
if ( pb[0].p_buff == NULL ) /*didn't get space - do error\  
recovery here* ;/
    return(-1);
/*set up the first range values*/
pb[0].p_low = *(uint*)start;  /*start of main module*/
pb[0].p_high = (caddr_t)etext;  /*end of main module*/
pb[0].p_BufferSize = BufferSize1;  /*prog addr cnt space + \  
func cnt space*/
/*set up last element marker*/
pb[2].p_high = (caddr_t)0;
_mondata.prof_type = _PROF_TYPE_IS_P;  /*define -p\  
profiling*/
rc = monitor( (caddr_t)1, (caddr_t)1, pb, -1, numfunc); \  
/*start*/
if ( rc != 0 ) /*profiling did not start - do error recovery\  
here*/
    return (-1);
/*other code for analysis ...*/
rc = monitor( (caddr_t)0);  /*stop profiling and write data \  
file mon.out*/
if ( rc != 0 ) /*did not stop correctly - do error recovery\  
here*/
    return (-1);

3. This example shows how to profile contiguously loaded functions beginning at zit up to but not including zot with -pg profiling:

#include <sys/types.h>
#include <mon.h>
main()
{
extern zit();  /*first function to profile*/
extern zot();  /*upper bound function*/
external struct monglobal _mondata;  /*profiling global variables*/
int rc;  /*monstartup return code*/
_mondata.prof_type = _PROF_TYPE_IS_PG;  /*define -pg profiling*/
/*Note cast used to obtain function code addresses*/
rc = monstartup(*((uint *)zit),((uint *)zot));  /*start*/
if ( rc != 0 ) /*profiling did not start, do error recovery\  
here*/
    return(-1);
/*other code for analysis ...*/
exit(0);  /*stop profiling and write data file gmon.out*/
}

Files

mon.out Data file for -p profiling.
gmon.out Data file for -pg profiling.
/usr/include/mon.h Defines the _mondata.prof_type global variable in the monglobal data structure,  
the prof structure, and the functions referred to in the previous examples.

Related Information

The moncontrol ("moncontrol Subroutine" on page 840) subroutine, monstartup ("monstartup Subroutine"  
on page 847) subroutine, profi ("profi Subroutine" on page 1155) subroutine.

The gprof command, prof command.

The _end, _etext, or _edata ("_end, _etext, or _edata Identifier" on page 223) Identifier.
monstartup Subroutine

Purpose
Starts and stops execution profiling using default-sized data areas.

Library
Standard C Library (libc.a)

Syntax
#include <mon.h>

int monstartup (LowProgramCounter, HighProgramCounter)

OR

int monstartup((caddr_t)-1), (caddr_t)FragBuffer)

OR

int monstartup((caddr_t)-1, (caddr_t)0)

caddr_t LowProgramCounter;
caddr_t HighProgramCounter;

Description
The monstartup subroutine allocates data areas of default size and starts profiling. Profiling causes periodic sampling and recording of the program location within the program address ranges specified, and accumulation of function-call count data for functions that have been compiled with the -p or -pg option.

Executable programs created with the cc -p or cc -pg command automatically include a call to the monstartup subroutine to profile the complete user program, including system libraries. In this case, you do not need to call the monstartup subroutine.

The monstartup subroutine is called by the mcr0.o (-p) file or the gcr0.o (-pg) file to begin profiling. The monstartup subroutine requires a global data variable to define whether -p or -pg profiling is to be in effect. The monstartup subroutine calls the monitor subroutine to initialize the data areas and start profiling.

The prof command is used to process the data file produced by -p profiling. The gprof command is used to process the data file produced by -pg profiling.

The monstartup subroutine examines the global and parameter data in the following order:
1. When the _mondata.prof_type global variable is neither -1 (-p profiling defined) nor +1 (-pg profiling defined), an error is returned and the function is considered complete.
   The global variable is set to -1 in the mcr0.o file and to +1 in the gcr0.o file, and defaults to 0 when crt0.o is used.
2. When the LowProgramCounter value is not -1:
   • A single program address range is defined for profiling
     AND
   • The first monstartup definition in the syntax is used to define the program range.
3. When the *LowProgramCounter* value is -1 and the *HighProgramCounter* value is not 0:
   - Multiple program address ranges are defined for profiling
   
   AND
   
   - The second *monstartup* definition in the syntax is used to define multiple ranges. The *HighProgramCounter* parameter, in this case, is the address of a *frag* structure array. The *frag* array size is denoted by a zero value for the *HighProgramCounter* (*p_high*) field of the last element of the array. Each array element except the last defines one programming address range to be profiled. Programming ranges must be in ascending order of the program addresses with ascending order of the *prof* array index. Program ranges may not overlap.

4. When the *LowProgramCounter* value is -1 and the *HighProgramCounter* value is 0:
   - The whole program is defined for profiling
   
   AND
   
   - The third *monstartup* definition in the syntax is used. The program ranges are determined by *monstartup* and may be single range or multirange.

**Parameters**

*LowProgramCounter* (*frag* name: *p_low*)

- Defines the lowest execution-time program address in the range to be profiled.

*HighProgramCounter* (*frag* name: *p_high*)

- Defines the next address after the highest execution-time program address in the range to be profiled.

   The program address parameters may be defined by function names or address expressions. If defined by a function name, then a function name expression must be used to dereference the function pointer to get the address of the first instruction in the function. This is required because the function reference in this context produces the address of the function descriptor. The first field of the descriptor is the address of the function code. See the examples for typical expressions to use.

*FragBuffer*

- Specifies the address of a frag structure array.

**Examples**

1. This example shows how to profile the main load module of a program with `-p` profiling:

```c
#include <sys/types.h>
#include <mon.h>
main()
{
    extern caddr_t etext; /*system end of text symbol*/
    extern int start(); /*first function in main*/
    extern struct monglobal _mondata; /*profiling global variables*/
    struct desc {
        /*function descriptor fields*/
        caddr_t begin; /*initial code address*/
        caddr_t toc; /*table of contents address*/
        caddr_t env; /*environment pointer*/
    }
    ; /*function descriptor structure*/
    struct desc *fd; /*pointer to function*/
    ;
```
int rc; /*monstartup
return code*/
fd = (struct desc *)start; /*init descriptor pointer to
start
function*/
_mondata.prof_type = _PROF_TYPE_IS_P; /*define -p profiling*/
rc = monstartup( fd->begin, (caddr_t)&etext); /*start*/
if ( rc != 0 ) /*profiling did
not start - do
error
recovery here*/ return(-1);
/*other code
for analysis ...*/
return(0); /*stop profiling and
write data\
file
mon.out*/
}

2. This example shows how to profile the complete program with -p profiling:
#include <sys/types.h>
#include <mon.h>
main()
{
extern struct monglobal _mondata; /*profiling global\nvariables*/
int rc; /*monstartup
return code*/
_mondata.prof_type = _PROF_TYPE_IS_P; /*define -p profiling*/
rc = monstartup( (caddr_t)-1, (caddr_t)0); /*start*/
if ( rc != 0 ) /*profiling did
not start - do
error
recovery here*/ return(-1);
/*other code
for analysis ...*/
return(0); /*stop profiling and
write data\
file
mon.out*/
}

3. This example shows how to profile contiguously loaded functions beginning at zit up to but not
including zot with -pg profiling:
#include <sys/types.h>
#include <mon.h>
main()
{
extern zit(); /*first function
to profile*/
extern zot(); /*upper bound
function*/
extern struct monglobal _mondata; /*profiling global variables*/
int rc; /*monstartup
return code*/
_mondata.prof_type = _PROF_TYPE_IS_PG; /*define -pg profiling*/
/*Note cast used to obtain function code addresses*/
rc = monstartup(*(uint *)zit,*(uint *)zot); /*start*/
if ( rc != 0 ) /*profiling did
not start - do
error
recovery here*/
return(-1);
Return Values
The monstartup subroutine returns 0 upon successful completion.

Error Codes
If an error is found, the monstartup subroutine outputs an error message to stderr and returns -1.

Files
- mon.out: Data file for -p profiling.
- gmon.out: Data file for -pg profiling.
- mon.h: Defines the _mondata.prof_type variable in the monglobal data structure, the prof structure, and the functions referred to in the examples.

Related Information
The moncontrol subroutine, monitor subroutine, prof subroutine.
The gprof command, prof command.
The _end, _etext, or _edata identifier.
List of Memory Manipulation Services in AIX 5L Version 5.3 General Programming Concepts: Writing and Debugging Programs.

mprotect Subroutine

Purpose
Modifies access protections for memory mapping or shared memory.

Library
Standard C Library (libc.a)

Syntax
#include <sys/types.h>
#include <sys/mman.h>

int mprotect (void *addr, size_t len, int prot);

Description
The mprotect subroutine modifies the access protection of a mapped file or shared memory region or anonymous memory region created by the mmap subroutine. Processes running in an environment where the MPROTECT_SHM=ON environmental variable is defined can also use the mprotect subroutine to modify the access protection of a shared memory region created by the shmat subroutine, or attached by the shmget subroutine.
The user who protects shared memory with the `mprotect` subroutine must be also be either the user who created the shared memory descriptor, the user who owns the shared memory descriptor, or the root user.

The `mprotect` subroutine can only be used on shared memory regions backed with 4 KB or 64 KB pages; shared memory regions backed by 16 MB and 16 GB pages are not supported by the `mprotect` subroutine. The page size used to back a shared memory region can be obtained using the `vmgetinfo` subroutine and specifying VM_PAGE_INFO for the `command` parameter.

The `mprotect` subroutine cannot be used for shared memory that has been pre-translated. This includes shared memory regions created with the SHM_PIN flag specified to the `shmget` subroutine as well as shared memory regions that have been pinned using the `shmctl` subroutine with the SHM_LOCK flag specified.

**Parameters**

`addr` Specifies the address of the region to be modified. Must be a multiple of the page size backing the memory region.

`len` Specifies the length, in bytes, of the region to be modified. For shared memory regions backed with 4 KB pages, the `len` parameter will be rounded off to the next multiple of the page size. Otherwise, the `len` parameter must be a multiple of the page size backing the memory region.

`prot` Specifies the new access permissions for the mapped region. Legitimate values for the `prot` parameter are the same as those permitted for the `mmap` subroutine, as follows:

- **PROT_READ**
  Region can be read.

- **PROT_WRITE**
  Region can be written.

- **PROT_EXEC**
  Region can be executed.

- **PROT_NONE**
  Region cannot be accessed. PROT_NONE is not a valid `prot` parameter for shared memory attached with the `shmat` subroutine.

**Return Values**

When successful, the `mprotect` subroutine returns 0. Otherwise, it returns -1 and sets the `errno` global variable to indicate the error.

**Note:** The return value for the `mprotect` subroutine is 0 if it fails because the region given was not created by `mmap` unless XPG 1170 behavior is requested by setting the `XPG_SUS_ENV` environment variable to `ON`.

**Error Codes**

If the `mprotect` subroutine is unsuccessful, the `errno` global variable might be set to one of the following values:

**Attention:** If the `mprotect` subroutine is unsuccessful because of a condition other than that specified by the `EINVAL` error code, the access protection for some pages in the `(addr, addr + len)` range might have been changed.

- **EACCES**
  The `prot` parameter specifies a protection that conflicts with the access permission set for the underlying file.
The user is not the creator or owner of the shared memory region and is not the root user.

The `prot` parameter specified is not valid for the region specified.

The `addr` or `len` parameter is not a multiple of the page size backing the memory region.

The application has requested Single UNIX Specification, Version 2 compliant behavior, but addresses in the range are not valid for the address space of the process, or the addresses specify one or more pages that are not attached to the user’s address space by a previous `mmap` or `shmat` subroutine call.

The shared memory region specified is backed by 64 KB pages, but the `addr` or `len` parameter is not 64 KB aligned, or PROT_NONE protection was specified for a shared memory region, or a pre-translated shared memory region was specified, or a shared memory region backed by 16 MB or 16 GB pages was specified.

**Related Information**
The `vmgetinfo` subroutine, `shmget` subroutine, `shmctl` subroutine.

### mq_close Subroutine

**Purpose**
Closes a message queue.

**Library**
Standard C Library (`libc.a`)

**Syntax**
```c
#include <mqueue.h>

int mq_close (mqdes);
```

**Description**
The `mq_close` subroutine removes the association between the message queue descriptor, `mqdes`, and its message queue. The results of using this message queue descriptor after successful return from the `mq_close` subroutine, and until the return of this message queue descriptor from a subsequent `mq_open` call, are undefined.

If the process has successfully attached a notification request to the message queue through the `mqdes` parameter, this attachment is removed, and the message queue is available for another process to attach for notification.

**Parameters**

`mqdes` Specifies the message queue descriptor.

**Return Values**
Upon successful completion, the `mq_close` subroutine returns a zero. Otherwise, the subroutine returns a `-1` and sets `errno` to indicate the error.

**Error Codes**
The `mq_close` subroutine fails if:

**EBADF** The `mqdes` parameter is not a valid message queue descriptor.
mq_getattr Subroutine

Purpose
Gets message queue attributes.

Library
Standard C Library (libc.a)

Syntax
#include <mqueue.h>

int mq_getattr (mqdes, mqstat)
mqd_t mqdes;
struct mq_attr *mqstat;

Description
The mq_getattr subroutine obtains status information and attributes of the message queue and the open message queue description associated with the message queue descriptor.

The results are returned in the mq_attr structure referenced by the mqstat parameter.

Upon return, the following members have the values associated with the open message queue description as set when the message queue was opened and as modified by subsequent calls to the mq_setattr subroutine:
  • mq_flags

The following attributes of the message queue are returned as set at message queue creation:
  • mq_maxmsg
  • mq_msgsize

Upon return, the following member within the mq_attr structure referenced by the mqstat parameter is set to the current state of the message queue:

mq_curmsgs The number of messages currently on the queue.

Parameters

mqdes Specifies a message queue descriptor.

mqstat Points to the mq_attr structure.

Return Values
Upon successful completion, the mq_getattr subroutine returns zero. Otherwise, the subroutine returns -1 and sets errno to indicate the error.
Error Codes

The **mq_getattr** subroutine fails if:

- **EBADF** The `mqdes` parameter is not a valid message queue descriptor.
- **EFAULT** Invalid user address.
- **EINVAL** The `mqstat` parameter value is not valid.
- **ENOMEM** Insufficient memory for the required operation.
- **ENOTSUP** This function is not supported with processes that have been checkpoint-restart'ed.

Related Information

“mq_open Subroutine” on page 855 and “mq_setattr Subroutine” on page 860.

mq_notify Subroutine

Purpose

Notifies a process that a message is available.

Library

Standard C Library (**libc.a**)

Syntax

```c
#include <mqueue.h>

int mq_notify(mqdes, notification)
mqd_t mqdes;
const struct sigevent *notification;
```

Description

If the `notification` parameter is not NULL, the **mq_notify** subroutine registers the calling process to be notified of message arrival at an empty message queue associated with the specified message queue descriptor, `mqdes`. The notification specified by the `notification` parameter is sent to the process when the message queue transitions from empty to non-empty. At any time only one process may be registered for notification by a message queue. If the calling process or any other process has already registered for notification of message arrival at the specified message queue, subsequent attempts to register for that message queue fails.

If notification is NULL and the process is currently registered for notification by the specified message queue, the existing registration is removed.

When the notification is sent to the registered process, its registration is removed. The message queue is then available for registration.

If a process has registered for notification of message arrival at a message queue and a thread is blocked in the **mq_receive** subroutine waiting to receive a message, the arriving message satisfies the appropriate **mq_receive**. The resulting behavior is as if the message queue remains empty, and no notification is sent.

Parameters

- `mqdes` Specifies a message queue descriptor.
- `notification` Points to the `sigevent` structure.
Return Values
Upon successful completion, the `mq_notify` subroutine returns a zero. Otherwise, it returns a value of -1 and sets `errno` to indicate the error.

Error Codes
The `mq_notify` subroutine fails if:
- **EBADF** The `mqdes` parameter is not a valid message queue descriptor.
- **EBUSY** A process is already registered for notification by the message queue.
- **EFAULT** Invalid used address.
- **ENOMEM** Insufficient memory for the required operation.
- **ENOTSUP** This function is not supported with processes that have been checkpoint-restart'ed.
- **EINVAL** The current process is not registered for notification for the specified message queue and registration removal was requested.

Related Information
“mq_open Subroutine.”

mq_open Subroutine

Purpose
Opens a message queue.

Library
Standard C Library (`libc.a`)

Syntax
```c
#include <mqueue.h>

mqd_t mq_open(const char *name, int oflag, mode_t mode, mq_attr *attr);
```

Description
The `mq_open` subroutine establishes a connection between a process and a message queue with a message queue descriptor. It creates an open message queue description that refers to the message queue, and a message queue descriptor that refers to that open message queue description. The message queue descriptor is used by other subroutines to refer to that message queue.

The `name` parameter points to a string naming a message queue, and has no representation in the file system. The `name` parameter conforms to the construction rules for a pathname. It may or may not begin with a slash character, but contains at least one character. Processes calling the `mq_open` subroutine with the same value of `name` refer to the same message queue object, as long as that name has not been removed. If the `name` parameter is not the name of an existing message queue and creation is not requested, the `mq_open` subroutine will fail and return an error.

The `oflag` parameter requests the desired receive and send access to the message queue. The requested access permission to receive messages or send messages is granted if the calling process would be granted read or write access, respectively, to an equivalently protected file.
The value of the oflag parameter is the bitwise-inclusive OR of values from the following list. Applications specify exactly one of the first three values (access modes) below in the value of the oflag parameter:

**O_RDONLY**
- Open the message queue for receiving messages. The process can use the returned message queue descriptor with the `mq_receive` subroutine, but not the `mq_send` subroutine. A message queue may be open multiple times in the same or different processes for receiving messages.

**O_WRONLY**
- Open the queue for sending messages. The process can use the returned message queue descriptor with the `mq_send` subroutine but not the `mq_receive` subroutine. A message queue may be open multiple times in the same or different processes for sending messages.

**O_RDWR**
- Open the queue for both receiving and sending messages. The process can use any of the functions allowed for the **O_RDONLY** and **O_WRONLY** flags. A message queue may be open multiple times in the same or different processes for sending messages.

Any combination of the remaining flags may be specified in the value of the oflag parameter:

**O_CREAT**
- Create a message queue. It requires two additional arguments: mode, which is of `mode_t` type, and attr, which is a pointer to an `mq_attr` structure. If the pathname name has already been used to create a message queue that still exists, this flag has no effect, except as noted under the **O_EXCL** flag. Otherwise, a message queue is created without any messages in it. The user ID of the message queue is set to the effective user ID of the process, and the group ID of the message queue is set to the effective group ID of the process. The file permission bits are set to the value of mode. When bits in the mode parameter other than file permission bits are set, they have no effect. If attr is NULL, the message queue is created with default message queue attributes. Default values are 128 for `mq_maxmsg` and 1024 for `mq_msgsize`. If attr is non-NULL, the message queue `mq_maxmsg` and `mq_msgsize` attributes are set to the values of the corresponding members in the `mq_attr` structure referred to by attr.

**O_EXCL**
- If the **O_EXCL** and **O_CREAT** flags are set, the `mq_open` subroutine fails if the message queue name exists. The check for the existence of the message queue and the creation of the message queue if it does not exist is atomic with respect to other threads executing `mq_open` naming the same name with the **O_EXCL** and **O_CREAT** flags set. If the **O_EXCL** flag is set and the **O_CREAT** flag is not set, the **O_EXCL** flag is ignored.

**O_NONBLOCK**
- Determines whether the `mq_send` or `mq_receive` subroutine waits for resources or messages that are not currently available, or fails with errno set to EAGAIN; see `mq_send Subroutine` on page 858 and `mq_receive Subroutine` on page 857 for details.

The `mq_open` subroutine does not add or remove messages from the queue.

**Parameters**

- **name**: Points to a string naming a message queue.
- **oflag**: Requests the desired receive and send access to the message queue.
- **mode**: Specifies the value of the file permission bits. Used with **O_CREAT** to create a message queue.
- **attr**: Points to an `mq_attr` structure. Used with **O_CREAT** to create a message queue.

**Return Values**

Upon successful completion, the `mq_open` subroutine returns a message queue descriptor. Otherwise, it returns `(mqd_t)-1` and sets errno to indicate the error.
Error Codes
The `mq_open` subroutine fails if:

- **EACCES** The message queue exists and the permissions specified by the `oflag` parameter are denied.
- **EEXIST** The `O_CREAT` and `O_EXCL` flags are set and the named message queue already exists.
- **EFAULT** Invalid used address.
- **EINVAL** The `mq_open` subroutine is not supported for the given name.
- **EINVAL** The `oflag` parameter value is not valid.
- **EMFILE** Too many message queue descriptors are currently in use by this process.
- **ENAME_TOO_LONG** The length of the `name` parameter exceeds `PATH_MAX` or a pathname component is longer than `NAME_MAX`.
- **ENFILE** Too many message queues are currently open in the system.
- **ENOENT** The `O_CREAT` flag is not set and the named message queue does not exist.
- **ENOMEM** Insufficient memory for the required operation.
- **ENOSPC** There is insufficient space for the creation of the new message queue.
- **ENOTSUP** This function is not supported with processes that have been checkpoint-restart'ed.

Related Information

**mq_receive Subroutine**

**Purpose**
Receives a message from a message queue.

**Library**
Standard C Library (`libc.a`)

**Syntax**

```c
#include <mqueue.h>

ssize_t mq_receive (mqdes, msg_ptr, msg_len, msg_prio);

mqd_t mqdes;
char *msg_ptr;
size_t msg_len;
unsigned *msg_prio;
```

**Description**
The `mq_receive` subroutine receives the oldest of the highest priority messages from the message queue specified by the `mqdes` parameter. If the size of the buffer in bytes, specified by the `msg_len` parameter, is less than the `mq_msgsize` attribute of the message queue, the subroutine fails and returns an error. Otherwise, the selected message is removed from the queue and copied to the buffer pointed to by the `msg_ptr` parameter.

If the `msg_prio` parameter is not NULL, the priority of the selected message is stored in the location referenced by `msg_prio`. 
If the specified message queue is empty and the O_NONBLOCK flag is not set in the message queue description associated with the mqdes parameter, the mq_receive subroutine blocks until a message is enqueued on the message queue or until mq_receive is interrupted by a signal. If more than one thread is waiting to receive a message when a message arrives at an empty queue and the Priority Scheduling option is supported, the thread of highest priority that has been waiting the longest is selected to receive the message. If the specified message queue is empty and the O_NONBLOCK flag is set in the message queue description associated with the mqdes parameter, no message is removed from the queue, and the mq_receive subroutine returns an error.

**Parameters**

- **mqdes**: Specifies the message queue descriptor.
- **msg_ptr**: Points to the buffer where the message is copied.
- **msg_len**: Specifies the length of the message, in bytes.
- **msg_prio**: Stores the priority of the selected message.

**Return Values**

Upon successful completion, the mq_receive subroutine returns the length of the selected message in bytes and the message is removed from the queue. Otherwise, no message is removed from the queue, and the subroutine returns -1 and sets errno to indicate the error.

**Error Codes**

The mq_receive subroutine fails if:

- **EAGAIN**: The O_NONBLOCK flag was set in the message description associated with the mqdes parameter, and the specified message queue is empty.
- **EBADF**: The mqdes parameter is not a valid message queue descriptor open for reading.
- **EFAULT**: Invalid used address.
- **EIDRM**: The specified message queue was removed during the required operation.
- **EINVAL**: The msg_ptr parameter is null.
- **EMSGSIZE**: The specified message buffer size, msg_len, is less than the message size attribute of the message queue.
- **ENOMEM**: Insufficient memory for the required operation.
- **ENOTSUP**: This function is not supported with processes that have been checkpoint-restart'ed.

**Related Information**

See "mq_open Subroutine" on page 855 and "mq_send Subroutine."
mqd_t mqdes;
const char *msg_ptr;
size_t msg_len;
unsigned *msg_prio;

Description
The `mq_send` subroutine adds the message pointed to by the `msg_ptr` parameter to the message queue specified by the `mqdes` parameter. The `msg_len` parameter specifies the length of the message, in bytes, pointed to by `msg_ptr`. The value of `msg_len` is less than or equal to the `mq_msgsize` attribute of the message queue, or the `mq_send` subroutine will fail.

If the specified message queue is not full, the `mq_send` subroutine behaves as if the message is inserted into the message queue at the position indicated by the `msg_prio` parameter. A message with a larger numeric value of `msg_prio` will be inserted before messages with lower values of `msg_prio`. A message will be inserted after other messages in the queue with equal `msg_prio`. The value of `msg_prio` will be less than `MQ_PRIO_MAX`.

If the specified message queue is full and `O_NONBLOCK` is not set in the message queue description associated with `mqdes`, the `mq_send` subroutine will block until space becomes available to enqueue the message, or until `mq_send` is interrupted by a signal. If more than one thread is waiting to send when space becomes available in the message queue and the Priority Scheduling option is supported, the thread of the highest priority that has been waiting the longest is unblocked to send its message. Otherwise, it is unspecified which waiting thread is unblocked. If the specified message queue is full and `O_NONBLOCK` is set in the message queue description associated with `mqdes`, the message is not queued and the `mq_send` subroutine returns an error.

Parameters
- `mqdes`: Specifies the message queue descriptor.
- `msg_ptr`: Points to the message to be added.
- `msg_len`: Specifies the length of the message, in bytes.
- `msg_prio`: Specifies the position of the message in the message queue.

Return Values
Upon successful completion, the `mq_send` subroutine returns a zero. Otherwise, no message is enqueued, the subroutine returns -1, and `errno` is set to indicate the error.

Error Codes
The `mq_send` subroutine fails if:

- `EAGAIN`: The `O_NONBLOCK` flag is set in the message queue description associated with the `mqdes` parameter, and the specified message queue is full (maximum number of messages in the queue or maximum number of bytes in the queue is reached).
- `EBADF`: The `mqdes` parameter is not a valid message queue descriptor open for writing.
- `EFAULT`: Invalid used address.
- `EIDRM`: The specified message queue was removed during the required operation.
- `EINTR`: A signal interrupted the call to the `mq_send` subroutine.
- `EINVAL`: The value of the `msg_prio` parameter was outside the valid range.
- `EINVAL`: The `msg_ptr` parameter is null.
- `EMSGSIZE`: The specified message length, `msg_len`, exceeds the message size attribute of the message queue.
- `ENOMEM`: Insufficient memory for the required operation.
- `ENOTSUP`: This function is not supported with processes that have been checkpoint-restart’ed.
mq_setattr Subroutine

Purpose
Sets message queue attributes.

Library
Standard C Library (libc.a)

Syntax
#include <mqueue.h>

int mq_setattr (mqdes, mqstat, omqstat);

mqd_t mqdes;
const struct mq_attr *mqstat;
struct mq_attr *omqstat;

Description
The mq_setattr subroutine sets attributes associated with the open message queue description referenced by the message queue descriptor specified by mqdes.

The message queue attributes corresponding to the following members defined in the mq_attr structure are set to the specified values upon successful completion of the mq_setattr subroutine.

The value of the mq_flags member is either zero or O_NONBLOCK.

The values of the mq_maxmsg, mq_msgsize, and mq_curmsgs members of the mq_attr structure are ignored by the mq_setattr subroutine.

If the omqstat parameter is non-NULL, the mq_setattr subroutine stores, in the location referenced by omqstat, the previous message queue attributes and the current queue status. These values are the same as would be returned by a call to the mq_getattr subroutine at that point.

Parameters
mqdes Specifies the message queue descriptor.
mqstat Specifies the status of the message queue.
omqstat Specifies the status of the previous message queue.

Return Values
Upon successful completion, the mq_setattr subroutine returns a zero and the attributes of the message queue are changed as specified.

Otherwise, the message queue attributes are unchanged, and the subroutine returns a -1 and sets errno to indicate the error.

Error Codes
The mq_setattr subroutine fails if:

EBADF The mqdes parameter is not a valid message queue descriptor.
EFAULT
Invalid user address.
EINVAL
The mqstat parameter value is not valid.
ENOMEM
Insufficient memory for the required operation.
ENOTSUP
This function is not supported with processes that have been checkpoint-restart’ed.

Related Information
“mq_open Subroutine” on page 855 and “mq_getattr Subroutine” on page 853.

mq_receive, mq_timedreceive Subroutine

Purpose
Receives a message from a message queue (REALTIME).

Syntax

```c
#include <mqueue.h>

ssize_t mq_receive(mqd_t mqdes, char *msg_ptr,
    size_t msg_len, unsigned *msg_prio,

#include <mqueue.h>
#include <time.h>

ssize_t mq_timedreceive(mqd_t mqdes, char *restrict msg_ptr,
    size_t msg_len, unsigned *restrict msg_prio,
    const struct timespec *restrict abs_timeout);
```

Description

The `mq_receive()` function receives the oldest of the highest priority messages from the message queue specified by `mqdes`. If the size of the buffer, in bytes, specified by the `msg_len` argument is less than the `mq_msgsize` attribute of the message queue, the function fails and returns an error. Otherwise, the selected message is removed from the queue and copied to the buffer pointed to by the `msg_ptr` argument.

If the value of `msg_len` is greater than `{SSIZE_MAX}`, the result is implementation-defined.

If the `msg_prio` argument is not NULL, the priority of the selected message is stored in the location referenced by `msg_prio`.

If the specified message queue is empty and O_NONBLOCK is not set in the message queue description associated with `mqdes`, `mq_receive()` blocks until a message is enqueued on the message queue or until `mq_receive()` is interrupted by a signal. If more than one thread is waiting to receive a message when a message arrives at an empty queue and the Priority Scheduling option is supported, then the thread of highest priority that has been waiting the longest is selected to receive the message. Otherwise, it is unspecified which waiting thread receives the message. If the specified message queue is empty and O_NONBLOCK is set in the message queue description associated with `mqdes`, no message is removed from the queue, and `mq_receive()` returns an error.

The `mq_timedreceive()` function receives the oldest of the highest priority messages from the message queue specified by `mqdes` as described for the `mq_receive()` function. However, if O_NONBLOCK was not specified when the message queue was opened by the `mq_open()` function, and no message exists on the queue to satisfy the receive, the wait for such a message is terminated when the specified timeout expires. If O_NONBLOCK is set, this function matches `mq_receive()`.
The timeout expires when the absolute time specified by `abs_timeout` passes—as measured by the clock on which timeouts are based (that is, when the value of that clock equals or exceeds `abs_timeout`), or when the absolute time specified by `abs_timeout` has already been passed at the time of the call.

If the `Timers` option is supported, the timeout is based on the CLOCK_REALTIME clock; if the `Timers` option is not supported, the timeout is based on the system clock as returned by the `time()` function.

The resolution of the timeout matches the resolution of the clock on which it is based. The `timespec` argument is defined in the `<time.h>` header.

The operation never fails with a timeout if a message can be removed from the message queue immediately. The validity of the `abs_timeout` parameter does not need to be checked if a message can be removed from the message queue immediately.

**Return Values**

Upon successful completion, the `mq_receive()` and `mq_timedreceive()` functions return the length of the selected message in bytes and the message is removed from the queue. Otherwise, no message shall be removed from the queue, the functions return a value of -1, and `errno` is set to indicate the error.

**Error Codes**

The `mq_receive()` and `mq_timedreceive()` functions fail if:

- **[EAGAIN]** `O_NONBLOCK` was set in the message description associated with `mqdes`, and the specified message queue is empty.
- **[EBADF]** The `mqdes` argument is not a valid message queue descriptor open for reading.
- **[EFAULT]** `abs_timeout` references invalid memory.
- **[EIDRM]** Specified message queue was removed during required operation.
- **[EINVAL]** The `mq_receive()` or `mq_timedreceive()` operation was interrupted by a signal.
- **[EINVAL]** The `abs_timeout` parameter specified a nanoseconds field value less than 0 or greater than or equal to 1000 million.
- **[EINVAL]** `msg_ptr` value was null.
- **[EMSGSIZE]** The specified message buffer size, `msg_len`, is less than the message size attribute of the message queue.
- **[ENOTSUP]** Function is not supported with checkpoint-restart'ed processes.
- **[ETIMEDOUT]** The `O_NONBLOCK` flag was not set when the message queue was opened, but no message arrived on the queue before the specified timeout expired.

The `mq_receive()` and `mq_timedreceive()` functions might fail if:

- **[EBADMSG]** The implementation has detected a data corruption problem with the message.

**Related Information**

- "mq_send, mq_timedsend Subroutine" on page 863
- "msgctl Subroutine" on page 870
- "msgget Subroutine" on page 872
- "msgsnd Subroutine" on page 873
- "msgsrv Subroutine" on page 876
- "posix_trace_getnext_event, posix_trace_timedgetnext_event, posix_trace_trygetnext_event Subroutine" on page 1143
- "pthread_mutex_timedlock Subroutine" on page 1251
- "pthread_rwlock_timedrdlock Subroutine" on page 1266
- "pthread_rwlock_timedwrlock Subroutine" on page 1268


The `mqueue.h` and `time.h` file.
mq-send, mq_timedsend Subroutine

Purpose
Sends a message to a message queue (REALTIME).

Syntax
```
#include <mqueue.h>

int mq_send(mqd_t mqdes, const char *msg_ptr,
            size_t msg_len, unsigned *msg_prio,

#include <mqueue.h>
#include <time.h>

int mq_timedsend(mqd_t mqdes, const char *msg_ptr,
                 size_t msg_len, unsigned msg_prio,
                 const struct timespec *abs_timeout);
```

Description
The `mq_send()` function adds the message pointed to by the argument `msg_ptr` to the message queue specified by `mqdes`. The `msg_len` argument specifies the length of the message, in bytes, pointed to by `msg_ptr`. The value of `msg_len` is less than or equal to the `mq_msgsize` attribute of the message queue, or `mq_send()` fails.

If the specified message queue is not full, `mq_send()` behaves as if the message is inserted into the message queue at the position indicated by the `msg_prio` argument. A message with a larger numeric value of `msg_prio` is inserted before messages with lower values of `msg_prio`. A message is inserted after other messages in the queue, if any, with equal `msg_prio` values. The value of `msg_prio` is less than `{MQ_PRIO_MAX}`.

If the specified message queue is full and `O_NONBLOCK` is not set in the message queue description associated with `mqdes`, `mq_send()` blocks until space becomes available to enqueue the message, or until `mq_send()` is interrupted by a signal. If more than one thread is waiting to send when space becomes available in the message queue and the `Priority Scheduling` option is supported, then the thread of the highest priority that has been waiting the longest is unblocked to send its message. Otherwise, it is unspecified which waiting thread is unblocked. If the specified message queue is full and `O_NONBLOCK` is set in the message queue description associated with `mqdes`, the message is not queued and `mq_send()` returns an error.

The `mq_timedsend()` function adds a message to the message queue specified by `mqdes` in the manner defined for the `mq_send()` function. However, if the specified message queue is full and `O_NONBLOCK` is not set in the message queue description associated with `mqdes`, the wait for sufficient room in the queue is terminated when the specified timeout expires. If `O_NONBLOCK` is set in the message queue description, this function matches `mq_send()`.

The timeout expires when the absolute time specified by `abs_timeout` passes—as measured by the clock on which timeouts are based (that is, when the value of that clock equals or exceeds `abs_timeout`)—or when the absolute time specified by `abs_timeout` has already been passed at the time of the call.

If the `Timers` option is supported, the timeout is based on the CLOCK_REALTIME clock; if the `Timers` option is not supported, the timeout is based on the system clock as returned by the `time()` function.

The operation never fails with a timeout if there is sufficient room in the queue to add the message immediately. The validity of the `abs_timeout` parameter does not need to be checked when there is sufficient room in the queue.
Application Usage
The value of the symbol {MQ_PRIO_MAX} limits the number of priority levels supported by the application. Message priorities range from 0 to {MQ_PRIO_MAX}-1.

Return Values
Upon successful completion, the `mq_send()` and `mq_timedsend()` functions return a value of 0. Otherwise, no message is enqueued, the functions return -1, and `errno` is set to indicate the error.

Error Codes
The `mq_send()` and `mq_timedsend()` functions fail if:

- **[EAGAIN]** The O_NONBLOCK flag is set in the message queue description associated with `mqdes`, and the specified message queue is full.
- **[EBADF]** The `mqdes` argument is not a valid message queue descriptor open for writing.
- **[EFAULT]** `abs_timeout` references invalid memory.
- **[EIDRM]** Specified message queue was removed during required operation.
- **[EINVAL]** The value of `msg_prio` was outside the valid range.
- **[EINVAL]** `msg_ptr` value was null.
- **[EINVAL]** The process or thread would have blocked, and the `abs_timeout` parameter specified a nanoseconds field value less than 0 or greater than or equal to 1000 million.
- **[EMSGSIZE]** The specified message length, `msg_len`, exceeds the message size attribute of the message queue.
- **[ENOTSUP]** Function is not supported with checkpoint-restart'ed processes.
- **[ETIMEDOUT]** The O_NONBLOCK flag was not set when the message queue was opened, but the timeout expired before the message could be added to the queue.

The `mq_send()` and `mq_timedsend()` functions might fail if:

- **[EBADMSG]** The implementation has detected a data corruption problem with the message.

Related Information

- `mq_receive, mq_timedreceive Subroutine` on page 861
- `msgctl Subroutine` on page 870
- `msgget Subroutine` on page 872
- `msgset Subroutine` on page 873
- `msgsnd Subroutine` on page 876
- `posix_trace_getnext_event, posix_trace_timedgetnext_event, posix_trace_trygetnext_event Subroutine` on page 1143
- `pthread_mutex_timedlock Subroutine` on page 1251
- `pthread_rwlock_timedrdlock Subroutine` on page 1266
- `pthread_rwlock_timedwrlock Subroutine` on page 1268


The `mqueue.h` and `time.h` file.

**mq_unlink Subroutine**

**Purpose**
Removes a message queue.

**Library**
Standard C Library (libc.a)
Syntax
#include <mqueue.h>

int mq_unlink (const char *name);

Description
The `mq_unlink` subroutine removes the message queue named by the pathname `name`. After a successful call to the `mq_unlink` subroutine with the `name` parameter, a call to the `mq_open` subroutine with the `name` parameter and the `O_CREAT` flag will create a new message queue. If one or more processes have the message queue open when the `mq_unlink` subroutine is called, destruction of the message queue is postponed until all references to the message queue have been closed.

After a successful completion of the `mq_unlink` subroutine, calls to the `mq_open` subroutine to recreate a message queue with the same name will succeed. The `mq_unlink` subroutine never blocks even if all references to the message queue have not been closed.

Parameters

`name` Specifies the message queue to be removed.

Return Values
Upon successful completion, the `mq_unlink` subroutine returns a zero. Otherwise, the named message queue is unchanged, and the `mq_unlink` subroutine returns a -1 and sets `errno` to indicate the error.

Error Codes
The `mq_unlink` subroutine fails if:

- **EACCES** Permission is denied to unlink the named message queue.
- **EFAULT** Invalid used address.
- **EINVAL** The `name` parameter value is not valid.
- **ENAMETOOLONG** The length of the `name` parameter exceeds `PATH_MAX` or a pathname component is longer than `NAME_MAX`.
- **ENOENT** The named message queue does not exist.
- **ENOTSUP** This function is not supported with processes that have been checkpoint-restart'ed.

Related Information
- “mq_open Subroutine” on page 855 and “mq_close Subroutine” on page 852

---

### msem_init Subroutine

**Purpose**
Initializes a semaphore in a mapped file or shared memory region.

**Library**
Standard C Library (`libc.a`)

**Syntax**
```c
#include <sys/mman.h>
```
`msemaphore *msem_init (Sem, InitialValue)`

`msemaphore *Sem;`  
`int InitialValue;`

**Description**

The `msem_init` subroutine allocates a new binary semaphore and initializes the state of the new semaphore.

If the value of the `InitialValue` parameter is `MSEM_LOCKED`, the new semaphore is initialized in the locked state. If the value of the `InitialValue` parameter is `MSEM_UNLOCKED`, the new semaphore is initialized in the unlocked state.

The `msemaphore` structure is located within a mapped file or shared memory region created by a successful call to the `mmap` subroutine and having both read and write access.

Whether a semaphore is created in a mapped file or in an anonymous shared memory region, any reference by a process that has mapped the same file or shared region, using an `msemaphore` structure pointer that resolved to the same file or start of region offset, is taken as a reference to the same semaphore.

Any previous semaphore state stored in the `msemaphore` structure is ignored and overwritten.

**Parameters**

- **Sem**  
  Points to an `msemaphore` structure in which the state of the semaphore is stored.

- **Initial Value**  
  Determines whether the semaphore is locked or unlocked at allocation.

**Return Values**

When successful, the `msem_init` subroutine returns a pointer to the initialized `msemaphore` structure. Otherwise, it returns a null value and sets the `errno` global variable to indicate the error.

**Error Codes**

If the `msem_init` subroutine is unsuccessful, the `errno` global variable is set to one of the following values:

- **EINVAL**  
  Indicates the `InitialValue` parameter is not valid.

- **ENOMEM**  
  Indicates a new semaphore could not be created.

**Related Information**

The `mmap` subroutine, `msem_lock` subroutine, `msem_remove` subroutine, and `msem_unlock` subroutine.  

- List of Memory Mapping Services and Understanding Memory Mapping in AIX 5L Version 5.3 General Programming Concepts: Writing and Debugging Programs.

**msem_lock Subroutine**

**Purpose**

Locks a semaphore.
Library
Standard C Library (libc.a)

Syntax
#include <sys/mman.h>

int msem_lock (Sem, Condition)
msemaphore *Sem;
int Condition;

Description
The msem_lock subroutine attempts to lock a binary semaphore.

If the semaphore is not currently locked, it is locked and the msem_lock subroutine completes successfully.

If the semaphore is currently locked, and the value of the Condition parameter is MSEM_IF_NOWAIT, the msem_lock subroutine returns with an error. If the semaphore is currently locked, and the value of the Condition parameter is 0, the msem_lock subroutine does not return until either the calling process is able to successfully lock the semaphore or an error condition occurs.

All calls to the msem_lock and msem_unlock subroutines by multiple processes sharing a common ms semaphore structure behave as if the call were serialized.

If the ms semaphore structure contains any value not resulting from a call to the msem_init subroutine, followed by a (possibly empty) sequence of calls to the msem_lock and msem_unlock subroutines, the results are undefined. The address of an ms semaphore structure is significant. If the ms semaphore structure contains any value copied from an ms semaphore structure at a different address, the result is undefined.

Parameters
Sem Points to an ms semaphore structure that specifies the semaphore to be locked.
Condition Determines whether the msem_lock subroutine waits for a currently locked semaphore to unlock.

Return Values
When successful, the msem_lock subroutine returns a value of 0. Otherwise, it returns a value of -1 and sets the errno global variable to indicate the error.

Error Codes
If the msem_lock subroutine is unsuccessful, the errno global variable is set to one of the following values:

EAGAIN Indicates a value of MSEM_IF_NOWAIT is specified for the Condition parameter and the semaphore is already locked.
EINVAL Indicates the Sem parameter points to an ms semaphore structure specifying a semaphore that has been removed, or the Condition parameter is invalid.
EINTR Indicates the msem_lock subroutine was interrupted by a signal that was caught.
Related Information
The `msem_init` subroutine, `msem_remove` subroutine, `msem_unlock` subroutine.

List of Memory Mapping Services and Understanding Memory Mapping in AIX 5L Version 5.3 General Programming Concepts: Writing and Debugging Programs.

msem_remove Subroutine

Purpose
Removes a semaphore.

Library
Standard C Library (`libc.a`)

Syntax
`#include <sys/mman.h>`

```c
int msem_remove (Sem)
msemaphore *Sem;
```

Description
The `msem_remove` subroutine removes a binary semaphore. Any subsequent use of the `msemaphore` structure before it is again initialized by calling the `msem_init` subroutine will have undefined results.

The `msem_remove` subroutine also causes any process waiting in the `msem_lock` subroutine on the removed semaphore to return with an error.

If the `msemaphore` structure contains any value not resulting from a call to the `msem_init` subroutine, followed by a (possibly empty) sequence of calls to the `msem_lock` and `msem_unlock` subroutines, the result is undefined. The address of an `msemaphore` structure is significant. If the `msemaphore` structure contains any value copied from an `msemaphore` structure at a different address, the result is undefined.

Parameters

- **Sem**: Points to an `msemaphore` structure that specifies the semaphore to be removed.

Return Values
When successful, the `msem_remove` subroutine returns a value of 0. Otherwise, it returns a -1 and sets the `errno` global variable to indicate the error.

Error Codes
If the `msem_remove` subroutine is unsuccessful, the `errno` global variable is set to the following value:

- **EINVAL**: Indicates the `Sem` parameter points to an `msemaphore` structure that specifies a semaphore that has been removed.
Related Information
The msem_init subroutine, msem_lock subroutine, msem_unlock subroutine.

List of Memory Mapping Services and Understanding Memory Mapping in AIX 5L Version 5.3 General Programming Concepts: Writing and Debugging Programs.

msem_unlock Subroutine

Purpose
Unlocks a semaphore.

Library
Standard C Library (libc.a)

Syntax
#include <sys/mman.h>

int msem_unlock (Sem, Condition)
msemaphore *Sem;
int Condition;

Description
The msem_unlock subroutine attempts to unlock a binary semaphore.

If the semaphore is currently locked, it is unlocked and the msem_unlock subroutine completes successfully.

If the Condition parameter is 0, the semaphore is unlocked, regardless of whether or not any other processes are currently attempting to lock it. If the Condition parameter is set to the MSEM_IF_WAITERS value, and another process is waiting to lock the semaphore or it cannot be reliably determined whether some process is waiting to lock the semaphore, the semaphore is unlocked by the calling process. If the Condition parameter is set to the MSEM_IF_WAITERS value and no process is waiting to lock the semaphore, the semaphore will not be unlocked and an error will be returned.

Parameters
Sem Points to an msemaphore structure that specifies the semaphore to be unlocked.
Condition Determines whether the msem_unlock subroutine unlocks the semaphore if no other processes are waiting to lock it.

Return Values
When successful, the msem_unlock subroutine returns a value of 0. Otherwise, it returns a value of -1 and sets the errno global variable to indicate the error.

Error Codes
If the msem_unlock subroutine is unsuccessful, the errno global variable is set to one of the following values:

EAGAIN Indicates a Condition value of MSEM_IF_WAITERS was specified and there were no waiters.
EINVAL Indicates the Sem parameter points to an msemaphore structure specifying a semaphore that has been removed, or the Condition parameter is not valid.

Related Information
The msem_init subroutine, msem_lock subroutine, msem_remove subroutine.

List of Memory Mapping Services and Understanding Memory Mapping in AIX 5L Version 5.3 General Programming Concepts: Writing and Debugging Programs.

msgctl Subroutine

Purpose
Provides message control operations.

Library
Standard C Library (libc.a)

Syntax
#include <sys/msg.h>

int msgctl (MessageQueueID, Command, Buffer)

int MessageQueueID, Command;
struct msqid_ds * Buffer;

Description
The msgctl subroutine provides a variety of message control operations as specified by the Command parameter and stored in the structure pointed to by the Buffer parameter. The msqid_ds structure is defined in the sys/msg.h file.

The following limits apply to the message queue:
• Maximum message size is 65,535 bytes for releases prior to AIX 4.1.5 and is 4 Megabytes for release AIX 4.1.5 and later releases.
• Maximum number of messages per queue is 524288.
• Maximum number of message queue IDs is 4096 for releases before AIX 4.3.2 and 131072 for AIX 4.3.2 and following.
• Maximum number of bytes in a queue is 4 65,535 for releases prior to AIX 4.1.5 and is 4 Megabytes for release 4.1.5 and later releases.

Parameters
MessageQueueID Specifies the message queue identifier.
Command

The following values for the Command parameter are available:

**IPC_STAT**

Stores the current value of the above fields of the data structure associated with the MessageQueueID parameter into the msqid_ds structure pointed to by the Buffer parameter.

The current process must have read permission in order to perform this operation.

**IPC_SET**

Sets the value of the following fields of the data structure associated with the MessageQueueID parameter to the corresponding values found in the structure pointed to by the Buffer parameter:

- msg_perm.uid
- msg_perm.gid
- msg_perm.mode/*Only the low-order nine bits*/
- msg_qbytes

The effective user ID of the current process must have root user authority or must equal the value of the msg_perm.uid or msg_perm.cuid field in the data structure associated with the MessageQueueID parameter in order to perform this operation. To raise the value of the msg_qbytes field, the effective user ID of the current process must have root user authority.

**IPC_RMID**

Removes the message queue identifier specified by the MessageQueueID parameter from the system and destroys the message queue and data structure associated with it. The effective user ID of the current process must have root user authority or be equal to the value of the msg_perm.uid or msg_perm.cuid field in the data structure associated with the MessageQueueID parameter to perform this operation.

Buffer

Points to a msqid_ds structure.

Return Values

Upon successful completion, the msgctl subroutine returns a value of 0. Otherwise, a value of -1 is returned and the errno global variable is set to indicate the error.

Error Codes

The msgctl subroutine is unsuccessful if any of the following conditions is true:

**EINVAL**

The Command or MessageQueueID parameter is not valid.

**EACCES**

The Command parameter is equal to the IPC_STAT value, and the calling process was denied read permission.

**EPERM**

The Command parameter is equal to the IPC_RMID value and the effective user ID of the calling process does not have root user authority. Or, the Command parameter is equal to the IPC_SET value, and the effective user ID of the calling process is not equal to the value of the msg_perm.uid field or the msg_perm.cuid field in the data structure associated with the MessageQueueID parameter.

**EPERM**

The Command parameter is equal to the IPC_SET value, an attempt was made to increase the value of the msg_qbytes field, and the effective user ID of the calling process does not have root user authority.

**EFAULT**

The Buffer parameter points outside of the process address space.

Related Information

The msgget subroutine, msgrcv subroutine, msgrcv subroutine, msgsnd subroutine, msgxrcv subroutine.
msgget Subroutine

Purpose
Gets a message queue identifier.

Library
Standard C Library (libc.a)

Syntax
#include <sys/msg.h>

int msgget (Key, MessageFlag)
key_t Key;
int MessageFlag;

Description
The msgget subroutine returns the message queue identifier associated with the specified Key parameter.

A message queue identifier, associated message queue, and data structure are created for the value of the Key parameter if one of the following conditions is true:
- The Key parameter is equal to the IPC_PRIVATE value.
- The Key parameter does not already have a message queue identifier associated with it, and the IPC_CREAT value is set.

Upon creation, the data structure associated with the new message queue identifier is initialized as follows:
- The msg_perm.cuid, msg_perm.uid, msg_perm.cgid, and msg_perm.gid fields are set equal to the effective user ID and effective group ID, respectively, of the calling process.
- The low-order 9 bits of the msg_perm.mode field are set equal to the low-order 9 bits of the MessageFlag parameter.
- The msg_qnum, msg_lspid, msg_lrpid, msg_stime, and msg_rtime fields are set equal to 0.
- The msg_ctime field is set equal to the current time.
- The msg_qbytes field is set equal to the system limit.

The msgget subroutine performs the following actions:
- The msgget subroutine either finds or creates (depending on the value of the MessageFlag parameter) a queue with the Key parameter.
- The msgget subroutine returns the ID of the queue header to its caller.

Limits on message size and number of messages in the queue can be found in AIX 5L Version 5.3 General Programming Concepts: Writing and Debugging Programs.

Parameters
Key Specifies either the value IPC_PRIVATE or an Interprocess Communication (IPC) key constructed by the ftok ftok Subroutine on page 318 subroutine (or by a similar algorithm).
**MessageFlag**  
Constructed by logically ORing one or more of the following values:

- **IPC_CREAT**  
  Creates the data structure if it does not already exist.

- **IPC_EXCL**  
  Causes the msgget subroutine to fail if the IPC_CREAT value is also set and the data structure already exists.

- **S_IRUSR**  
  Permits the process that owns the data structure to read it.

- **S_IWUSR**  
  Permits the process that owns the data structure to modify it.

- **S_IRGRP**  
  Permits the group associated with the data structure to read it.

- **S_IWGRP**  
  Permits the group associated with the data structure to modify it.

- **S_IROTH**  
  Permits others to read the data structure.

- **S_IWOTH**  
  Permits others to modify the data structure.

Values that begin with **S_**I are defined in the **sys mode.h** file and are a subset of the access permissions that apply to files.

**Return Values**

Upon successful completion, the msgget subroutine returns a message queue identifier. Otherwise, a value of -1 is returned and the errno global variable is set to indicate the error.

**Error Codes**

The msgget subroutine is unsuccessful if any of the following conditions is true:

- **EACCES**  
  A message queue identifier exists for the Key parameter, but operation permission as specified by the low-order 9 bits of the MessageFlag parameter is not granted.

- **ENOENT**  
  A message queue identifier does not exist for the Key parameter and the IPC_CREAT value is not set.

- **ENOSPC**  
  A message queue identifier is to be created, but the system-imposed limit on the maximum number of allowed message queue identifiers system-wide would be exceeded.

- **EEXIST**  
  A message queue identifier exists for the Key parameter, and both IPC_CREAT and IPC_EXCL values are set.

**Related Information**

The ftok (**ftok Subroutine** on page 318) subroutine, msgctl (**msgctl Subroutine** on page 870) subroutine, msgrcv (**msgrcv Subroutine** on page 870) subroutine, msgsnd (**msgsnd Subroutine** on page 876) subroutine, msgxrcv (**msgxrcv Subroutine** on page 878) subroutine.

The **mode.h** file.

**msgrcv Subroutine**

**Purpose**

Reads a message from a queue.
Library
Standard C Library (libc.a)

Syntax
#include <sys/msg.h>

int msgrcv (MessageQueueID, MessagePointer, MessageSize, MessageType, MessageFlag)
int MessageQueueID, MessageFlag;
void *MessagePointer;
size_t MessageSize;
long int MessageType;

Description
The msgrcv subroutine reads a message from the queue specified by the MessageQueueID parameter and stores it into the structure pointed to by the MessagePointer parameter. The current process must have read permission in order to perform this operation.

Note: The routine may coredump instead of returning EFAULT when an invalid pointer is passed in case of 64-bit application calling 32-bit kernel interface.

Limits on message size and number of messages in the queue can be found in AIX 5L Version 5.3 General Programming Concepts: Writing and Debugging Programs.

Note: For a 64-bit process, the mtype field is 64 bits long. However, for compatibility with 32-bit processes, the mtype field must be a 32-bit signed value that is sign-extended to 64 bits. The most significant 32 bits are not put on the message queue. For a 64-bit process, the mtype field is again sign-extended to 64 bits.

Parameters
MessageQueueID Specifies the message queue identifier.
MessagePointer Points to a msgbuf structure containing the message. The msgbuf structure is defined in the sys/msg.h file and contains the following fields:

mtyp_t mtype; /* Message type */
char mtext[1]; /* Beginning of message text */

The mtype field contains the type of the received message as specified by the sending process. The mtext field is the text of the message.

MessageSize Specifies the size of the mtext field in bytes. The received message is truncated to the size specified by the MessageSize parameter if it is longer than the size specified by the MessageSize parameter and if the MSG_NOERROR value is set in the MessageFlag parameter. The truncated part of the message is lost and no indication of the truncation is given to the calling process.

MessageType Specifies the type of message requested as follows:
- If equal to the value of 0, the first message on the queue is received.
- If greater than 0, the first message of the type specified by the MessageType parameter is received.
- If less than 0, the first message of the lowest type that is less than or equal to the absolute value of the MessageType parameter is received.
MessageFlag

Specifies either a value of 0 or is constructed by logically ORing one or more of the following values:

**MSG_NOERROR**

Truncates the message if it is longer than the `MessageSize` parameter.

**IPC_NOWAIT**

Specifies the action to take if a message of the desired type is not on the queue:

- If the `IPC_NOWAIT` value is set, the calling process returns a value of -1 and sets the `errno` global variable to the `ENOMSG` error code.
- If the `IPC_NOWAIT` value is not set, the calling process suspends execution until one of the following occurs:
  - A message of the desired type is placed on the queue.
  - The message queue identifier specified by the `MessageQueueID` parameter is removed from the system. When this occurs, the `errno` global variable is set to the `EIDRM` error code, and a value of -1 is returned.
  - The calling process receives a signal that is to be caught. In this case, a message is not received and the calling process resumes in the manner described in the `sigaction` subroutine.

**Return Values**

Upon successful completion, the `msgrcv` subroutine returns a value equal to the number of bytes actually stored into the `mtext` field and the following actions are taken with respect to fields of the data structure associated with the `MessageQueueID` parameter:

- The `msg_qnum` field is decremented by 1.
- The `msg_lrpid` field is set equal to the process ID of the calling process.
- The `msg_rtime` field is set equal to the current time.

If the `msgrcv` subroutine is unsuccessful, a value of -1 is returned and the `errno` global variable is set to indicate the error.

**Error Codes**

The `msgrcv` subroutine is unsuccessful if any of the following conditions is true:

- **EINVAL**  The `MessageQueueID` parameter is not a valid message queue identifier.
- **EACCES**  The calling process is denied permission for the specified operation.
- **E2BIG**  The `mtext` field is greater than the `MessageSize` parameter, and the `MSG_NOERROR` value is not set.
- **ENOMSG**  The queue does not contain a message of the desired type and the `IPC_NOWAIT` value is set.
- **EFAULT**  The `MessagePointer` parameter points outside of the allocated address space of the process.
- **EINTR**  The `msgrcv` subroutine is interrupted by a signal.
- **EIDRM**  The message queue identifier specified by the `MessageQueueID` parameter has been removed from the system.

**Related Information**

The `msgct` subroutine, `msgget` subroutine, `msgsnd` subroutine, `msgxrcv` subroutine, `sigaction` subroutine.
msgsnd Subroutine

Purpose
Sends a message.

Library
Standard C Library (libc.a)

Syntax
#include <sys/msg.h>

int msgsnd (MessageQueueID, MessagePointer, MessageSize, MessageFlag)

int MessageQueueID, MessageFlag;
const void *MessagePointer;
size_t MessageSize;

Description
The msgsnd subroutine sends a message to the queue specified by the MessageQueueID parameter. The current process must have write permission to perform this operation. The MessagePointer parameter points to an msgbuf structure containing the message. The sys/msg.h file defines the msgbuf structure. The structure contains the following fields:

mtyp_t mtype; /* Message type */
char mtext[1]; /* Beginning of message text */

The mtype field specifies a positive integer used by the receiving process for message selection. The mtext field can be any text of the length in bytes specified by the MessageSize parameter. The MessageSize parameter can range from 0 to the maximum limit imposed by the system.

The following example shows a typical user-defined msgbuf structure that includes sufficient space for the largest message:

struct my_msgbuf
mtyp_t mtype;
char mtext[MSGSIZ]; /* MSGSIZ is the size of the largest message */

Note: The routine may coredump instead of returningEFAULT when an invalid pointer is passed in case of 64-bit application calling 32-bit kernel interface.

The following system limits apply to the message queue:
• Maximum message size is 65,535 bytes for releases prior to AIX 4.1.5 and is 4 Megabytes for AIX 4.1.5 and later releases.
• Maximum number of messages per queue is 524288.
• Maximum number of message queue IDs is 4096 for releases before AIX 4.3.2 and 131072 for AIX 4.3.2 and following.
• Maximum number of bytes in a queue is 4 65,535 bytes for releases prior to AIX 4.1.5 is 4 Megabytes for AIX 4.1.5 and later releases.

Note: For a 64-bit process, the mtype field is 64 bits long. However, for compatibility with 32-bit processes, the mtype field must be a 32-bit signed value that is sign-extended to 64 bits. The most significant 32 bits are not put on the message queue. For a 64-bit process, the mtype field is again sign-extended to 64 bits.
The `MessageFlag` parameter specifies the action to be taken if the message cannot be sent for one of the following reasons:

- The number of bytes already on the queue is equal to the number of bytes defined by the `msg_qbytes` structure.
- The total number of messages on the queue is equal to a system-imposed limit.

These actions are as follows:

- If the `MessageFlag` parameter is set to the `IPC_NOWAIT` value, the message is not sent, and the `msgsnd` subroutine returns a value of -1 and sets the `errno` global variable to the `EAGAIN` error code.
- If the `MessageFlag` parameter is set to 0, the calling process suspends execution until one of the following occurs:
  - The condition responsible for the suspension no longer exists, in which case the message is sent.
  - The `MessageQueueID` parameter is removed from the system. (For information on how to remove the `MessageQueueID` parameter, see the `msgctl` subroutine.) When this occurs, the `errno` global variable is set equal to the `EIDRM` error code, and a value of -1 is returned.
  - The calling process receives a signal that is to be caught. In this case the message is not sent and the calling process resumes execution in the manner prescribed in the `sigaction` subroutine.

### Parameters

- **MessageQueueID**: Specifies the queue to which the message is sent.
- **MessagePointer**: Points to a `msgbuf` structure containing the message.
- **MessageSize**: Specifies the length, in bytes, of the message text.
- **MessageFlag**: Specifies the action to be taken if the message cannot be sent.

### Return Values

Upon successful completion, a value of 0 is returned and the following actions are taken with respect to the data structure associated with the `MessageQueueID` parameter:

- The `msg_qnum` field is incremented by 1.
- The `msg_lspid` field is set equal to the process ID of the calling process.
- The `msg_stime` field is set equal to the current time.

If the `msgsnd` subroutine is unsuccessful, a value of -1 is returned and the `errno` global variable is set to indicate the error.

### Error Codes

The `msgsnd` subroutine is unsuccessful and no message is sent if one or more of the following conditions is true:

- **EACCES**: The calling process is denied permission for the specified operation.
- **EAGAIN**: The message cannot be sent for one of the reasons stated previously, and the `MessageFlag` parameter is set to the `IPC_NOWAIT` value or the system has temporarily run out of memory resource.
- **EFAULT**: The `MessagePointer` parameter points outside of the address space of the process.
- **EIDRM**: The message queue identifier specified by the `MessageQueueID` parameter has been removed from the system.
- **EINVAL**: The `msgsnd` subroutine received a signal.
- **EINVAL**: The `MessageQueueID` parameter is not a valid message queue identifier.
- **EINVAL**: The `MessageSize` parameter is less than 0 or greater than the system-imposed limit.
- **EINVAL**: The upper 32-bits of the 64-bit `mtype` field for a 64-bit process is not 0.
- **ENOMEM**: The message could not be sent because not enough storage space was available.
Related Information
The msgctl subroutine, msgget subroutine, msgrcv subroutine, sigaction subroutine.

msgxrcv Subroutine

Purpose
Receives an extended message.

Library
Standard C Library (libc.a)

Syntax
For releases prior to AIX 4.3:

```c
#include <sys/msg.h>

int msgxrcv (MessageQueueID, MessagePointer, MessageSize, MessageType, MessageFlag)
```

For AIX 4.3 and later releases:

```c
#include <sys/msg.h>

int msgxrcv (MessageQueueID, MessagePointer, MessageSize, MessageType, MessageFlag)
```

Description
The msgxrcv subroutine reads a message from the queue specified by the MessageQueueID parameter and stores it into the extended message receive buffer pointed to by the MessagePointer parameter. The current process must have read permission in order to perform this operation. The msgxbuf structure is defined in the sys/msg.h file.

Note: The routine may coredump instead of returning EFAULT when an invalid pointer is passed in case of 64-bit application calling 32-bit kernel interface.

The following limits apply to the message queue:

- Maximum message size is 65,535 bytes for releases prior to AIX 4.1.5 and is 4 Megabytes for AIX 4.1.5 and later releases.
- Maximum number of messages per queue is 8192.
- Maximum number of message queue IDs is 4096 for releases before AIX 4.3.2 and 131072 for AIX 4.3.2 and following.
- Maximum number of bytes in a queue is 4 65,535 for releases prior to AIX 4.1.5 and is 4 Megabytes for AIX 4.1.5 later releases.
Note: For a 64-bit process, the mtype field is 64 bits long. However, for compatibility with 32-bit processes, the mtype field must be a 32-bit signed value that is sign-extended to 64 bits. The most significant 32 bits are not put on the message queue. For a 64-bit process, the mtype field is again sign-extended to 64 bits.

Parameters

MessageQueueID
Specifies the message queue identifier.

MessagePointer
Specifies a pointer to an extended message receive buffer where a message is stored.

MessageSize
Specifies the size of the mtext field in bytes. The receive message is truncated to the size specified by the MessageSize parameter if it is larger than the MessageSize parameter and the MSG_NOERROR value is true. The truncated part of the message is lost and no indication of the truncation is given to the calling process. If the message is longer than the number of bytes specified by the MessageSize parameter and the MSG_NOERROR value is not set, the msgxrcv subroutine is unsuccessful and sets the errno global variable to the E2BIG error code.

MessageType
Specifies the type of message requested as follows:
- If the MessageType parameter is equal to 0, the first message on the queue is received.
- If the MessageType parameter is greater than 0, the first message of the type specified by the MessageType parameter is received.
- If the MessageType parameter is less than 0, the first message of the lowest type that is less than or equal to the absolute value of the MessageType parameter is received.

MessageFlag
Specifies a value of 0 or a value constructed by logically ORing one or more of the following values:
- MSG_NOERROR Truncates the message if it is longer than the number of bytes specified by the MessageSize parameter.
- IPC_NOWAIT Specifies the action to take if a message of the desired type is not on the queue:
  - If the IPC_NOWAIT value is set, the calling process returns a value of -1 and sets the errno global variable to the ENOMSG error code.
  - If the IPC_NOWAIT value is not set, the calling process suspends execution until one of the following occurs:
    - A message of the desired type is placed on the queue.
    - The message queue identifier specified by the MessageQueueID parameter is removed from the system. When this occurs, the errno global variable is set to the EIDRM error code, and a value of -1 is returned.
    - The calling process receives a signal that is to be caught. In this case, a message is not received and the calling process resumes in the manner prescribed in the sigaction subroutine.

Return Values

Upon successful completion, the msgxrcv subroutine returns a value equal to the number of bytes actually stored into the mtext field, and the following actions are taken with respect to the data structure associated with the MessageQueueID parameter:
- The msg_qnum field is decremented by 1.
- The msg_lrpيد field is set equal to the process ID of the calling process.
- The msg_rtime field is set equal to the current time.

If the msgxrcv subroutine is unsuccessful, a value of -1 is returned and the errno global variable is set to indicate the error.
Error Codes
The msgxrcv subroutine is unsuccessful if any of the following conditions is true:

- **EINVAL** The MessageQueueID parameter is not a valid message queue identifier.
- **EACCES** The calling process is denied permission for the specified operation.
- **EINVAL** The MessageSize parameter is less than 0.
- **E2BIG** The mtext field is greater than the MessageSize parameter, and the MSG_NOERROR value is not set.
- **ENOMEM** The queue does not contain a message of the desired type and the IPC_NOWAIT value is set.
- **EFAULT** The MessagePointer parameter points outside of the process address space.
- **EINTR** The msgxrcv subroutine was interrupted by a signal.
- **EIDRM** The message queue identifier specified by the MessageQueueID parameter is removed from the system.

Related Information
The msgctl (*msgctl Subroutine* on page 870) subroutine, msgget (*msgget Subroutine* on page 872) subroutine, msgrcv (*msgrcv Subroutine* on page 873) subroutine, msgsnd (*msgsnd Subroutine* on page 876) subroutine, sigaction subroutine.

**msleep Subroutine**

**Purpose**
Put a process to sleep when a semaphore is busy.

**Library**
Standard C Library (libc.a)

**Syntax**
```
#include <sys/mman.h>

int msleep (Sem)
    msemaphore *Sem;
```

**Description**
The msleep subroutine puts a calling process to sleep when a semaphore is busy. The semaphore should be located in a shared memory region. Use the mmap subroutine to create the shared memory section.

All of the values in the msemaphore structure must result from a msem_init subroutine call. This call may or may not be followed by a sequence of calls to the msem_lock subroutine or the msem_unlock subroutine. If the msemaphore structure value originates in another manner, the results of the msleep subroutine are undefined.

The address of the msemaphore structure is significant. You should be careful not to modify the structure's address. If the structure contains values copied from a msemaphore structure at another address, the results of the msleep subroutine are undefined.

**Parameters**

- **Sem** Points to the msemaphore structure that specifies the semaphore.
Error Codes
If the *msleep* subroutine is unsuccessful, the *errno* global variable is set to one of the following values:

EFAULT Indicates that the *Sem* parameter points to an invalid address or the address does not contain a valid *msemaphore* structure.
EINTR Indicates that the process calling the *msleep* subroutine was interrupted by a signal while sleeping.

Related Information

Related Information in *AIX 5L Version 5.3 General Programming Concepts: Writing and Debugging Programs*.

**msync Subroutine**

**Purpose**
Synchronize memory with physical storage.

**Library**
Standard C Library (*libc.a*).

**Syntax**
```c
#include <sys/types.h>
#include <sys/mman.h>

int msync ( void *addr, size_t len, int flags);
```

**Description**
The *msync* subroutine controls the caching operations of a mapped file or shared memory region. Use the *msync* subroutine to transfer modified pages in the region to the underlying file storage device.

If the application has requested Single UNIX Specification, Version 2 compliant behavior then the *st_ctime* and *st_mtime* fields of the mapped file are marked for update upon successful completion of the *msync* subroutine call if the file has been modified.

**Parameters**

*addr* Specifies the address of the region to be synchronized. Must be a multiple of the page size returned by the *sysconf* subroutine using the _SC_PAGE_SIZE value for the *Name* parameter.

*len* Specifies the length, in bytes, of the region to be synchronized. If the *len* parameter is not a multiple of the page size returned by the *sysconf* subroutine using the _SC_PAGE_SIZE value for the *Name* parameter, the length of the region is rounded up to the next multiple of the page size.
flags Specifies one or more of the following symbolic constants that determine the way caching operations are performed:

**MS_SYNC**
Specifies synchronous cache flush. The `msync` subroutine does not return until the system completes all I/O operations.

This flag is invalid when the `MAP_PRIVATE` flag is used with the `mmap` subroutine. `MAP_PRIVATE` is the default privacy setting. When the **MS_SYNC** and **MAP_PRIVATE** flags both are used, the `msync` subroutine returns an `errno` value of **EINVAL**.

**MS_ASYNC**
Specifies an asynchronous cache flush. The `msync` subroutine returns after the system schedules all I/O operations.

This flag is invalid when the **MAP_PRIVATE** flag is used with the `mmap` subroutine. `MAP_PRIVATE` is the default privacy setting. When the **MS_SYNC** and **MAP_PRIVATE** flags both are used, the `msync` subroutine returns an `errno` value of **EINVAL**.

**MS_INVALIDATE**
Specifies that the `msync` subroutine invalidates all cached copies of the pages. New copies of the pages must then be obtained from the file system the next time they are referenced.

**Return Values**
When successful, the `msync` subroutine returns 0. Otherwise, it returns -1 and sets the `errno` global variable to indicate the error.

**Error Codes**
If the `msync` subroutine is unsuccessful, the `errno` global variable is set to one of the following values:

- **EIO** An I/O error occurred while reading from or writing to the file system.
- **ENOMEM** The range specified by (`addr`, `addr` + `len`) is invalid for a process' address space, or the range specifies one or more unmapped pages.
- **EINVAL** The `addr` argument is not a multiple of the page size as returned by the `sysconf` subroutine using the `_SC_PAGE_SIZE` value for the `Name` parameter, or the `flags` parameter is invalid. The address of the region is within the process' inheritable address space.

**mt__trce Subroutine**

**Purpose**
Dumps traceback information into a lightweight core file.

**Library**
PTools Library (`libptools_ptr.a`)

**Syntax**

```
void mt__trce (int FileDescriptor, int Signal, struct sigcontext *Context, int Node);
```

**Description**
The `mt__trce` subroutine dumps traceback information of the calling thread and all other threads allocated in the process space into the file specified by the `FileDescriptor` parameter. The format of the output from this subroutine complies with the Parallel Tools Consortium Lightweight CoreFile Format. Threads, except the calling thread, will be suspended after the calling thread enters this subroutine and while the traceback information is being obtained. Threads execution resumes when this subroutine returns.
When using the `mt__trce` subroutine in a signal handler, it is recommended that the application be started with the environment variable AIXTHREAD_SCOPE set to S (As in `export AIXTHREAD_SCOPE=S`). If this variable is not set, the application may hang.

**Parameters**

**Context**
Points to the `sigcontext` structure containing the context of the thread when the signal happens. The context is used to generate the traceback information for the calling thread. This is used only if the `Signal` parameter is nonzero. If the `mt__trce` subroutine is called with the `Signal` parameter set to zero, the `Context` parameter is ignored and the traceback information is generated based on the current context of the calling thread. Refer to the `sigaction` subroutine for further description about signal handlers and how the `sigcontext` structure is passed to a signal handler.

**File Descriptor**
The file descriptor of the lightweight core file. It specifies the target file into which the traceback information is written.

**Node**
Specifies the number of the tasks or nodes where this subroutine is executing and is used only for a parallel application consisting of multiple tasks. The `Node` parameter will be used in section headers of the traceback information to identify the task or node from which the information is generated.

**Signal**
The number of the signal that causes the signal handler to be executed. This is used only if the `mt__trce` subroutine is called from a signal handler. A Fault-Info section defined by the Parallel Tools Consortium Lightweight Core File Format will be written into the output lightweight core file based on this signal number. If the `mt__trce` subroutine is not called from a signal handler, the `Signal` parameter must be set to 0 and a Fault-Info section will not be generated.

**Notes:**
1. To obtain source line information in the traceback, the programs must have been compiled with the `-g` option to include the necessary line number information in the executable files. Otherwise, address offset from the beginning of the function is provided.
2. Line number information is not provided for shared objects even if they were compiled with the `-g` option.
3. Function names are not provided if a program or a library is compiled with optimization. To obtain function name information in the traceback and still have the object code optimized, compiler option `-qtbtable=full` must be specified.
4. In rare cases, the traceback of a thread may seem to skip one level of procedure calls. This is because the traceback is obtained at the moment the thread entered a procedure and has not yet allocated a stack frame.

**Return Values**
Upon successful completion, the `mt__trce` subroutine returns a value of 0. Otherwise, an error number is returned to indicate the error.

**Error Codes**
If an error occurs, the subroutine returns -1 and the `errno` global variable is set to indicate the error, as follows:

- **EBADF**
  The `FileDescriptor` parameter does not specify a valid file descriptor open for writing.
- **ENOSPC**
  No free space is left in the file system containing the file.
- **EDQUOT**
  New disk blocks cannot be allocated for the file because the user or group quota of blocks has been exhausted on the file system.
- **EINVAL**
  The value of the `Signal` parameter is invalid or the `Context` parameter points to an invalid context.
- **ENOMEM**
  Insufficient memory exists to perform the operation.
Examples

1. The following example calls the `mt__trce` subroutine to generate traceback information in a signal handler.

```c
void
my_handler(int signal,
    int code,
    struct sigcontext *sigcontext_data)
{
    int lcf_fd;
    ....
    lcf_fd = open(file_name, O_WRONLY|O_CREAT|O_APPEND, 0666);
    ....
    rc = mt__trce(lcf_fd, signal, sigcontext_data, 0);
    ....
    close(lcf_fd);
    ....
}
```

2. The following is an example of the lightweight core file generated by the `mt__trce` subroutine. Notice the thread ID in the information is the unique sequence number of a thread for the life time of the process containing the thread.

```plaintext
+++PARALLEL TOOLS CONSORTIUM LIGHTWEIGHT COREFILE FORMAT version 1.0
+++LCB 1.0 Thu Jun 30 16:02:35 1999 Generated by AIX
#
+++ID Node 0 Process 21084 Thread 1
+++FAULT "SIGABRT - Abort"
+++STACK
func2 : 123 # in file
func1 : 272 # in file
main : 49 # in file
---STACK
---ID Node 0 Process 21084 Thread 1
#
+++ID Node 0 Process 21084 Thread 2
+++STACK
nsleep : 0x00000001c
sleep : 0x00000030
_f_mt_exec : 21 # in file
_pthread_body : 0x00000114
---STACK
---ID Node 0 Process 21084 Thread 2
#
+++ID Node 0 Process 21084 Thread 3
+++STACK
nsleep : 0x00000001c
sleep : 0x00000030
_f_mt_exec : 21 # in file
_pthread_body : 0x00000114
---STACK
---ID Node 0 Process 21084 Thread 3
---LCB
```

Related Information
The `install_lwcf_handler` and `sigaction` subroutines.

munmap Subroutine

Purpose
Unmaps pages of memory.
Library
Standard C Library (libc.a)

Syntax
#include <sys/types.h>
#include <sys/mman.h>

int munmap (addr, len);
void *addr;
size_t len;

Description
The munmap subroutine unmaps a mapped file or shared memory region or anonymous memory region. The munmap subroutine unmaps regions created from calls to the mmap subroutine only.

If an address lies in a region that is unmapped by the munmap subroutine and that region is not subsequently mapped again, any reference to that address will result in the delivery of a SIGSEGV signal to the process.

Parameters
addr Specifies the address of the region to be unmapped. Must be a multiple of the page size returned by the sysconf subroutine using the _SC_PAGE_SIZE value for the Name parameter.
len Specifies the length, in bytes, of the region to be unmapped. If the len parameter is not a multiple of the page size returned by the sysconf subroutine using the _SC_PAGE_SIZE value for the Name parameter, the length of the region is rounded up to the next multiple of the page size.

Return Values
When successful, the munmap subroutine returns 0. Otherwise, it returns -1 and sets the errno global variable to indicate the error.

Error Codes
If the munmap subroutine is unsuccessful, the errno global variable is set to the following value:

EINVAL The addr parameter is not a multiple of the page size as returned by the sysconf subroutine using the _SC_PAGE_SIZE value for the Name parameter.
EINVAL The application has requested Single UNIX Specification, Version 2 compliant behavior and the len argument is 0.

mwakeup Subroutine

Purpose
Wakes up a process that is waiting on a semaphore.

Library
Standard C Library (libc.a)
#include <sys/mman.h>
int mwakeup (Sem)
msemaphore *Sem;

Description
The mwakeup subroutine wakes up a process that is sleeping and waiting for an idle semaphore. The semaphore should be located in a shared memory region. Use the mmap subroutine to create the shared memory section.

All of the values in the msemaphore structure must result from a msem_init subroutine call. This call may or may not be followed by a sequence of calls to the msem_lock subroutine or the msem_unlock subroutine. If the msemaphore structure value originates in another manner, the results of the mwakeup subroutine are undefined.

The address of the msemaphore structure is significant. You should be careful not to modify the structure’s address. If the structure contains values copied from a msemaphore structure at another address, the results of the mwakeup subroutine are undefined.

Parameters
Sem Points to the msemaphore structure that specifies the semaphore.

Return Values
When successful, the mwakeup subroutine returns a value of 0. Otherwise, this routine returns a value of -1 and sets the errno global variable to EFAULT.

Error Codes
A value of EFAULT indicates that the Sem parameter points to an invalid address or that the address does not contain a valid msemaphore structure.

Related Information
The mmap ("mmap or mmap64 Subroutine" on page 834) subroutine, msem_init ("msem_init Subroutine" on page 865) subroutine, msem_lock ("msem_lock Subroutine" on page 866) subroutine, msem_unlock ("msem_unlock Subroutine" on page 869) subroutine, and the msleep ("msleep Subroutine" on page 880) subroutine.

Understanding Memory Mapping in AIX 5L Version 5.3 General Programming Concepts: Writing and Debugging Programs.

nan, nanf, or nanl Subroutine

Purpose
Returns quiet NaN.

Syntax
#include <math.h>

double nan (tagp)
const char *tagp;

float nanf (tagp)

const char *tagp;

long double nanl (tagp)
const char *tagp;

Description
The function call nan("n-char-sequence") is equivalent to:
strtod("NAN(n-char-sequence)", (char **) NULL);

The function call nan(" ") is equivalent to:
strtod("NAN()", (char **) NULL)

If tagp does not point to an n-char sequence or an empty string, the function call is equivalent to:
strtod("NAN", (char **) NULL)

Function calls to nanf and nanl are equivalent to the corresponding function calls to strtod and strtod.

Parameters
tagp Indicates the content of the quiet NaN.

Return Values
The nan, nanf, and nanl subroutines return a quiet NaN with content indicated through tagp.

Related Information
The "atof atoff Subroutine" on page 96.

math.h in AIX 5L Version 5.3 Files Reference.

nanosleep Subroutine

Purpose
Causes the current thread to be suspended from execution.

Library
Standard C Library (libc.a)

Syntax
#include <time.h>

int nanosleep (rqtp, rmtp)
const struct timespec *rqtp;
struct timespec *rmtp;

Description
The nanosleep subroutine causes the current thread to be suspended from execution until either the time
interval specified by the rqtp parameter has elapsed or a signal is delivered to the calling thread and its
action is to invoke a signal-catching function or to terminate the process. The suspension time may be
longer than requested because the argument value is rounded up to an integer multiple of the sleep
resolution. This can also occur because of the scheduling of other activity by the system. Unless it is
interrupted by a signal, the suspension time will not be less than the time specified by the rqtp parameter,
as measured by the system clock CLOCK_REALTIME.
The use of the **nanosleep** subroutine has no effect on the action or blockage of any signal.

**Parameters**

\[ \text{rqtp} \quad \text{Specifies the time interval that the thread is suspended.} \]

\[ \text{rmtp} \quad \text{Points to the } \text{timespec} \text{ structure.} \]

**Return Values**

If the **nanosleep** subroutine returns because the requested time has elapsed, its return value is zero.

If the **nanosleep** subroutine returns because it has been interrupted by a signal, it returns -1 and sets **errno** to indicate the interruption. If the *rmtp* parameter is non-NULL, the **timespec** structure is updated to contain the amount of time remaining in the interval (the requested time minus the time actually slept). If the *rmtp* parameter is NULL, the remaining time is not returned.

If the **nanosleep** subroutine fails, it returns -1 and sets **errno** to indicate the error.

**Error Codes**

The **nanosleep** subroutine fails if:

- **EINTR** The **nanosleep** subroutine was interrupted by a signal.
- **EINVAL** The *rqtp* parameter specified a nanosecond value less than zero or greater than or equal to 1000 million.

**Related Information**

The **sleep** subroutine in *AIX 5L Version 5.3 Technical Reference: Base Operating System and Extensions* Volume 2.

### nearbyint, nearbyintf, or nearbyintl Subroutine

**Purpose**

Rounds numbers to an integer value in floating-point format.

**Syntax**

```c
#include <math.h>

double nearbyint (x);

double x;

float nearbyintf (x);

float x;

long double nearbyintl (x);

long double x;
```

**Description**

The **nearbyint**, **nearbyintf**, and **nearbyintl** subroutines round the *x* parameter to an integer value in floating-point format, using the current rounding direction and without raising the inexact floating-point exception.
An application wishing to check for error situations should set the `errno` global variable to zero and call `feclearexcept(FE_ALL_EXCEPT)` before calling these subroutines. Upon return, if `errno` is nonzero or `fetestexcept(FE_INVALID | FE_DIVBYZERO | FE_OVERFLOW | FE_UNDERFLOW)` is nonzero, an error has occurred.

**Parameters**

`x` Specifies the value to be computed.

**Return Values**

Upon successful completion, the `nearbyint`, `nearbyintf`, and `nearbyintl` subroutines return the rounded integer value.

If `x` is NaN, a NaN is returned.

If `x` is ±0, ±0 is returned.

If `x` is ±Inf, `x` is returned.

If the correct value would cause overflow, a range error occurs and the `nearbyint`, `nearbyintf`, and `nearbyintl` subroutines return the value of the macro ±`HUGE_VAL`, ±`HUGE_VALF`, and ±`HUGE_VALL` (with the same sign as `x`), respectively.

**Related Information**

“feclearexcept Subroutine” on page 262 and “fetestexcept Subroutine” on page 270.

`math.h` in AIX 5L Version 5.3 Files Reference.

**nextafter, nextafterf, nextafterl, nexttoward, nexttowardf, or nexttowardl Subroutine**

**Purpose**

Computes the next representable floating-point number.

**Syntax**

```c
#include <math.h>

float nextafterf (x, y)
float x;
float y;

long double nextafterl (x, y)
long double x;
long double y;

double nextafter (x, y)
double x, y;

double nexttoward (x, y)
double x;
long double y;

float nexttowardf (x, y)
float x;
long double y;
```
long double nexttowardl (x, y)
long double x;
long double y;

Description
The `nextafter`, `nextafterl`, and `nextafter` subroutines compute the next representable floating-point value following x in the direction of y. Thus, if y is less than x, the `nextafter` subroutine returns the largest representable floating-point number less than x.

The `nextafter`, `nextafterf`, and `nextafterl` subroutines return y if x equals y.

The `nexttoward`, `nexttowardf`, and `nexttowardl` subroutines are equivalent to the corresponding `nextafter` subroutine, except that the second parameter has type `long double` and the subroutines return y converted to the type of the subroutine if x equals y.

An application wishing to check for error situations should set the `errno` global variable to zero and call `feclearexcept(FE_ALL_EXCEPT)` before calling these subroutines. Upon return, if `errno` is nonzero or `fetestexcept(FE_INVALID | FE_DIVBYZERO | FE_OVERFLOW | FE_UNDERFLOW)` is nonzero, an error has occurred.

Parameters
x Specifies the starting value. The next representable floating-point number is found from x in the direction specified by y.
y Specifies the direction.

Return Values
Upon successful completion, the `nextafter`, `nextafterl`, `nextafter`, `nexttoward`, `nexttowardf`, and `nexttowardl` subroutines return the next representable floating-point value following x in the direction of y.

If x==y, y (of the type x) is returned.

If x is finite and the correct function value would overflow, a range error occurs and ±HUGE_VAL, ±HUGE_VALF, and ±HUGE_VALL (with the same sign as x) is returned as appropriate for the return type of the function.

If x or y is NaN, a NaN is returned.

If x!=y and the correct subroutine value is subnormal, zero, or underflows, a range error occurs, and either the correct function value (if representable) or 0.0 is returned.

Error Codes
For the `nextafter` subroutine, if the x parameter is finite and the correct function value would overflow, HUGE_VAL is returned and `errno` is set to ERANGE.

Related Information
"feclearexcept Subroutine" on page 262 and "fetestexcept Subroutine" on page 270.

`math.h` in AIX 5L Version 5.3 Files Reference.
newpass Subroutine

Purpose
Generates a new password for a user.

Library
Security Library (libc.a)

Syntax
#include <usersec.h>
#include <userpw.h>

char *newpass(Password);
struct userpw *Password;

Description

Note: This subroutine has been depreciated and its use is not recommended. The `chpass Subroutine` on page 154 should be used in its place.

The newpass subroutine generates a new password for the user specified by the Password parameter. This subroutine displays a dialogue to enter and confirm the user's new password.

Passwords can contain almost any legal value for a character but cannot contain (National Language Support (NLS) code points. Passwords cannot have more than the value specified by MAX_PASS.

If a password is successfully generated, a pointer to a buffer containing the new password is returned and the last update time is reset.

Note: The newpass subroutine is not safe in a multithreaded environment. To use newpass in a threaded application, the application must keep the integrity of each thread.
Parameters

Password  Specifies a user password structure. This structure is defined in the userpw.h file and contains the following members:

  upw_name
    A pointer to a character buffer containing the user name.

  upw_passwd
    A pointer to a character buffer containing the current password.

  upw_lastupdate
    The time the password was last changed, in seconds since the epoch.

  upw_flags
    A bit mask containing 0 or more of the following values:

      PW_ADMIN
        This bit indicates that password information for this user may only be changed by the root user.

      PW_ADMCHG
        This bit indicates that the password is being changed by root and the password will have to be changed upon the next successful running of the login or su commands to this account.

Security

Policy: Authentication  To change a password, the invoker must be properly authenticated.

Note: Programs that invoke the newpass subroutine should be written to conform to the authentication rules enforced by newpass. The PW_ADMCHG flag should always be explicitly cleared unless the invoker of the command is an administrator.

Return Values

If a new password is successfully generated, a pointer to the new encrypted password is returned. If an error occurs, a null pointer is returned and the errno global variable is set to indicate the error.

Error Codes

The newpass subroutine fails if one or more of the following are true:

EINVAL  The structure passed to the newpass subroutine is invalid.

ESAD  Security authentication is denied for the invoker.

EPERM  The user is unable to change the password of a user with the PW_ADMCHG bit set, and the real user ID of the process is not the root user.

ENOENT  The user is not properly defined in the database.

Implementation Specifics

This subroutine is part of Base Operating System (BOS) Runtime.

Related Information

The chpass Subroutine on page 154, getpass Subroutine on page 397 subroutine, getuserpw Subroutine on page 463 subroutine.

The pwdadm command.
newpassx Subroutine

Purpose
Generates a new password for a user (without a name length limit).

Library
Security Library (libc.a)

Syntax
#include <usersec.h>
#include <userpw.h>

char *newpassx (Password)
struct userpw *Password;

Description

Note: The newpassx subroutine has been obsoleted by the more current chpassx subroutine. Use the chpassx subroutine instead.

The newpassx subroutine generates a new password for the user specified by the Password parameter. The new password is then checked to ensure that it meets the password rules on the system unless the user is exempted from these restrictions. Users must have root user authority to invoke this subroutine. The password rules are defined in the /etc/security/user file or the administrative domain for the user and are described in both the user file and the passwd command.

Passwords can contain almost any legal value for a character but cannot contain National Language Support (NLS) code points. Passwords cannot have more characters than the value specified by PASS_MAX.

The newpassx subroutine authenticates the user prior to returning the new password. If the PW_ADMCHG flag is set in the upw_flags member of the Password parameter, the supplied password is checked against the calling user’s password. This is done to authenticate the user corresponding to the real user ID of the process instead of the user specified by the upw_name member of the Password parameter structure.

If a password is successfully generated, a pointer to a buffer containing the new password is returned and the last update time is set to the current system time. The password value in the /etc/security/passwd file or user’s administrative domain is not modified.

Note: The newpassx subroutine is not safe in a multithreaded environment. To use newpassx in a threaded application, the application must keep the integrity of each thread.

Parameters

Password Specifies a user password structure.

The fields in a userpw structure are defined in the userpw.h file, and they include the following members:

upw_name Specifies the user’s name.
upw_passwd Specifies the user’s encrypted password.
upw_lastupdate

Specifies the time, in seconds, since the epoch (that is, 00:00:00 GMT, 1 January 1970), when the password was last updated.

upw_flags

Specifies attributes of the password. This member is a bit mask of one or more of the following values, defined in the userpw.h file:

- **PW_NOCHECK**
  Specifies that new passwords need not meet password restrictions in effect for the system.

- **PW_ADMCHG**
  Specifies that the password was last set by an administrator and must be changed at the next successful use of the login or su command.

- **PW_ADMIN**
  Specifies that password information for this user can only be changed by the root user.

upw_authdb

Specifies the administrative domain containing the authentication data.

Security

Policy: Authentication
To change a password, the invoker must be properly authenticated.

Note: Programs that invoke the newpassx subroutine should be written to conform to the authentication rules enforced by newpassx. The PW_ADMCHG flag should always be explicitly cleared unless the invoker of the command is an administrator.

Return Values

If a new password is successfully generated, a pointer to the new encrypted password is returned. If an error occurs, a null pointer is returned and the errno global variable is set to indicate the error.

Error Codes

The newpassx subroutine fails if one or more of the following is true:

- **EINVAL**
  The structure passed to the newpassx subroutine is invalid.

- **ENOENT**
  The user is not properly defined in the database.

- **EPERM**
  The user is unable to change the password of a user with the PW_ADMCHG bit set, and the real user ID of the process is not the root user.

- **ESAD**
  Security authentication is denied for the invoker.

Related Information

The ["getpass Subroutine" on page 397](#), ["getuserpwx Subroutine" on page 465](#).

The [login Command](#), [passwd Command](#), [pwdadm Command](#)

["List of Security and Auditing Subroutines"](#), [Subroutines Overview](#) in AIX 5L Version 5.3 General Programming Concepts: Writing and Debugging Programs.

nftw or nftw64 Subroutine

Purpose

Walks a file tree.
Library
Standard C Library (libc.a)

Syntax
#include <ftw.h>

int nftw (Path, Function, Depth, Flags)
const char *Path;
int *(Function) ( );
int Depth;
int Flags;

int nftw64(Path, Function,Depth)
const char *Path;
int *(Function) ( );
int Depth;
int Flags;

Description
The nftw and nftw64 subroutines recursively descend the directory hierarchy rooted in the Path parameter. The nftw and nftw64 subroutines have a similar effect to ftw and ftw64 except that they take an additional argument flags, which is a bitwise inclusive-OR of zero or more of the following flags:

- **FTW_CHDIR**  If set, the current working directory will change to each directory as files are reported. If clear, the current working directory will not change.
- **FTW_DEPTH**  If set, all files in a directory will be reported before the directory itself. If clear, the directory will be reported before any files.
- **FTW_MOUNT**  If set, symbolic links will not be followed. If clear the links will be followed.
- **FTW_PHYS**  If set, symbolic links will not be followed. If clear the links will be followed, and will not report the same file more than once.

For each file in the hierarchy, the nftw and nftw64 subroutines call the function specified by the Function parameter. The nftw subroutine passes a pointer to a null-terminated character string containing the name of the file, a pointer to a stat structure containing information about the file, an integer and a pointer to an FTW structure. The nftw64 subroutine passes a pointer to a null-terminated character string containing the name of the file, a pointer to a stat64 structure containing information about the file, an integer and a pointer to an FTW structure.

The nftw subroutine uses the stat system call which will fail on files of size larger than 2 Gigabytes. The nftw64 subroutine must be used if there is a possibility of files of size larger than 2 Gigabytes.

The integer passed to the Function parameter identifies the file type with one of the following values:

- **FTW_F**  Regular file
- **FTW_D**  Directory
- **FTW_DNR**  Directory that cannot be read
- **FTW_DP**  The Object is a directory and subdirectories have been visited. (This condition will only occur if FTW_DEPTH is included in flags).
- **FTW_SL**  Symbolic Link
- **FTW_SLN**  Symbolic Link that does not name an existin file. (This condition will only occur if the FTW_PHYS flag is not included in flags).
- **FTW_NS**  File for which the stat structure could not be executed successfully

If the integer is **FTW_DNR**, the files and subdirectories contained in that directory are not processed.
If the integer is **FTW_NS**, the `stat` structure contents are meaningless. An example of a file that causes **FTW_NS** to be passed to the `Function` parameter is a file in a directory for which you have read permission but not execute (search) permission.

The **FTW** structure pointer passed to the `Function` parameter contains `base` which is the offset of the object’s filename in the pathname passed as the first argument to `Function`. The value of `level` indicates depth relative to the root of the walk.

The **nftw** and **nftw64** subroutines use one file descriptor for each level in the tree. The `Depth` parameter specifies the maximum number of file descriptors to be used. In general, the **nftw** and **nftw64** run faster of the value of the `Depth` parameter is at least as large as the number of levels in the tree. However, the value of the `Depth` parameter must not be greater than the number of file descriptors currently available for use. If the value of the `Depth` parameter is 0 or a negative number, the effect is the same as if it were 1.

Because the **nftw** and **nftw64** subroutines are recursive, it is possible for it to terminate with a memory fault due to stack overflow when applied to very deep file structures.

The **nftw** and **nftw64** subroutines use the `malloc` subroutine to allocate dynamic storage during its operation. If the **nftw** subroutine is terminated prior to its completion, such as by the `longjmp` subroutine being executed by the function specified by the `Function` parameter or by an interrupt routine, the **nftw** subroutine cannot free that storage. The storage remains allocated. A safe way to handle interrupts is to store the fact that an interrupt has occurred, and arrange to have the function specified by the `Function` parameter return a nonzero value the next time it is called.

### Parameters

- **Path**: Specifies the directory hierarchy to be searched.
- **Function**: User supplied function that is called for each file encountered.
- **Depth**: Specifies the maximum number of file descriptors to be used. `Depth` cannot be greater than `OPEN_MAX` which is described in the `sys/limits.h` header file.

### Return Values

If the tree is exhausted, the **nftw** and **nftw64** subroutine returns a value of 0. If the subroutine pointed to by `fn` returns a nonzero value, **nftw** and **nftw64** stops its tree traversal and returns whatever value was returned by the subroutine pointed to by `fn`. If the **nftw** and **nftw64** subroutine detects an error, it returns a -1 and sets the `errno` global variable to indicate the error.

### Error Codes

If the **nftw** or **nftw64** subroutines detect an error, a value of -1 is returned and the `errno` global variable is set to indicate the error.

The **nftw** and **nftw64** subroutine are unsuccessful if:

- **EACCESS**: Search permission is denied for any component of the `Path` parameter or read permission is denied for `Path`.
- **ENAMETOOLONG**: The length of the path exceeds `PATH_MAX` while `POSIX_NO_TRUNC` is in effect.
- **ENOENT**: The `Path` parameter points to the name of a file that does not exist or points to an empty string.
- **ENOTDIR**: A component of the `Path` parameter is not a directory.

The **nftw** subroutine is unsuccessful if:

- **EOVERFLOW**: A file in `Path` is of a size larger than 2 Gigabytes.
Related Information

The `malloc` subroutine.

The `ftw` subroutine.

---

**nl_langinfo Subroutine**

**Purpose**

Returns information on the language or cultural area in a program's locale.

**Library**

Standard C Library (`libc.a`)

**Syntax**

```c
#include <nl_types.h>
#include <langinfo.h>

char *nl_langinfo (Item)
    nl_item Item;
```

**Description**

The `nl_langinfo` subroutine returns a pointer to a string containing information relevant to the particular language or cultural area defined in the program's locale and corresponding to the `Item` parameter. The active language or cultural area is determined by the default value of the environment variables or by the most recent call to the `setlocale` subroutine. If the `setlocale` subroutine has not been called in the program, then the default C locale values will be returned from `nl_langinfo`.

Values for the `Item` parameter are defined in the `langinfo.h` file.

The following table summarizes the categories for which `nl_langinfo()` returns information, the values the `Item` parameter can take, and descriptions of the returned strings. In the table, radix character refers to the character that separates whole and fractional numeric or monetary quantities. For example, a period (.) is used as the radix character in the U.S., and a comma (,) is used as the radix character in France.

<table>
<thead>
<tr>
<th>Category</th>
<th>Value of item</th>
<th>Returned Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>LC_MONETARY</td>
<td>CRNCYSTR</td>
<td>Currency symbol and its position.</td>
</tr>
<tr>
<td>LC_NUMERIC</td>
<td>RADIXCHAR</td>
<td>Radix character.</td>
</tr>
<tr>
<td>LC_NUMERIC</td>
<td>THOUSEP</td>
<td>Separator for the thousands.</td>
</tr>
<tr>
<td>LC_MESSAGES</td>
<td>YESSTR</td>
<td>Affirmative response for yes/no queries.</td>
</tr>
<tr>
<td>LC_MESSAGES</td>
<td>NOSTR</td>
<td>Negative response for yes/no queries.</td>
</tr>
<tr>
<td>LC_TIME</td>
<td>D_T_FMT</td>
<td>String for formatting date and time.</td>
</tr>
<tr>
<td>LC_TIME</td>
<td>D_FMT</td>
<td>String for formatting date.</td>
</tr>
<tr>
<td>LC_TIME</td>
<td>T_FMT</td>
<td>String for formatting time.</td>
</tr>
<tr>
<td>LC_TIME</td>
<td>AM_STR</td>
<td>Antemeridian affix.</td>
</tr>
<tr>
<td>LC_TIME</td>
<td>PM_STR</td>
<td>Postmeridian affix.</td>
</tr>
<tr>
<td>LC_TIME</td>
<td>DAY_1 through DAY_7</td>
<td>Name of the first day of the week to the seventh day of the week.</td>
</tr>
<tr>
<td>Category</td>
<td>Value of Item</td>
<td>Returned Result</td>
</tr>
<tr>
<td>----------</td>
<td>--------------</td>
<td>-----------------</td>
</tr>
<tr>
<td>LC_TIME</td>
<td>ABDAY_1 through ABDAY-7</td>
<td>Abbreviated name of the first day of the week to the seventh day of the week.</td>
</tr>
<tr>
<td>LC_TIME</td>
<td>MON_1 through MON_12</td>
<td>Name of the first month of the year to the twelfth month of the year.</td>
</tr>
<tr>
<td>LC_TIME</td>
<td>ABMON_1 through ABMON_12</td>
<td>Abbreviated name of the first month of the year to the twelfth month.</td>
</tr>
<tr>
<td>LC_CTYPE</td>
<td>CODESET</td>
<td>Code set currently in use in the program.</td>
</tr>
</tbody>
</table>

**Note:** The information returned by the `nl_langinfo` subroutine is located in a static buffer. The contents of this buffer are overwritten in subsequent calls to the `nl_langinfo` subroutine. Therefore, you should save the returned information.

**Parameter**

*Item*  Information needed from locale.

**Return Values**

In a locale where language information data is not defined, the `nl_langinfo` subroutine returns a pointer to the corresponding string in the C locale. In all locales, the `nl_langinfo` subroutine returns a pointer to an empty string if the *Item* parameter contains an invalid setting.

The `nl_langinfo` subroutine returns a pointer to a static area. Subsequent calls to the `nl_langinfo` subroutine overwrite the results of a previous call.

**Related Information**

The `localeconv` subroutine, `rpmatch` subroutine, `setlocale` subroutine.

Subroutines, Example Programs, and Libraries in *AIX 5L Version 5.3 General Programming Concepts: Writing and Debugging Programs.*


**nlist, nlist64 Subroutine**

**Purpose**

Gets entries from a name list.

**Library**

Standard C Library (`libc.a`)

Berkeley Compatibility Library (`libbsd.a`) for the `nlist` subroutine, 32-bit programs, and POWER-based platforms
Syntax

```
#include <nlist.h>

int nlist ( FileName, NL )
const char *FileName;
struct nlist *NL;

int nlist64 ( FileName, NL64 )
const char *FileName;
struct nlist64 *NL64;
```

Description

The `nlist` and `nlist64` subroutines examine the name list in the object file named by the `FileName` parameter. The subroutine selectively reads a list of values and stores them into an array of `nlist` or `nlist64` structures pointed to by the `NL` or `NL64` parameter, respectively.

The name list specified by the `NL` or `NL64` parameter consists of an array of `nlist` or `nlist64` structures containing symbol names and other information. The list is terminated with an element that has a null pointer or a pointer to a null string in the `n_name` structure member. Each symbol name is looked up in the name list of the file. If the name is found, the value of the symbol is stored in the structure, and the other fields are filled in. If the program was not compiled with the `-g` flag, the `n_type` field may be 0.

If multiple instances of a symbol are found, the information about the last instance is stored. If a symbol is not found, all structure fields except the `n_name` field are set to 0. Only global symbols will be found.

The `nlist` and `nlist64` subroutines run in both 32-bit and 64-bit programs that read the name list of both 32-bit and 64-bit object files, with one exception: in 32-bit programs, `nlist` will return -1 if the specified file is a 64-bit object file.

The `nlist` and `nlist64` subroutines are used to read the name list from XCOFF object files.

The `nlist64` subroutine can be used to examine the system name list kept in the kernel, by specifying `/unix` as the `FileName` parameter. The `knlist` subroutine can also be used to look up symbols in the current kernel namespace.

Note: The `nlist.h` header file has a `#define` field for `n_name`. If a source file includes both `nlist.h` and `netdb.h`, there will be a conflict with the use of `n_name`. If `netdb.h` is included after `nlist.h`, `n_name` will be undefined. To correct this problem, `_n_n_name` should be used instead. If `netdb.h` is included before `nlist.h`, and you need to refer to the `n_name` field of `struct netent`, you should undefine `n_name` by entering:

```
#define n_name
```

The `nlist` subroutine in `libbsd.a` is supported only in 32-bit mode.

Parameters

- `FileName`: Specifies the name of the file containing a name list.
- `NL`: Points to the array of `nlist` structures.
- `NL64`: Points to the array of `nlist64` structures.
Return Values
Upon successful completion, a 0 is returned, even if some symbols could not be found. In the libbsd.a version of nlist, the number of symbols not found in the object file’s name list is returned. If the file cannot be found or if it is not a valid name list, a value of -1 is returned.

Compatibility Interfaces
To obtain the BSD-compatible version of the subroutine 32-bit applications, compile with the libbsd.a library by using the -lbsd flag.

Related Information
The knlist subroutine.
The a.out file in XCOFF format.

Subroutines Overview in AIX 5L Version 5.3 General Programming Concepts: Writing and Debugging Programs.

ns_addr Subroutine

Purpose
Address conversion routines.

Library
Standard C Library (libc.a)

Syntax
#include <sys/types.h>
#include <netns/ns.h>

struct ns_addr(char *cp)

Description
The ns_addr subroutine interprets character strings representing addresses, returning binary information suitable for use in system calls.

The ns_addr subroutine separates an address into one to three fields using a single delimiter and examines each field for byte separators (colon or period). The delimiters are:

. period
: colon
# pound sign

If byte separators are found, each subfield separated is taken to be a small hexadecimal number, and the entirety is taken as a network-byte-ordered quantity to be zero extended in the high-networked-order bytes. Next, the field is inspected for hyphens, which would indicate the field is a number in decimal notation with hyphens separating the millenia. The field is assumed to be a number, interpreted as hexadecimal, if a leading 0x (as in C), a trailing H, (as in Mesa), or any super-octal digits are present. The field is interpreted as octal if a leading 0 is present and there are no super-octal digits. Otherwise, the field is converted as a decimal number.
Parameter

\( cp \) Returns a pointer to the address of a \texttt{ns\_addr} structure.

---

\texttt{ns\_ntoa Subroutine}

\textbf{Purpose}
Address conversion routines.

\textbf{Library}
Standard C Library (\texttt{libc.a})

\textbf{Syntax}

```c
#include <sys/types.h>
#include <netns/ns.h>

char *ns_ntoa (struct ns_addr \textit{ns})
```

\textbf{Description}
The \texttt{ns\_ntoa} subroutine takes addresses and returns ASCII strings representing the address in a notation in common use in the Xerox Development Environment:

\texttt{\langle network number\rangle \langle host number\rangle \langle port number\rangle}

Trailing zero fields are suppressed, and each number is printed in hexadecimal, in a format suitable for input to the \texttt{ns\_addr} subroutine. Any fields lacking super-decimal digits will have a trailing \texttt{H} appended.

\textbf{Note:} The string returned by \texttt{ns\_ntoa} resides in static memory.

\textbf{Parameter}

\textit{ns} Returns a pointer to a string.

---

\texttt{odm\_add\_obj Subroutine}

\textbf{Purpose}
Adds a new object into an object class.

\textbf{Library}
Object Data Manager Library (\texttt{libodm.a})

\textbf{Syntax}

```c
#include <odmi.h>

int odm_add_obj (\texttt{\textbf{ClassSymbol}}, \texttt{DataStructure})
CLASS_SYMBOL ClassSymbol;
struct ClassName \texttt{*DataStructure};
```
Description
The `odm_add_obj` subroutine takes as input the class symbol that identifies both the object class to add and a pointer to the data structure containing the object to be added.

The `odm_add_obj` subroutine opens and closes the object class around the subroutine if the object class was not previously opened. If the object class was previously opened, the subroutine leaves the object class open when it returns.

Parameters

`ClassSymbol` Specifies a class symbol identifier returned from an `odm_open_class` subroutine. If the `odm_open_class` subroutine has not been called, then this identifier is the `ClassName_CLASS` structure that was created by the `odmcreate` command.

`DataStructure` Specifies a pointer to an instance of the C language structure corresponding to the object class referenced by the `ClassSymbol` parameter. The structure is declared in the `.h` file created by the `odmcreate` command and has the same name as the object class.

Return Values
Upon successful completion, an identifier for the object that was added is returned. If the `odm_add_obj` subroutine is unsuccessful, a value of -1 is returned and the `odmerrno` variable is set to an error code.

Error Codes
Failure of the `odm_add_obj` subroutine sets the `odmerrno` variable to one of the following error codes:

- `ODMI_CLASS_DNE`
- `ODMI_CLASS_PERMS`
- `ODMI_INVALID_CLXN`
- `ODMI_INVALID_PATH`
- `ODMI_MAGICNO_ERR`
- `ODMI_OPEN_ERR`
- `ODMI_PARAMS`
- `ODMI_READ_ONLY`
- `ODMI_TOOMANYCLASSES`

See Appendix B, "ODM Error Codes" for explanations of the ODM error codes.

Related Information
The `odm_create_class` subroutine, `odm_open_class` subroutine, `odm_rm_obj` subroutine.

The `odmcreate` command.

See ODM Example Code and Output in AIX 5L Version 5.3 General Programming Concepts: Writing and Debugging Programs for an example of a `.h` file.

Object Data Manager (ODM) Overview for Programmers in AIX 5L Version 5.3 General Programming Concepts: Writing and Debugging Programs.
odm_change_obj Subroutine

Purpose
Changes an object in the object class.

Library
Object Data Manager Library (libodm.a)

Syntax
#include <odmi.h>

int odm_change_obj (ClassSymbol, DataStructure)
CLASS_SYMBOL ClassSymbol;
struct ClassName *DataStructure;

Description
The odm_change_obj subroutine takes as input the class symbol that identifies both the object class to change and a pointer to the data structure containing the object to be changed. The application program must first retrieve the object with an odm_get_obj subroutine call, change the data values in the returned structure, and then pass that structure to the odm_change_obj subroutine.

The odm_change_obj subroutine opens and closes the object class around the change if the object class was not previously opened. If the object class was previously opened, then the subroutine leaves the object class open when it returns.

Parameters
ClassSymbol Specifies a class symbol identifier returned from an odm_open_class subroutine. If the odm_open_class subroutine has not been called, then this identifier is the ClassName_CLASS structure that is created by the odmcreate command.

DataStructure Specifies a pointer to an instance of the C language structure corresponding to the object class referenced by the ClassSymbol parameter. The structure is declared in the .h file created by the odmcreate command and has the same name as the object class.

Return Values
Upon successful completion, a value of 0 is returned. If the odm_change_obj subroutine fails, a value of -1 is returned and the odmerrno variable is set to an error code.

Error Codes
Failure of the odm_change_obj subroutine sets the odmerrno variable to one of the following error codes:
- ODMI_CLASS_DNE
- ODMI_CLASS_PERMS
- ODMI_INVALID_CLXN
- ODMI_INVALID_PATH
- ODMI_MAGICNO_ERR
- ODMI_NO_OBJECT
- ODMI_OPEN_ERR
- ODMI_PARAMS
• **ODMI_READ_ONLY**
• **ODMI_TOOMANYCLASSES**

See Appendix B, "ODM Error Codes" for explanations of the ODM error codes.

**Related Information**
The `odm_get_obj` subroutine.

The `odmchange` command, `odmcreate` command.

See ODM Example Code and Output in AIX 5L Version 5.3 General Programming Concepts: Writing and Debugging Programs for an example of a .h file.

Object Data Manager (ODM) Overview for Programmers in AIX 5L Version 5.3 General Programming Concepts: Writing and Debugging Programs.

---

**odm_close_class Subroutine**

**Purpose**
Closes an ODM object class.

**Library**
Object Data Manager Library (`libodm.a`)

**Syntax**

```c
#include <odmi.h>

int odm_close_class (CLASS_SYMBOL ClassSymbol);
```

**Description**
The `odm_close_class` subroutine closes the specified object class.

**Parameters**

*ClassSymbol* Specifies a class symbol identifier returned from an `odm_open_class` subroutine. If the `odm_open_class` subroutine has not been called, then this identifier is the `ClassName_CLASS` structure that was created by the `odmcreate` command.

**Return Values**
Upon successful completion, a value of 0 is returned. If the `odm_close_class` subroutine is unsuccessful, a value of -1 is returned and the `odmerrno` variable is set to an error code.

**Error Codes**
Failure of the `odm_close_class` subroutine sets the `odmerrno` variable to one of the following error codes:

• **ODMI_CLASS_DNE**
• **ODMI_CLASS_PERMS**
• **ODMI_INVALID_CLXN**
• **ODMI_INVALID_PATH**
Related Information

The `odm_open_class` subroutine creates an object class. However, the `.c` and `.h` files generated by the `odmcreate` command are required to be part of the application.

Parameters

`ClassSymbol` Specifies a class symbol of the form `ClassName_CLASS`, which is declared in the `.h` file created by the `odmcreate` command.

Return Values

Upon successful completion, a value of 0 is returned. If the `odm_create_class` subroutine is unsuccessful, a value of -1 is returned and the `odmerrno` variable is set to an error code.

Error Codes

Failure of the `odm_create_class` subroutine sets the `odmerrno` variable to one of the following error codes:

- `ODMI_CLASS_EXISTS`
- `ODMI_CLASS_PERMS`
- `ODMI_INVALID_CLXN`
- `ODMI_INVALID_PATH`
- `ODMI_MAGICNO_ERR`
- `ODMI_OPEN_ERR`
See Appendix B, "ODM Error Codes" for explanations of the ODM error codes.

Related Information
The odm_mount_class subroutine.

The odmcreate command.

See ODM Example Code and Output in AIX 5L Version 5.3 General Programming Concepts: Writing and Debugging Programs for an example of a .h file.

Object Data Manager (ODM) Overview for Programmers in AIX 5L Version 5.3 General Programming Concepts: Writing and Debugging Programs.

---

odm_err_msg Subroutine

Purpose
Returns an error message string.

Library
Object Data Manager Library (libodm.a)

Syntax
```
#include <odmi.h>

int odm_err_msg (ODMErrno, MessageString)
long ODMErrno;
char **MessageString;
```

Description
The odm_err_msg subroutine takes as input an ODMErrno parameter and an address in which to put the string pointer of the message string that corresponds to the input ODM error number. If no corresponding message is found for the input error number, a null string is returned and the subroutine is unsuccessful.

Parameters
- ODMErrno: Specifies the error code for which the message string is retrieved.
- MessageString: Specifies the address of a string pointer that will point to the returned error message string.

Return Values
Upon successful completion, a value of 0 is returned. If the odm_err_msg subroutine is unsuccessful, a value of -1 is returned, and the MessageString value returned is a null string.

Examples
The following example shows the use of the odm_err_msg subroutine:
```
#include <odmi.h>
char *error_message;
...
*/--------------------------------------------------*/
*/ODMErrno was returned from a previous ODM subroutine call.*/
*/--------------------------------------------------*/
returnstatus = odm_err_msg (odmerrno, &error_message);
```
if ( returnstatus < 0 )
    printf ( "Retrieval of error message failed\n" );
else
    printf ( error_message );

Related Information

See Appendix B, “ODM Error Codes,” on page 1325 for explanations of the ODM error codes.

Object Data Manager (ODM) Overview for Programmers in AIX 5L Version 5.3 General Programming Concepts: Writing and Debugging Programs.

odm_free_list Subroutine

Purpose
Frees memory previously allocated for an odm_get_list subroutine.

Library
Object Data Manager Library (libodm.a)

Syntax
#include <odm.h>

int odm_free_list (ReturnData, DataInfo)
struct ClassName *ReturnData;
struct listinfo *DataInfo;

Description
The odm_free_list subroutine recursively frees up a tree of memory object lists that were allocated for an odm_get_list subroutine.

Parameters
ReturnData
Points to the array of ClassName structures returned from the odm_get_list subroutine.

DataInfo
Points to the listinfo structure that was returned from the odm_get_list subroutine. The listinfo structure has the following form:

```
struct listinfo {
    char ClassName[16]; /* class name for query */
    char criteria[256]; /* query criteria */
    int num; /* number of matches found */
    int valid; /* for ODM use */
    CLASS_SYMBOL class; /* symbol for queried class */
};
```

Return Values
Upon successful completion, a value of 0 is returned. If the odm_free_list subroutine is unsuccessful, a value of -1 is returned and the odmerrno variable is set to an error code.
Error Codes
Failure of the odm_free_list subroutine sets the odmerrno variable to one of the following error codes:

- ODMI_MAGICNO_ERR
- ODMI_PARAMS

See Appendix B, "ODM Error Codes" for explanations of the ODM error codes.

Related Information
The odm_get_list subroutine.

Object Data Manager (ODM) Overview for Programmers in AIX 5L Version 5.3 General Programming Concepts: Writing and Debugging Programs.

odm_get_by_id Subroutine

Purpose
Retrieves an object from an ODM object class by its ID.

Library
Object Data Manager Library (libodm.a)

Syntax
#include <odmi.h>

struct ClassName *odm_get_by_id(ClassSymbol, ObjectID, ReturnData);

Description
The odm_get_by_id subroutine retrieves an object from an object class. The object to be retrieved is specified by passing its ObjectID parameter from its corresponding ClassName structure.

Parameters

ClassSymbol
Specifies a class symbol identifier of the form ClassName_CLASS, which is declared in the .h file created by the odmcreate command.

ObjectID
Specifies an identifier retrieved from the corresponding ClassName structure of the object class.

ReturnData
Specifies a pointer to an instance of the C language structure corresponding to the object class referenced by the ClassSymbol parameter. The structure is declared in the .h file created by the odmcreate command and has the same name as the object class.

Return Values
Upon successful completion, a pointer to the ClassName structure containing the object is returned. If the odm_get_by_id subroutine is unsuccessful, a value of -1 is returned and the odmerrno variable is set to an error code.
Error Codes
Failure of the `odm_get_by_id` subroutine sets the `odmerrno` variable to one of the following error codes:

- `ODMI_CLASS_DNE`
- `ODMI_CLASS_PERMS`
- `ODMI_INVALID_CLXN`
- `ODMI_INVALID_PATH`
- `ODMI_MAGICNO_ERR`
- `ODMI_MALLOC_ERR`
- `ODMI_NO_OBJECT`
- `ODMI_OPEN_ERR`
- `ODMI_PARAMS`
- `ODMI_TOOMANYCLASSES`

See Appendix B, "ODM Error Codes" for explanations of the ODM error codes.

Related Information
The `odm_get_obj` subroutine takes an object class and criteria as input, and returns a list of objects that satisfy the input criteria. The subroutine opens and closes the object class around the subroutine if the object class was not previously opened. If the object class was previously opened, the subroutine leaves the object class open when it returns.

The `odmcreate` command.

See ODM Example Code and Output in AIX 5L Version 5.3 General Programming Concepts: Writing and Debugging Programs for an example of a .h file.

---

**odm_get_list Subroutine**

**Purpose**
Retrieves all objects in an object class that match the specified criteria.

**Library**
Object Data Manager Library (libodm.a)

**Syntax**
```
#include <odmi.h>

struct ClassName *odm_get_list (ClassSymbol, Criteria, ListInfo, MaxReturn, LinkDepth)
struct ClassName_CLASS [ClassSymbol]
char *Criteria
struct listinfo *ListInfo
int MaxReturn [LinkDepth]
```

**Description**
The `odm_get_list` subroutine takes an object class and criteria as input, and returns a list of objects that satisfy the input criteria. The subroutine opens and closes the object class around the subroutine if the object class was not previously opened. If the object class was previously opened, the subroutine leaves the object class open when it returns.
Parameters

ClassSymbol  Specifies a class symbol identifier returned from an `odm_open_class` subroutine. If the `odm_open_class` subroutine has not been called, then this is the `ClassName_CLASS` structure created by the `odmcreate` command.

Criteria  Specifies a string that contains the qualifying criteria for selecting the objects to remove.

ListInfo  Specifies a structure containing information about the retrieval of the objects. The `listinfo` structure has the following form:

```
struct listinfo {
    char ClassName[16];  /* class name used for query */
    char criteria[256];  /* query criteria */
    int num;  /* number of matches found */
    int valid;  /* for ODM use */
    CLASS_SYMBOL class;  /* symbol for queried class */
};
```

MaxReturn  Specifies the expected number of objects to be returned. This is used to control the increments in which storage for structures is allocated, to reduce the `realloc` subroutine copy overhead.

LinkDepth  Specifies the number of levels to recurse for objects with `ODM_LINK` descriptors. A setting of 1 indicates only the top level is retrieved; 2 indicates `ODM_LINK`s will be followed from the top/first level only; 3 indicates `ODM_LINK`s will be followed at the first and second level, and so on.

Return Values

Upon successful completion, a pointer to an array of C language structures containing the objects is returned. This structure matches that described in the `.h` file that is returned from the `odmcreate` command. If no match is found, null is returned. If the `odm_get_list` subroutine fails, a value of -1 is returned and the `odmerrno` variable is set to an error code.

Error Codes

Failure of the `odm_get_list` subroutine sets the `odmerrno` variable to one of the following error codes:

- `ODMI_BAD_CRIT`
- `ODMI_CLASS_DNE`
- `ODMI_CLASS_PERMS`
- `ODMI_INTERNAL_ERR`
- `ODMI_INVALID_CLXN`
- `ODMI_INVALID_PATH`
- `ODMI_LINK_NOT_FOUND`
- `ODMI_MAGICNO_ERR`
- `ODMI_MALLOC_ERR`
- `ODMI_OPEN_ERR`
- `ODMI_PARAMS`
- `ODMI_TOOMANYCLASSES`

See Appendix B, “ODM Error Codes” for explanations of the ODM error codes.

Related Information

The `odm_get_by_id` subroutine, `odm_get_obj` subroutine, `odm_get_first`, or `odm_get_next` subroutine, `odm_open_class` subroutine, or `odm_free_list` subroutine.

The `odmcreate` command, `odmget` command.
For information on qualifying criteria, see "Understanding ODM Object Searches" in AIX 5L Version 5.3 General Programming Concepts: Writing and Debugging Programs.

See "Object Data Manager (ODM) Overview for Programmers" in AIX 5L Version 5.3 General Programming Concepts: Writing and Debugging Programs for an example of a .h file.

Object Data Manager (ODM) Overview for Programmers in AIX 5L Version 5.3 General Programming Concepts: Writing and Debugging Programs.

### odm_get_obj, odm_get_first, or odm_get_next Subroutine

#### Purpose
Retrieves objects, one object at a time, from an ODM object class.

#### Library
Object Data Manager Library (libodm.a)

#### Syntax
```c
#include <odmi.h>

struct ClassName *odm_get_obj ( ClassSymbol, Criteria, ReturnData, FIRST_NEXT)
struct ClassName *odm_get_first (ClassSymbol, Criteria, ReturnData)
struct ClassName *odm_get_next (ClassSymbol, ReturnData)

CLASS_SYMBOL ClassSymbol;
char *Criteria;
struct ClassName *ReturnData;
int FIRST_NEXT;
```

#### Description
The `odm_get_obj`, `odm_get_first`, and `odm_get_next` subroutines retrieve objects from ODM object classes and return the objects into C language structures defined by the .h file produced by the `odmcreate` command.

The `odm_get_obj`, `odm_get_first`, and `odm_get_next` subroutines open and close the specified object class if the object class was not previously opened. If the object class was previously opened, the subroutines leave the object class open upon return.

#### Parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ClassSymbol</td>
<td>Specifies a class symbol identifier returned from an <code>odm_create_class</code> subroutine. If the <code>odm_open_class</code> subroutine has not been called, then this identifier is the <code>ClassName_CLASS</code> structure that was created by the <code>odmcreate</code> command.</td>
</tr>
<tr>
<td>Criteria</td>
<td>Specifies the string that contains the qualifying criteria for retrieval of the objects.</td>
</tr>
<tr>
<td>ReturnData</td>
<td>Specifies the pointer to the data structure in the .h file created by the <code>odmcreate</code> command. The name of the structure in the .h file is <code>ClassName</code>. If the <code>ReturnData</code> parameter is null (ReturnData == null), space is allocated for the parameter and the calling application is responsible for freeing this space at a later time.</td>
</tr>
</tbody>
</table>

If variable length character strings (vchar) are returned, they are referenced by pointers in the `ReturnData` structure. Calling applications must free each vchar between each call to the `odm_get` subroutines; otherwise storage will be lost.
**FIRST_NEXT** Specifies whether to get the first object that matches the criteria or the next object. Valid values are:

- **ODM_FIRST**
  Retrieve the first object that matches the search criteria.

- **ODM_NEXT**
  Retrieve the next object that matches the search criteria. The *Criteria* parameter is ignored if the *FIRST_NEXT* parameter is set to **ODM_NEXT**.

### Return Values
Upon successful completion, a pointer to the retrieved object is returned. If no match is found, null is returned. If an `odm_get_obj`, `odm_get_first`, or `odm_get_next` subroutine is unsuccessful, a value of -1 is returned and the `odmerrno` variable is set to an error code.

### Error Codes
Failure of the `odm_get_obj`, `odm_get_first` or `odm_get_next` subroutine sets the `odmerrno` variable to one of the following error codes:

- **ODMI_BAD_CRIT**
- **ODMI_CLASS_DNE**
- **ODMI_CLASS_PERMS**
- **ODMI_INTERNAL_ERR**
- **ODMI_INVALID_CLXN**
- **ODMI_INVALID_PATH**
- **ODMI_MAGICNO_ERR**
- **ODMI_MALLOC_ERR**
- **ODMI_OPEN_ERR**
- **ODMI_TOOMANYCLASSES**

See Appendix B, "ODM Error Codes" for explanations of the ODM error codes.

### Related Information
The `odm_get_list` ("odm_get_list Subroutine" on page 909) subroutine, `odm_open_class` ("odm_open_class or odm_open_class_rdonly Subroutine" on page 916) subroutine, `odm_rm_by_id` ("odm_rm_by_id Subroutine" on page 917) subroutine, `odm_rm_obj` ("odm_rm_obj Subroutine" on page 919) subroutine.

The `odmcreate` command, `odmget` command.

For more information about qualifying criteria, see "Understanding ODM Object Searches" in AIX 5L Version 5.3 General Programming Concepts: Writing and Debugging Programs.

See ODM Example Code and Output in AIX 5L Version 5.3 General Programming Concepts: Writing and Debugging Programs for an example of a .h file.

Object Data Manager (ODM) Overview for Programmers in AIX 5L Version 5.3 General Programming Concepts: Writing and Debugging Programs.
**odm_initialize Subroutine**

**Purpose**
Prepares ODM for use by an application.

**Library**
Object Data Manager Library (libodm.a)

**Syntax**
```
#include <odmi.h>
int odm_initialize();
```

**Description**
The `odm_initialize` subroutine starts ODM for use with an application program.

**Return Values**
Upon successful completion, a value of 0 is returned. If the `odm_initialize` subroutine is unsuccessful, a value of -1 is returned and the `odmerrno` variable is set to an error code.

**Error Codes**
Failure of the `odm_initialize` subroutine sets the `odmerrno` variable to one of the following error codes:
- `ODMI_INVALID_PATH`
- `ODMI_MALLOC_ERR`

See Appendix B, "ODM Error Codes" for explanations of the ODM error codes.

**Related Information**
The `odm_terminate` subroutine.

---

**odm_lock Subroutine**

**Purpose**
Puts an exclusive lock on the requested path name.

**Library**
Object Data Manager Library (libodm.a)

**Syntax**
```
#include <odmi.h>

int odm_lock (LockPath, TimeOut);
char *LockPath;
int TimeOut;
```

---
Description
The **odm_lock** subroutine is used by an application to prevent other applications or methods from accessing an object class or group of object classes. A lock on a directory path name does not prevent another application from acquiring a lock on a subdirectory or object class within that directory.

*Note:* Coordination of locking is the responsibility of the application accessing the object classes.

The **odm_lock** subroutine returns a lock identifier that is used to call the **odm_unlock** subroutine.

Parameters

- **LockPath**
  Specifies a string containing the path name in the file system in which to locate object classes or the path name to an object class to lock.

- **TimeOut**
  Specifies the amount of time, in seconds, to wait if another application or method holds a lock on the requested object class or classes. The possible values for the **TimeOut** parameter are:

  - **TimeOut = ODM_NOWAIT**
    The **odm_lock** subroutine is unsuccessful if the lock cannot be granted immediately.

  - **TimeOut = Integer**
    The **odm_lock** subroutine waits the specified amount of seconds to retry an unsuccessful lock request.

  - **TimeOut = ODM_WAIT**
    The **odm_lock** subroutine waits until the locked path name is freed from its current lock and then locks it.

Return Values

Upon successful completion, a lock identifier is returned. If the **odm_lock** subroutine is unsuccessful, a value of -1 is returned and the **odmerrno** variable is set to an error code.

Error Codes

Failure of the **odm_lock** subroutine sets the **odmerrno** variable to one of the following error codes:

- **ODMI_BAD_LOCK**
- **ODMI_BAD_TIMEOUT**
- **ODMI_BAD_TOKEN**
- **ODMI_LOCK_BLOCKED**
- **ODMI_LOCK_ENV**
- **ODMI_MALLOC_ERR**
- **ODMI_UNLOCK**

See Appendix B, "ODM Error Codes" for explanations of the ODM error codes.

Related Information

The **odm_unlock** subroutine.

See also "Object Data Manager (ODM) Overview for Programmers" in AIX 5L Version 5.3 General Programming Concepts: Writing and Debugging Programs."
**odm_mount_class Subroutine**

**Purpose**
Retrieves the class symbol structure for the specified object class name.

**Library**
Object Data Manager Library (`libodm.a`)

**Syntax**
```
#include <odmi.h>

CLASS_SYMBOL odm_mount_class (ClassName);
char *ClassName;
```

**Description**
The `odm_mount_class` subroutine retrieves the class symbol structure for a specified object class. The subroutine can be called by applications (for example, the ODM commands) that have no previous knowledge of the structure of an object class before trying to access that class. The `odm_mount_class` subroutine determines the class description from the object class header information and creates a `CLASS_SYMBOL` object class that is returned to the caller.

The object class is not opened by the `odm_mount_class` subroutine. Calling the subroutine subsequent times for an object class that is already open or mounted returns the same `CLASS_SYMBOL` object class.

Mounting a class that links to another object class recursively mounts to the linked class. However, if the recursive mount is unsuccessful, the original `odm_mount_class` subroutine does not fail; the `CLASS_SYMBOL` object class is set up with a null link.

**Parameters**
- `ClassName` specifies the name of an object class from which to retrieve the class description.

**Return Values**
Upon successful completion, a `CLASS_SYMBOL` is returned. If the `odm_mount_class` subroutine is unsuccessful, a value of -1 is returned and the `odmerrno` variable is set to an error code.

**Error Codes**
Failure of the `odm_mount_class` subroutine sets the `odmerrno` variable to one of the following error codes:
- `ODMI_BAD_CLASSNAME`
- `ODMI_BAD_CLXNNAME`
- `ODMI_CLASS_DNE`
- `ODMI_CLASS_PERMS`
- `ODMI_CLXNMAGICNO_ERR`
- `ODMI_INVALID_CLASS`
- `ODMI_INVALID_CLXN`
- `ODMI_MAGICNO_ERR`
- `ODMI_MALLOC_ERR`
- `ODMI_OPEN_ERR`
See Appendix B, "ODM Error Codes" for explanations of the ODM error codes.

Related Information

The `odm_create_class` subroutine.

Object Data Manager (ODM) Overview for Programmers in AIX 5L Version 5.3 General Programming Concepts: Writing and Debugging Programs.

### odm_open_class or odm_open_class_rdonly Subroutine

**Purpose**

Opens an ODM object class.

**Library**

Object Data Manager Library (`libodm.a`)

**Syntax**

```c
#include <odmi.h>

CLASS_SYMBOL odm_open_class (ClassSymbol)
CLASS_SYMBOL ClassSymbol;

CLASS_SYMBOL odm_open_class_rdonly (ClassSymbol)
CLASS_SYMBOL ClassSymbol;
```

**Description**

The `odm_open_class` subroutine can be called to open an object class. Most subroutines implicitly open a class if the class is not already open. However, an application may find it useful to perform an explicit open if, for example, several operations must be done on one object class before closing the class. The `odm_open_class_rdonly` subroutine opens an `odm` database in read-only mode.

**Parameter**

*ClassSymbol* Specifies a class symbol of the form `ClassName_CLASS` that is declared in the `.h` file created by the `odmcreate` command.

**Return Values**

Upon successful completion, a `ClassSymbol` parameter for the object class is returned. If the `odm_open_class` or `odm_open_class_rdonly` subroutine is unsuccessful, a value of -1 is returned and the `odmerrno` variable is set to an error code.

**Error Codes**

Failure of the `odm_open_class` or `odm_open_class_rdonly` subroutine sets the `odmerrno` variable to one of the following error codes:

- ODMI_CLASS_DNE
- ODMI_CLASS_PERMS
ODMI_INVALID_PATH
ODMI_MAGICNO_ERR
ODMI_OPEN_ERR
ODMI_TOOMANYCLASSES

See Appendix B, “ODM Error Codes” for explanations of the ODM error codes.

Related Information
The odm_close_class subroutine.

The odmcreate command.

See ODM Example Code and Output in AIX 5L Version 5.3 General Programming Concepts: Writing and Debugging Programs for an example of a .h file.

Object Data Manager (ODM) Overview for Programmers in AIX 5L Version 5.3 General Programming Concepts: Writing and Debugging Programs.

odm_rm_by_id Subroutine

Purpose
Removes objects specified by their IDs from an ODM object class.

Library
Object Data Manager Library (libodm.a)

Syntax
#include <odmi.h>

int odm_rm_by_id(ClassSymbol, ObjectID)
CLASS_SYMBOL ClassSymbol;
int ObjectID;

Description
The odm_rm_by_id subroutine is called to delete an object from an object class. The object to be deleted is specified by passing its object ID from its corresponding ClassName structure.

Parameters
ClassSymbol Identifies a class symbol returned from an odm_open_class subroutine. If the odm_open_class subroutine has not been called, this is the ClassName_CLASS structure that was created by the odmcreate command.

ObjectID Identifies the object. This information is retrieved from the corresponding ClassName structure of the object class.

Return Values
Upon successful completion, a value of 0 is returned. If the odm_rm_by_id subroutine is unsuccessful, a value of -1 is returned and the odmerrno variable is set to an error code.
Error Codes

Failure of the `odm_rm_by_id` subroutine sets the `odmerrno` variable to one of the following error codes:

- ODMI_CLASS_DNE
- ODMI_CLASS_PERMS
- ODMI_FORK
- ODMI_INVALID_CLXN
- ODMI_INVALID_PATH
- ODMI_MAGICNO_ERR
- ODMI_MALLOC_ERR
- ODMI_NO_OBJECT
- ODMI_OPEN_ERR
- ODMI_OPEN_PIPE
- ODMI_PARAMS
- ODMI_READ_ONLY
- ODMI_READ_PIPE
- ODMI_TOOMANYCLASSES
- ODMI_TOOMANYCLASSES

See Appendix B, “ODM Error Codes” for explanations of the ODM error codes.

Related Information

The `odm_get_obj` subroutine, `odm_get_first`, or `odm_get_next` subroutine, `odm_open_class` subroutine, or `odm_open_class_rdonly` subroutine.

The `odmdelete` command.

Object Data Manager (ODM) Overview for Programmers in AIX 5L Version 5.3 General Programming Concepts: Writing and Debugging Programs.

odm_rm_class Subroutine

Purpose

Removes an object class from the file system.

Library

Object Data Manager Library (`libodm.a`)

Syntax

```
#include <odmi.h>

int odm_rm_class (CLASS_SYMBOL ClassSymbol);
```

Description

The `odm_rm_class` subroutine removes an object class from the file system. All objects in the specified class are deleted.
Parameter

ClassSymbol

Identifies a class symbol returned from the `odm_open_class` subroutine. If the `odm_open_class` subroutine has not been called, this is the `ClassName_CLASS` structure created by the `odmcreate` command.

Return Values

Upon successful completion, a value of 0 is returned. If the `odm_rm_class` subroutine is unsuccessful, a value of -1 is returned and the `odmerrno` variable is set to an error code.

Error Codes

Failure of the `odm_rm_class` subroutine sets the `odmerrno` variable to one of the following error codes:

- `ODMI_CLASS_DNE`
- `ODMI_CLASS_PERMS`
- `ODMI_INVALID_CLXN`
- `ODMI_INVALID_PATH`
- `ODMI_MAGICNO_ERR`
- `ODMI_OPEN_ERR`
- `ODMI_TOOMANYCLASSES`
- `ODMI_UNLINKCLASS_ERR`
- `ODMI_UNLINKCLXN_ERR`

See Appendix B, “ODM Error Codes” for explanations of the ODM error codes.

Related Information

The `odm_open_class` subroutine.

The `odmcreate` command, `odmdrop` command.

Object Data Manager (ODM) Overview for Programmers in AIX 5L Version 5.3 General Programming Concepts: Writing and Debugging Programs.

`odm_rm_obj` Subroutine

Purpose

Removes objects from an ODM object class.

Library

Object Data Manager Library (libodm.a)

Syntax

```
#include <odmi.h>

int odm_rm_obj (ClassSymbol, Criteria)
CLASS_SYMBOL ClassSymbol;
char *Criteria;
```
Description
The **odm_rm_obj** subroutine deletes objects from an object class.

Parameters

- **ClassSymbol**
  Identifies a class symbol returned from an **odm_open_class** subroutine. If the **odm_open_class** subroutine has not been called, this is the **ClassName_CLASS** structure that was created by the **odmcreate** command.

- **Criteria**
  Contains as a string the qualifying criteria for selecting the objects to remove.

Return Values
Upon successful completion, the number of objects deleted is returned. If the **odm_rm_obj** subroutine is unsuccessful, a value of -1 is returned and the **odmerrno** variable is set to an error code.

Error Codes
Failure of the **odm_rm_obj** subroutine sets the **odmerrno** variable to one of the following error codes:

- `ODMI_BAD_CRIT`
- `ODMI_CLASS_DNE`
- `ODMI_CLASS_PERMS`
- `ODMI_FORK`
- `ODMI_INTERNAL_ERR`
- `ODMI_INVALID_CLXN`
- `ODMI_INVALID_PATH`
- `ODMI_MAGICNO_ERR`
- `ODMI_MALLOC_ERR`
- `ODMI_OPEN_ERR`
- `ODMI_OPEN_PIPE`
- `ODMI_PARAMS`
- `ODMI_READ_ONLY`
- `ODMI_READ_PIPE`
- `ODMI_TOOMANYCLASSES`

See Appendix B, “ODM Error Codes” for explanations of the ODM error codes.

Related Information
The **odm_add_obj** subroutine, **odm_open_class** subroutine, and **odmcreate** command. For information on qualifying criteria, see “Understanding ODM Object Searches” in *AIX 5L Version 5.3 General Programming Concepts: Writing and Debugging Programs*. 

Object Data Manager (ODM) Overview for Programmers in *AIX 5L Version 5.3 General Programming Concepts: Writing and Debugging Programs*. 

920 Technical Reference, Volume 1: Base Operating System and Extensions
odm_run_method Subroutine

Purpose
Runs a specified method.

Library
Object Data Manager Library (libodm.a)

Syntax
#include <odmi.h>

int odm_run_method(MethodName, MethodParameters, NewStdOut, NewStdError)
char *MethodName, *MethodParameters;
char **NewStdOut, **NewStdError;

Description
The odm_run_method subroutine takes as input the name of the method to run, any parameters for the method, and addresses of locations for the odm_run_method subroutine to store pointers to the stdout (standard output) and stderr (standard error output) buffers. The application uses the pointers to access the stdout and stderr information generated by the method.

Parameters
MethodName Indicates the method to execute. The method can already be known by the applications, or can be retrieved as part of an odm_get_obj subroutine call.
MethodParameters Specifies a list of parameters for the specified method.
NewStdOut Specifies the address of a pointer to the memory where the standard output of the method is stored. If the NewStdOut parameter points to a null value (*NewStdOut == NULL), standard output is not captured.
NewStdError Specifies the address of a pointer to the memory where the standard error output of the method will be stored. If the NewStdError parameter points to a null value (*NewStdError == NULL), standard error output is not captured.

Return Values
If successful, the odm_run_method subroutine returns the exit status and out_ptr and err_ptr should contain the relevant information. If unsuccessful, the odm_run_method subroutine will return -1 and set the odmerrno variable to an error code.

Note: AIX methods usually return the exit code defined in the cf.h file if the methods exit on error.

Error Codes
Failure of the odm_run_method subroutine sets the odmerrno variable to one of the following error codes:

- ODMI_FORK
- ODMI_MALLOC_ERR
- ODMI_OPENPIPE
- ODMI_PARAMS
- ODMI_READ_PIPE

See Appendix B, "ODM Error Codes" for explanations of the ODM error codes.
Related Information
The odm_get_obj subroutine. ("odm_get_obj, odm_get_first, or odm_get_next Subroutine on page 911) subroutine.

Object Data Manager (ODM) Overview for Programmers in AIX 5L Version 5.3 General Programming Concepts: Writing and Debugging Programs.

**odm_set_path Subroutine**

**Purpose**
Sets the default path for locating object classes.

**Library**
Object Data Manager Library (libodm.a)

**Syntax**
#include <odmi.h>

char *odm_set_path (NewPath)
char *NewPath;

**Description**
The odm_set_path subroutine is used to set the default path for locating object classes. The subroutine allocates memory, sets the default path, and returns the pointer to memory. Once the operation is complete, the calling application should free the pointer using the free subroutine.

**Parameters**
NewPath Contains, as a string, the path name in the file system in which to locate object classes.

**Return Values**
Upon successful completion, a string pointing to the previous default path is returned. If the odm_set_path subroutine is unsuccessful, a value of -1 is returned and the odmerrno variable is set to an error code.

**Error Codes**
Failure of the odm_set_path subroutine sets the odmerrno variable to one of the following error codes:
- ODM_INVALID_PATH
- ODM_MALLOC_ERR

See Appendix B, "ODM Error Codes" for explanations of the ODM error codes.

**Related Information**
The free subroutine. ("malloc, free, realloc, calloc, mallopt, mallinfo, mallinfo_heap, alloca, valloc, or posix_memalign Subroutine on page 769) subroutine.

Object Data Manager (ODM) Overview for Programmers in AIX 5L Version 5.3 General Programming Concepts: Writing and Debugging Programs.
**odm_set_perms Subroutine**

**Purpose**
Sets the default permissions for an ODM object class at creation time.

**Library**
Object Data Manager Library (`libodm.a`)

**Syntax**
```
#include <odmi.h>

int odm_set_perms (NewPermissions);
int NewPermissions;
```

**Description**
The `odm_set_perms` subroutine defines the default permissions to assign to object classes at creation.

**Parameters**
*NewPermissions* Specifies the new default permissions parameter as an integer.

**Return Values**
Upon successful completion, the current default permissions are returned. If the `odm_set_perms` subroutine is unsuccessful, a value of -1 is returned.

**Related Information**
See Appendix B, "ODM Error Codes," on page 1325 for explanations of the ODM error codes.

---

**odm_terminate Subroutine**

**Purpose**
Terminates an ODM session.

**Library**
Object Data Manager Library (`libodm.a`)

**Syntax**
```
#include <odmi.h>

int odm_terminate ( )
```

**Description**
The `odm_terminate` subroutine performs the cleanup necessary to terminate an ODM session. After running an `odm_terminate` subroutine, an application must issue an `odm_initialize` subroutine to resume ODM operations.
Return Values
Upon successful completion, a value of 0 is returned. If the `odm.terminate` subroutine is unsuccessful, a value of -1 is returned and the `odmerrno` variable is set to an error code.

Error Codes
Failure of the `odm.terminate` subroutine sets the `odmerrno` variable to one of the following error codes:
- `ODMI_CLASS_DNE`
- `ODMI_CLASS_PERMS`
- `ODMI_INVALID_CLXN`
- `ODMI_INVALID_PATH`
- `ODMI_LOCK_ID`
- `ODMI_MAGICNO_ERR`
- `ODMI_OPEN_ERR`
- `ODMI_TOOMANYCLASSES`
- `ODMI_UNLOCK`

See Appendix B, "ODM Error Codes" for explanations of the ODM error codes.

Related Information
The `odm.initialize` subroutine ("odm.initialize Subroutine" on page 913) subroutine.

Object Data Manager (ODM) Overview for Programmers in AIX 5L Version 5.3 General Programming Concepts: Writing and Debugging Programs.

odm_unlock Subroutine

Purpose
Releases a lock put on a path name.

Library
Object Data Manager Library (libodm.a)

Syntax
```c
#include <odmi.h>

int odm_unlock (LockID);
int LockID;
```

Description
The `odm_unlock` subroutine releases a previously granted lock on a path name. This path name can be a directory containing subdirectories and object classes.

Parameters
- `LockID` Identifies the lock returned from the `odm_lock` subroutine.
Return Values
Upon successful completion a value of 0 is returned. If the odm_unlock subroutine is unsuccessful, a value of -1 is returned and the odmerrno variable is set to an error code.

Error Codes
Failure of the odm_unlock subroutine sets the odmerrno variable to one of the following error codes:
  - ODMI_LOCK_ID
  - ODMI_UNLOCK

See Appendix B, “ODM Error Codes” for explanations of the ODM error codes.

Related Information
The odm_lock subroutine.

Object Data Manager (ODM) Overview for Programmers in AIX 5L Version 5.3 General Programming Concepts: Writing and Debugging Programs.

open, openx, open64, creat, or creat64 Subroutine

Purpose
Opens a file for reading or writing.

Syntax
```
#include <fcntl.h>

int open(const char *Path, int OFlag [, int Mode]);
int openx(const char *Path, int OFlag, int Mode, int Extension);
int creat(const char *Path, int Mode);
```

Note: The open64 and creat64 subroutines apply to AIX 4.2 and later releases.
```
int open64(const char *Path, int OFlag [, int Mode]);
int creat64(const char *Path, int Mode);
```

Description
Note: The open64 and creat64 subroutines apply to AIX 4.2 and later releases.
The `open`, `openx`, and `creat` subroutines establish a connection between the file named by the `Path` parameter and a file descriptor. The opened file descriptor is used by subsequent I/O subroutines, such as `read` and `write`, to access that file.

The `openx` subroutine is the same as the `open` subroutine, with the addition of an `Extension` parameter, which is provided for device driver use. The `creat` subroutine is equivalent to the `open` subroutine with the `O_WRONLY`, `O_CREAT`, and `O_TRUNC` flags set.

The returned file descriptor is the lowest file descriptor not previously open for that process. No process can have more than `OPEN_MAX` file descriptors open simultaneously.

The file offset, marking the current position within the file, is set to the beginning of the file. The new file descriptor is set to remain open across exec subroutine executions.

The `open64` and `creat64` subroutines are equivalent to the `open` and `creat` subroutines except that the `O_LARGEFILE` flag is set in the open file description associated with the returned file descriptor. This flag allows files larger than `OFF_MAX` to be accessed. If the caller attempts to open a file larger than `OFF_MAX` and `O_LARGEFILE` is not set, the open will fail and `errno` will be set to `EOVERFLOW`.

In the large file enabled programming environment, `open` is redefined to be `open64` and `creat` is redefined to be `creat64`.

**Parameters**

- `Path` Specifies the file to be opened.
**Mode**

Specifies the read, write, and execute permissions of the file to be created (requested by the `O_CREAT` flag). If the file already exists, this parameter is ignored. The Mode parameter is constructed by logically ORing one or more of the following values, which are defined in the `sys/mode.h` file:

- **S_ISUID**
  Enables the `setuid` attribute for an executable file. A process executing this program acquires the access rights of the owner of the file.

- **S_ISGID**
  Enables the `setgid` attribute for an executable file. A process executing this program acquires the access rights of the group of the file. Also, enables the group-inheritance attribute for a directory. Files created in this directory have a group equal to the group of the directory.

The following attributes apply only to files that are directly executable. They have no meaning when applied to executable text files such as shell scripts and `awk` scripts.

- **S_ISVTX**
  Enables the `link/unlink` attribute for a directory. Files cannot be linked to in this directory. Files can only be unlinked if the requesting process has write permission for the directory and is either the owner of the file or the directory.

- **S_ISVTX**
  Enables the save text attribute for an executable file. The program is not unmapped after usage.

- **S_ENFMT**
  Enables enforcement-mode record locking for a regular file. File locks requested with the `lockf` subroutine are enforced.

- **S_IRUSR**
  Permits the file's owner to read it.

- **S_IWUSR**
  Permits the file's owner to write to it.

- **S_IXUSR**
  Permits the file's owner to execute it (or to search the directory).

- **S_IRGRP**
  Permits the file's group to read it.

- **S_IWGRP**
  Permits the file's group to write to it.

- **S_IXGRP**
  Permits the file's group to execute it (or to search the directory).

- **S_IROTH**
  Permits others to read the file.

- **S_IWOTH**
  Permits others to write to the file.

- **S_IXOTH**
  Permits others to execute the file (or to search the directory).

Other mode values exist that can be set with the `mknod` subroutine but not with the `chmod` subroutine.

**Extension**

Provides communication with character device drivers that require additional information or return additional status. Each driver interprets the Extension parameter in a device-dependent way, either as a value or as a pointer to a communication area. Drivers must apply reasonable defaults when the Extension parameter value is 0.

**OFlag**

Specifies the type of access, special open processing, the type of update, and the initial state of the open file. The parameter value is constructed by logically ORing special open processing flags. These flags are defined in the `fcntl.h` file and are described in the following flags.
Flags That Specify Access Type
The following OFlag parameter flag values specify type of access:

- **O_RDONLY** The file is opened for reading only.
- **O_WRONLY** The file is opened for writing only.
- **O_RDWR** The file is opened for both reading and writing.

**Note:** One of the file access values must be specified. Do not use **O_RDONLY**, **O_WRONLY**, or **O_RDWR** together. If none is set, none is used, and the results are unpredictable.

Flags That Specify Special Open Processing
The following OFlag parameter flag values specify special open processing:

- **O_CREAT** If the file exists, this flag has no effect, except as noted under the **O_EXCL** flag. If the file does not exist, a regular file is created with the following characteristics:
  - The owner ID of the file is set to the effective user ID of the process.
  - The group ID of the file is set to the group ID of the parent directory if the parent directory has the SetGroupID attribute (S_ISGID bit) set. Otherwise, the group ID of the file is set to the effective group ID of the calling process.
  - The file permission and attribute bits are set to the value of the **Mode** parameter, modified as follows:
    - All bits set in the process file mode creation mask are cleared. (The file creation mask is described in the umask subroutine.)
    - The S_ISVTX attribute bit is cleared.

  **O_EXCL** If the **O_EXCL** and **O_CREAT** flags are set, the open is unsuccessful if the file exists.

  **Note:** The **O_EXCL** flag is not fully supported for Network File Systems (NFS). The NFS protocol does not guarantee the designed function of the **O_EXCL** flag.

- **O_NSHARE** Assures that no process has this file open and precludes subsequent opens. If the file is on a physical file system and is already open, this open is unsuccessful and returns immediately unless the OFlag parameter also specifies the **O_DEFER** flag. This flag is effective only with physical file systems.

  **Note:** This flag is not supported by NFS.

- **O_RSHARE** Assures that no process has this file open for writing and precludes subsequent opens for writing. The calling process can request write access. If the file is on a physical file system and is open for writing or open with the **O_NSHARE** flag, this open fails and returns immediately unless the OFlag parameter also specifies the **O_DEFER** flag.

  **Note:** This flag is not supported by NFS.

- **O_DEFER** The file is opened for deferred update. Changes to the file are not reflected on permanent storage until an fsync ("fsync or fsync_range Subroutine" on page 317) subroutine operation is performed. If no fsync subroutine operation is performed, the changes are discarded when the file is closed.

  **Note:** This flag is not supported by NFS or JFS2, and the flag will be quietly ignored.

- **O_NOCTTY** This flag specifies that the controlling terminal should not be assigned during this open.

- **O_TRUNC** If the file does not exist, this flag has no effect. If the file exists, is a regular file, and is successfully opened with the **O_RDWR** flag or the **O_WRONLY** flag, all of the following apply:
  - The length of the file is truncated to 0.
  - The owner and group of the file are unchanged.
  - The SetUserID attribute of the file mode is cleared.
  - The SetUserI attribute of the file is cleared.

- **O_DIRECT** This flag specifies that direct i/o will be used for this file while it is opened.
O_CIO This flag specifies that concurrent I/O (CIO) will be used for the file while it is opened. Because implementing concurrent readers and writers utilizes the direct I/O path (with more specific requirements to improve performance for running database on file system), this flag will override the O_DIRECT flag if the two options are specified at the same time. Currently, only JFS2 and namefs, which includes a selected subset of JFS2 files/directories, support CIO. The length of data to be read/written and the file offset must be page-aligned to be transferred as direct I/O with concurrent readers and writers.

The O_CIO flag is exclusive. If the file is opened in any other way (for example, using the O_DIO flag or opening the file normally), the open will fail. If the file is opened using the O_CIO and another process attempts to open the file another way, the open will fail. The O_CIO flag also prevents the mmap subroutine and the shmat subroutine access to the file. The mmap subroutine and the shmat subroutine return EINVAL if they are used on a file that was opened using the O_CIO flag.

O_SNAPSHOT The file being opened contains a JFS2 snapshot. Subsequent read calls using this file descriptor will read the cooked snapshot rather than the raw snapshot blocks. A snapshot can only have one active open file descriptor for it.

The open subroutine is unsuccessful if any of the following conditions are true:
- The file supports enforced record locks and another process has locked a portion of the file.
- The file is on a physical file system and is already open with the O_RSHARE flag or the O_NSHARE flag.
- The file does not allow write access.
- The file is already opened for deferred update.

Flag That Specifies Type of Update
A program can request some control on when updates should be made permanent for a regular file opened for write access. The following OFlag parameter values specify the type of update performed:

O_SYNC: If set, updates to regular files and writes to block devices are synchronous updates. File update is performed by the following subroutines:
- fclear
- ftruncate
- open with O_TRUNC
- write

On return from a subroutine that performs a synchronous update (any of the preceding subroutines, when the O_SYNC flag is set), the program is assured that all data for the file has been written to permanent storage, even if the file is also open for deferred update.

Note: The O_DSYNC flag applies to AIX 4.2.1 and later releases.

O_DSYNC: If set, the file data as well as all file system meta-data required to retrieve the file data are written to their permanent storage locations. File attributes such as access or modification times are not required to retrieve file data, and as such, they are not guaranteed to be written to their permanent storage locations before the preceding subroutines return. (Subroutines listed in the O_SYNC description.)

O_SYNC | O_DSYNC: If both flags are set, the file’s data and all of the file’s meta-data (including access time) are written to their permanent storage locations.
**Note:** The O_RSYNC flag applies to AIX 4.3 and later releases.

**O_RSYNC:** This flag is used in combination with O_SYNC or D_SYNC, and it extends their write operation behaviors to read operations. For example, when O_SYNC and R_SYNC are both set, a read operation will not return until the file’s data and all of the file’s meta-data (including access time) are written to their permanent storage locations.

---

### Flags That Define the Open File Initial State

The following OFlag parameter flag values define the initial state of the open file:

- **O_APPEND**
  - The file pointer is set to the end of the file prior to each write operation.

- **O_DELAY**
  - Specifies that if the open subroutine could not succeed due to an inability to grant the access on a physical file system required by the O_RSHARE flag or the O_NSHARE flag, the process blocks instead of returning the ETXTBSY error code.

- **O_NDELAY**
  - Opens with no delay.

- **O_NONBLOCK**
  - Specifies that the open subroutine should not block.

The O_NDELAY flag and the O_NONBLOCK flag are identical except for the value returned by the read and write subroutines. These flags mean the process does not block on the state of an object, but does block on input or output to a regular file or block device.

The O_DELAY flag is relevant only when used with the O_NSHARE or O_RSHARE flags. It is unrelated to the O_NDELAY and O_NONBLOCK flags.

### General Notes on OFlag Parameter Flags

- When opening a file on a physical file system with the O_NSHARE flag or the O_RSHARE flag, if the file is already open with conflicting access the following can occur:
  - If the O_DELAY flag is clear (the default), the open subroutine is unsuccessful.
  - If the O_DELAY flag is set, the open subroutine blocks until there is no conflicting open. There is no deadlock detection for processes using the O_DELAY flag.

- When opening a file on a physical file system that has already been opened with the O_NSHARE flag, the following can occur:
  - If the O_DELAY flag is clear (the default), the open is unsuccessful immediately.
  - If the O_DELAY flag is set, the open blocks until there is no conflicting open.

- When opening a file with the O_RDWR, O_WRONLY, or O_TRUNC flag, and the file is already open with the O_RSHARE flag:
  - If the O_DELAY flag is clear (the default), the open is unsuccessful immediately.
  - If the O_DELAY flag is set, the open blocks until there is no conflicting open.

- When opening a first-in-first-out (FIFO) with the O_RDONLY flag, the following can occur:
  - If the O_NDELAY and O_NONBLOCK flags are clear, the open opens the file for reading. If the file is already open for reading (even by the calling process), the open subroutine returns without delay.
  - If the O_NDELAY flag or the O_NONBLOCK flag is set, the open succeeds immediately even if no process has the FIFO open for writing.

- When opening a FIFO with the O_WRONLY flag, the following can occur:
If the **O_NDELAY** and **O_NONBLOCK** flags are clear (the default), the open blocks until a process opens the file for reading. If the file is already open for writing (even by the calling process), the **open** subroutine returns without delay.

If the **O_NDELAY** flag or the **O_NONBLOCK** flag is set, the **open** subroutine returns an error if no process currently has the file open for reading.

When opening a block special or character special file that supports nonblocking opens, such as a terminal device, the following can occur:

- If the **O_NDELAY** and **O_NONBLOCK** flags are clear (the default), the open blocks until the device is ready or available.
- If the **O_NDELAY** flag or the **O_NONBLOCK** flag is set, the **open** subroutine returns without waiting for the device to be ready or available. Subsequent behavior of the device is device-specific.

Any additional information on the effect, if any, of the **O_NDELAY**, **O_RDONLY**, **O_WRONLY**, and **O_RDWR** flags on a specific device is documented in the description of the special file related to the device type.

If path refers to a STREAMS file, **oflag** may be constructed from **O_NONBLOCK** OR-ed with either **O_RDONLY**, **O_WRONLY** or **O_RDWR**. Other flag values are not applicable to STREAMS devices and have no effect on them. The value **O_NONBLOCK** affects the operation of STREAMS drivers and certain functions applied to file descriptors associated with STREAMS files. For STREAMS drivers, the implementation of **O_NONBLOCK** is device-specific.

If path names the master side of a pseudo-terminal device, then it is unspecified whether **open** locks the slave side so that it cannot be opened. Portable applications must call **unlockpt** before opening the slave side.

The largest value that can be represented correctly in an object of type **off_t** will be established as the offset maximum in the open file description.

**Return Values**

Upon successful completion, the file descriptor, a nonnegative integer, is returned. Otherwise, a value of -1 is returned, no files are created or modified, and the **errno** global variable is set to indicate the error.

**Error Codes**

The **open**, **openx**, and **creat** subroutines are unsuccessful and the named file is not opened if one or more of the following are true:

- **EACCES**
  - The file exists and the type of access specified by the **OFlag** parameter is denied.
  - Search permission is denied on a component of the path prefix specified by the **Path** parameter. Access could be denied due to a secure mount.
  - The file does not exist and write permission is denied for the parent directory of the file to be created.
  - The **O_TRUNC** flag is specified and write permission is denied.

- **EAGAIN**
  - The **O_TRUNC** flag is set and the named file contains a record lock owned by another process.

- **EDQUOT**
  - The directory in which the entry for the new link is being placed cannot be extended, or an i-node could not be allocated for the file, because the user or group quota of disk blocks or i-nodes in the file system containing the directory has been exhausted.

- **EEXIST**
  - The **O_CREAT** and **O_EXCL** flags are set and the named file exists.
An attempt was made to write a file that exceeds the process' file limit or the maximum file size. If the user has set the environment variable XPG_SUS_ENV=ON prior to execution of the process, then the SIGXFSZ signal is posted to the process when exceeding the process' file size limit.

A signal was caught during the open subroutine.

The path parameter names a STREAMS file and a hangup or error occurred.

Named file is a directory and write access is required (the O_WRONLY or O_RDWR flag is set in the OFlag parameter).

The system limit for open file descriptors per process has already been reached (OPEN_MAX).

The length of the Path parameter exceeds the system limit (PATH_MAX); or a path-name component is longer than NAME_MAX and _POSIX_NO_TRUNC is in effect.

The system file table is full.

The O_CREAT flag is not set and the named file does not exist; or the O_CREAT flag is not set and either the path prefix does not exist or the Path parameter points to an empty string.

The Path parameter names a STREAMS file and the system is unable to allocate resources.

The directory or file system that would contain the new file cannot be extended.

The Path argument names a STREAMS-based file and the system is unable to allocate a STREAM.

A component of the path prefix specified by the Path component is not a directory.

One of the following is true:

- Named file is a character special or block special file, and the device associated with this special file does not exist.
- Named file is a multiplexed special file and either the channel number is outside of the valid range or no more channels are available.
- The O_DELAY flag or the O_NONBLOCK flag is set, the named file is a FIFO, the O_WRONLY flag is set, and no process has the file open for reading.

A file greater than one terabyte was opened on the 32-bit kernel in JFS2. The exact max size is specified in MAX_FILESIZE and may be obtained using the pathconf system call. Any file larger than that cannot be opened on the 32-bit kernel, but can be created and opened on the 64-bit kernel.

Named file resides on a read-only file system and write access is required (either the O_WRONLY, O_RDWR, O_CREAT (if the file does not exist), or O_TRUNC flag is set in the OFlag parameter).

File is on a physical file system and is already open in a manner (with the O_RSHARE or O_NSHARE flag) that precludes this open; or the O_NSHARE or O_RSHARE flag was requested with the O_NDELAY flag set, and there is a conflicting open on a physical file system.

Note: The EOVERFLOW error code applies to AIX 4.2 and later releases.

EOVERFLOW A call was made to open and creat and the file already existed and its size was larger than OFF_MAX and the O_LARGEFILE flag was not set.

The open, openx, and creat subroutines are unsuccessful if one of the following are true:

- The Path parameter points outside of the allocated address space of the process.
- The value of the OFlag parameter is not valid.
- Too many symbolic links were encountered in translating the Path parameter.
- The file specified by the Path parameter is a pure procedure (shared text) file that is currently executing, and the O_WRONLY or O_RDWR flag is set in the OFlag parameter.
Related Information
The chmod subroutine, close subroutine, exec subroutine, fcntl subroutine, dup subroutine, dup2 subroutine, fsync subroutine, ioctl subroutine, lockfx subroutine, lseek subroutine, read subroutine, stat subroutine, umask subroutine, write subroutine.

The Input and Output Handling in AIX 5L Version 5.3 General Programming Concepts: Writing and Debugging Programs.

opendir, readdir, telldir, seekdir, rewinddir, closedir, opendir64, readdir64, telldir64, seekdir64, rewinddir64, or closedir64 Subroutine

Purpose
Performs operations on directories.

Library
Standard C Library (libc.a)

Syntax
#include <dirent.h>

DIR *opendir (DirectoryName)
const char *DirectoryName;

struct dirent *readdir (DirectoryPointer)
DIR *DirectoryPointer;
long int telldir (DirectoryPointer)
DIR *DirectoryPointer;
void seekdir (DirectoryPointer, Location)
DIR *DirectoryPointer;
long Location;
void rewinddir (DirectoryPointer)
DIR *DirectoryPointer;
int closedir (DirectoryPointer)
DIR *DirectoryPointer;

DIR *opendir64 (DirectoryName)
const char *DirectoryName;

struct dirent64 *readdir64 (DirectoryPointer)
DIR64 *DirectoryPointer;
offset_t telldir64 (DirectoryPointer)
DIR64 *DirectoryPointer;
void seekdir64 (DirectoryPointer, Location)
DIR64 *DirectoryPointer;
offset_t Location;
void rewinddir64 (DirectoryPointer)
DIR64 *DirectoryPointer;
int closedir64 (DirectoryPointer)
DIR64 *DirectoryPointer;
Description

Attention: Do not use the readdir subroutine in a multithreaded environment. See the multithread alternative in the readdir subroutine article.

The opendir subroutine opens the directory designated by the DirectoryName parameter and associates a directory stream with it.

Note: An open directory must always be closed with the closedir subroutine to ensure that the next attempt to open that directory is successful.

The opendir subroutine also returns a pointer to identify the directory stream in subsequent operations. The null pointer is returned when the directory named by the DirectoryName parameter cannot be accessed or when not enough memory is available to hold the entire stream. A successful call to any of the exec (exec: execl, execlp, execv, execve, execvp, or execl Subroutine” on page 235) functions closes any directory streams opened in the calling process.

The readdir subroutine returns a pointer to the next directory entry. The readdir subroutine returns entries for . (dot) and .. (dot dot), if present, but never returns an invalid entry (with d_ino set to 0). When it reaches the end of the directory, or when it detects an invalid seekdir operation, the readdir subroutine returns the null value. The returned pointer designates data that may be overwritten by another call to the readdir subroutine on the same directory stream. A call to the readdir subroutine on a different directory stream does not overwrite this data. The readdir subroutine marks the st_atime field of the directory for update each time the directory is actually read.

The telldir subroutine returns the current location associated with the specified directory stream.

The seekdir subroutine sets the position of the next readdir subroutine operation on the directory stream. An attempt to seek an invalid location causes the readdir subroutine to return the null value the next time it is called. The position should be that returned by a previous telldir subroutine call.

The rewinddir subroutine resets the position of the specified directory stream to the beginning of the directory.

The closedir subroutine closes a directory stream and frees the structure associated with the DirectoryPointer parameter. If the closedir subroutine is called for a directory that is already closed, the behavior is undefined. To prevent this, always initialize the DirectoryPointer parameter to null after closure.

If you use the fork ("fork, f_fork, or vfork Subroutine” on page 287) subroutine to create a new process from an existing one, either the parent or the child (but not both) may continue processing the directory stream using the readdir, rewinddir, or seekdir subroutine.

The opendir64 subroutine is similar to the opendir subroutine except that it returns a pointer to an object of type DIR64.

Note: An open directory by opendir64 subroutine must always be closed with the closedir64 subroutine to ensure that the next attempt to open that directory is successful. In addition, it must be operated using the 64-bit interfaces (readdir64, telldir64, seekdir64, rewinddir64, and closedir64) to obtain the correct directory information.

The readdir64 subroutine is similar to the readdir subroutine except that it returns a pointer to an object of type struct dirent64.

The telldir64 subroutine is similar to the telldir subroutine except that it returns the current directory location in an offset_t format.
The `seekdir64` subroutine is similar to the `seekdir` subroutine except that the `Location` parameter is set in the format of `offset_t`.

The `rewinddir64` subroutine resets the position of the specified directory stream (obtained by the `opendir64` subroutine) to the beginning of the directory.

**Parameters**

- **DirectoryName**: Names the directory.
- **DirectoryPointer**: Points to the `DIR` or `DIR64` structure of an open directory.
- **Location**: Specifies the offset of an entry relative to the start of the directory.

**Return Values**

On successful completion, the `opendir` subroutine returns a pointer to an object of type `DIR`, and the `opendir64` subroutine returns a pointer to an object of type `DIR64`. Otherwise, a null value is returned and the `errno` global variable is set to indicate the error.

On successful completion, the `readdir` subroutine returns a pointer to an object of type `struct dirent`, and the `readdir64` subroutine returns a pointer to an object of type `struct dirent64`. Otherwise, a null value is returned and the `errno` global variable is set to indicate the error. When the end of the directory is encountered, a null value is returned and the `errno` global variable is not changed by this function call.

On successful completion, the `telldir` or `telldir64` subroutine returns the current location associated with the specified directory stream. Otherwise, a null value is returned.

On successful completion, the `closedir` or `closedir64` subroutine returns a value of 0. Otherwise, a value of -1 is returned and the `errno` global variable is set to indicate the error.

**Error Codes**

If the `opendir` subroutine is unsuccessful, it returns a null value and sets the `errno` global variable to one of the following values:

- **EACCES**: Indicates that search permission is denied for any component of the `DirectoryName` parameter, or read permission is denied for the `DirectoryName` parameter.
- **ENAMETOOLONG**: Indicates that the length of the `DirectoryName` parameter argument exceeds the `PATH_MAX` value, or a path-name component is longer than the `NAME_MAX` value while the `POSIX_NO_TRUNC` value is in effect.
- **ENOENT**: Indicates that the named directory does not exist.
- **ENOTDIR**: Indicates that a component of the `DirectoryName` parameter is not a directory.
- **EMFILE**: Indicates that too many file descriptors are currently open for the process.
- **ENFILE**: Indicates that too many file descriptors are currently open in the system.

If the `readdir` or `readdir64` subroutine is unsuccessful, it returns a null value and sets the `errno` global variable to the following value:

- **EBADF**: Indicates that the `DirectoryPointer` parameter argument does not refer to an open directory stream.

If the `closedir` or `closedir64` subroutine is unsuccessful, it returns a value of -1 and sets the `errno` global variable to the following value:

- **EBADF**: Indicates that the `DirectoryPointer` parameter argument does not refer to an open directory stream.
Examples

To search a directory for the entry name:

```c
len = strlen(name);
DirectoryPointer = opendir(".");
for (dp = readdir(DirectoryPointer); dp != NULL; dp = readdir(DirectoryPointer))
    if (dp->d_namlen == len && !strcmp(dp->d_name, name)) {
        closedir(DirectoryPointer);
        DirectoryPointer=NULL  //To prevent multiple closure
        return FOUND;
    }
}
closedir(DirectoryPointer);
DirectoryPointer=NULL  //To prevent multiple closure
```

Related Information

The `close` subroutine, `exec` subroutine, `fork` subroutine, `lseek` subroutine, `open` subroutine, or `creat` subroutine. These subroutines are generally used subsequent to a successful `pam_authenticate()` call in order to verify whether the authenticated user should be granted access.

Files, Directories, and File Systems for Programmers in AIX 5L Version 5.3 General Programming Concepts: Writing and Debugging Programs.

---

**pam_acct_mgmt Subroutine**

**Purpose**

Validates the user's account.

**Library**

PAM Library (libpam.a)

**Syntax**

```c
#include <security/pam_appl.h>

int pam_acct_mgmt (PAMHandle, Flags)
pam_handle_t *PAMHandle;
int Flags;
```

**Description**

The `pam_acct_mgmt` subroutine performs various checks on the user's account to determine if it is valid. These checks can include account and password expiration, and access restrictions. This subroutine is generally used subsequent to a successful `pam_authenticate()` call in order to verify whether the authenticated user should be granted access.

**Parameters**

- **PAMhandle**
  The PAM handle representing the current user authentication session. This handle is obtained by a call to `pam_start()`.
Flags

The Flags argument can be a logically OR'd combination of the following:

- **PAM_SILENT**
  - No messages should be displayed
- **PAM_DISALLOW_NULL_AUTHTOK**
  - Do not authenticate a user with a NULL authentication token.

Return Values

Upon successful completion, `pam_acct_mgmt` returns `PAM_SUCCESS`. If the routine fails, a different error will be returned, depending on the actual error.

Error Codes

- **PAM_ACCT_EXPIRED**
  - The user’s account has expired.
- **PAM_NEW_AUTHTOK_REQD**
  - The user’s password needs changed. This is usually due to password aging or because it was last set by an administrator. At this stage most user’s can still change their passwords; applications should call `pam_chauthtok()` and have the user promptly change their password.
- **PAM_AUTHTok_EXPIRED**
  - The user’s password has expired. Unlike `PAM_NEW_AUTHTOK_REQD`, the password cannot be changed by the user.
- **PAM_USER_UNKNOWN**
  - The user is not known.
- **PAM_OPEN_ERR**
  - One of the PAM authentication modules could not be loaded.
- **PAM_SYMBOL_ERR**
  - A necessary item is not available to a PAM module.
- **PAM_SERVICE_ERR**
  - An error occurred in a PAM module.
- **PAM_SYSTEM_ERR**
  - A system error occurred.
- **PAM_BUF_ERR**
  - A memory error occurred.
- **PAM_CONV_ERR**
  - A conversation error occurred.
- **PAM_PERM_DENIED**
  - Access permission was denied to the user.

Related Information

- "pam_authenticate Subroutine" on page 947
- "pam_open_session Subroutine" on page 951
- "pam_setcred Subroutine" on page 951
- "pam_sm_acct_mgmt Subroutine" on page 953
- "pam_start Subroutine" on page 960

pam_authenticate Subroutine

Purpose

Attempts to authenticate a user through PAM.

Library

PAM Library (libpam.a)

Syntax

```c
#include <security/pam_appl.h>

int pam_authenticate (PAMHandle *PAMHandle, Flags)

pam_handle_t *PAMHandle;
int Flags;
```
Description
The pam_authenticate subroutine authenticates a user through PAM. The authentication method used is determined by the authentication modules configured in the /etc/pam.conf stack. Most authentication requires a password or other user input but is dependent on the modules in use.

Before attempting authentication through pam_authenticate, ensure that all of the applicable PAM information has been set through the initial call to pam_start() and subsequent calls to pam_set_item(). If any necessary information is not set, PAM modules can prompt the user for information through the routine defined in PAM_CONV. If required information is not provided and PAM_CONV is not set, the authentication fails.

On failure, it is the responsibility of the calling application to maintain a count of authentication attempts and to reinvoke the subroutine if the count has not exceeded a defined limit. Some authentication modules maintain an internal count and return PAM_MAXTRIES if the limit is reached. After the stack of authentication modules has finished with either success or failure, PAM_AUTHTOK is cleared in the handle.

Parameters

PAMhandle
The PAM handle representing the current user authentication session. This handle is obtained by a call to pam_start().

Flags
The Flags argument can be a logically OR’d combination of the following:

- PAM_SILENT
  - No messages should be displayed
- PAM_DISALLOW_NULL_AUTHTOK
  - Do not authenticate a user with a NULL authentication token.

Return Values
Upon successful completion, pam_authenticate returns PAM_SUCCESS. If the routine fails, a different error will be returned, depending on the actual error.

Error Codes

PAM_AUTH_ERR
An error occurred in authentication, usually because of an invalid authentication token.

PAM_CRED_INSUFFICIENT
The user has insufficient credentials to access the authentication data.

PAM_AUTHINFO_UNAVAIL
The authentication information cannot be retrieved.

PAM_USER_UNKNOWN
The user is not known.

PAM_MAXTRIES
The maximum number of authentication retries has been reached.

PAM_OPEN_ERR
One of the PAM authentication modules could not be loaded.

PAM_SYMBOL_ERR
A necessary item is not available to a PAM module.

PAM_SERVICE_ERR
An error occurred in a PAM module.

PAM_SYSTEM_ERR
A system error occurred.

PAM_BUF_ERR
A memory error occurred.

PAM_CONV_ERR
A conversation error occurred.

PAM_PERM_DENIED
Access permission was denied to the user.
pam_chauthtok Subroutine

Purpose
Changes the user’s authentication token (typically passwords).

Library
PAM Library (libpam.a)

Syntax
```c
#include <security/pam_appl.h>

int pam_chauthtok (PAMHandle, Flags);
```

Description
The pam_chauthtok subroutine changes a user’s authentication token through the PAM framework. Prior to changing the password, the subroutine performs preliminary tests to ensure that necessary hosts and information, depending on the password service, are there. If any of these tests fail, PAM_TRY_AGAIN is returned. To request information from the user, pam_chauthtok can use the conversation function that is defined in the PAM handle, PAMHandle. After the subroutine is finished, the values of PAM_AUTHTOK and PAM_OLD_AUTHTOK are cleared in the handle for added security.

Parameters

- **PAMHandle**
  The PAM handle representing the current user authentication session. This handle is obtained by a call to pam_start().

- **Flags**
  The Flags argument can be a logically OR’d combination of the following:
  - **PAM_SILENT**
    - No messages should be displayed
  - **PAM_CHANGE_EXPIRED_AUTHTOK**
    - Only expired passwords should be changed. If this flag is not included, all users using the related password service are forced to update their passwords. This is typically used by a login application after determining password expiration. It should not generally be used by applications dedicated to changing passwords.

Return Values
Upon successful completion, pam_chauthtok returns PAM_SUCCESS and the authentication token of the user, as defined for a given password service, is changed. If the routine fails, a different error is returned, depending on the actual error.

Error Codes

- **PAM_AUTHTOK_ERR**
  A failure occurred while updating the authentication token.
PAM_TRY_AGAIN
Preliminary checks for changing the password have failed. Try again later.

PAM_AUTHTOK_RECOVERY_ERR
An error occurred while trying to recover the authentication information.

PAM_AUTHTOK_LOCK_BUSY
Cannot get the authentication token lock. Try again later.

PAM_AUTHTOK_DISABLE_AGING
Authentication token aging checks are disabled and were not performed.

PAM_USER_UNKNOWN
The user is not known.

PAM_OPEN_ERR
One of the PAM authentication modules could not be loaded.

PAM_SYMBOL_ERR
A necessary item is not available to a PAM module.

PAM_SERVICE_ERR
An error occurred in a PAM module.

PAM_SYSTEM_ERR
A system error occurred.

PAM_BUF_ERR
A memory error occurred.

PAM_CONV_ERR
A conversation error occurred.

PAM_PERM_DENIED
Access permission was denied to the user.

Related Information
“pam_acct_mgmt Subroutine” on page 936,  
“pam_authenticate Subroutine” on page 937,  
“pam_open_session Subroutine” on page 947,  
“pam_setcred Subroutine” on page 951,  
“pam_sm_chauthtok Subroutine” on page 955,  
“pam_start Subroutine” on page 960

pam_close_session Subroutine

Purpose
Ends a currently open PAM user session.

Library
PAM Library (libpam.a)

Syntax
#include <security/pam_appl.h>

int pam_close_session (PAMHandle, Flags)
pam_handle_t *PAMHandle;
int Flags;

Description
The pam_close_session subroutine ends a PAM user session started by pam_open_session().

Parameters

PAMhandle
The PAM handle representing the current user authentication session. This handle is obtained by a call to pam_start().

Flags
The following flag may be set:

• PAM_SILENT
  – No messages should be displayed
Return Values
Upon successful completion, `pam_close_session` returns `PAM_SUCCESS`. If the routine fails, a different error is returned, depending on the actual error.

Error Codes

<table>
<thead>
<tr>
<th>Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>PAM_SESSION_ERR</td>
<td>An error occurred while creating/removing an entry for the new session.</td>
</tr>
<tr>
<td>PAM_USER_UNKNOWN</td>
<td>The user is not known.</td>
</tr>
<tr>
<td>PAM_OPEN_ERR</td>
<td>One of the PAM authentication modules could not be loaded.</td>
</tr>
<tr>
<td>PAM_SYMBOL_ERR</td>
<td>A necessary item is not available to a PAM module.</td>
</tr>
<tr>
<td>PAM_SERVICE_ERR</td>
<td>An error occurred in a PAM module.</td>
</tr>
<tr>
<td>PAM_SYSTEM_ERR</td>
<td>A system error occurred.</td>
</tr>
<tr>
<td>PAM_BUF_ERR</td>
<td>A memory error occurred.</td>
</tr>
<tr>
<td>PAM_CONV_ERR</td>
<td>A conversation error occurred.</td>
</tr>
<tr>
<td>PAM_PERM_DENIED</td>
<td>Access permission was denied to the user.</td>
</tr>
</tbody>
</table>

Related Information

- “pam_open_session Subroutine” on page 947
- “pam_sm_close_session Subroutine” on page 957
- “pam_start Subroutine” on page 960

**pam_end Subroutine**

**Purpose**
Ends an existing PAM authentication session.

**Library**
PAM Library (`libpam.a`)

**Syntax**

```c
#include <security/pam_appl.h>

int pam_end (PAMHandle PAMHandle, Status Status);
pam_handle_t *PAMHandle;
int Status;
```

**Description**
The `pam_end` subroutine finishes and cleans up the authentication session represented by the PAM handle `PAMHandle`. `Status` denotes the current state of the `PAMHandle` and is passed through to a `cleanup()` function so that the memory used during that session can be properly unallocated. The `cleanup()` function can be set in the `PAMHandle` by PAM modules through the `pam_set_data()` routine. Upon completion of the subroutine, the PAM handle and associated memory is no longer valid.

**Parameters**

- **PAMhandle**
  The PAM handle representing the current user authentication session. This handle is obtained by a call to `pam_start()`.

- **Status**
  The state of the last PAM call. Some modules need to be cleaned according to error codes.
Return Values
Upon successful completion, pam_end returns PAM_SUCCESS. If the routine fails, a different error is returned, depending on the actual error.

Error Codes
PAM_SYSTEM_ERR A system error occurred.
PAM_BUF_ERR A memory error occurred.

Related Information
"pam_start Subroutine" on page 960

pam_get_data Subroutine

Purpose
Retrieves information for a specific PAM module for this PAM session.

Library
PAM Library (libpam.a)

Syntax
#include <security/pam_appl.h>

int pam_get_data(PAMHandle, ModuleDataName, Data)

Parameters
PAMHandle (in) The PAM handle representing the current user authentication session. This handle is obtained by a call to pam_start().
ModuleDataName A unique identifier for Data.
Data Returned reference to the data denoted by ModuleDataName.

Return Values
Upon successful completion, pam_get_data returns PAM_SUCCESS. If ModuleDataName exists and pam_get_data completes successfully, Data will be a valid reference. Otherwise, Data will be NULL. If the routine fails, either PAM_SYSTEM_ERR, PAM_BUF_ERR, or PAM_NO_MODULE_DATA is returned, depending on the actual error.
Error Codes

- **PAM_SYSTEM_ERR**: A system error occurred.
- **PAM_BUF_ERR**: A memory error occurred.
- **PAM_NO_MODULE_DATA**: No module-specific data was found.

Related Information

- "pam_get_item Subroutine," "pam_getenv Subroutine" on page 945
- "pam_getenvlist Subroutine" on page 946
- "pam_set_data Subroutine" on page 949

pam_get_item Subroutine

Purpose

Retrieves an item or information for this PAM session.

Library

PAM Library (libpam.a)

Syntax

```c
#include <security/pam_appl.h>

int pam_get_item (PAMHandle *PAMHandle, ItemType *ItemType, Item **Item);
```

Description

The `pam_get_item` subroutine returns the item requested by the `ItemType`. Any items returned by `pam_get_item` should not be modified or freed. They can be later used by PAM and will be cleaned-up by `pam_end()`. If a requested `ItemType` is not found, a NULL reference will be returned in `Item`.

Parameters

- **PAMHandle**: The PAM handle representing the current user authentication session. This handle is obtained by a call to `pam_start()`.
ItemType

The type of item that is being requested. The following values are valid item types:

- **PAM_SERVICE**
  - The service name requesting this PAM session.
- **PAM_USER**
  - The user name of the user being authenticated.
- **PAM_AUTHTOK**
  - The user’s current authentication token (password).
- **PAM_OLDAUTHOK**
  - The user’s old authentication token (old password).
- **PAM_TTY**
  - The terminal name.
- **PAM_RHOST**
  - The name of the remote host.
- **PAM_RUSER**
  - The name of the remote user.
- **PAM_CONV**
  - The `pam_conv` structure for conversing with the user.
- **PAM_USER_PROMPT**
  - The default prompt for the user (used by `pam_get_user()`).

For security, **PAM_AUTHTOK** and **PAM_OLDAUTHOK** are only available to PAM modules.

Item

The return value, holding a reference to a pointer of the requested ItemType.

Return Values

Upon successful completion, `pam_get_item` returns PAM_SUCCESS. Also, the address of a reference to the requested object is returned in Item. If the requested item was not found, a NULL reference is returned. If the routine fails, either PAM_SYSTEM_ERR or PAM_BUF_ERR is returned and Item is set to a NULL pointer.

Error Codes

<table>
<thead>
<tr>
<th>Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>PAM_SYSTEM_ERR</td>
<td>A system error occurred.</td>
</tr>
<tr>
<td>PAM_BUF_ERR</td>
<td>A memory error occurred.</td>
</tr>
<tr>
<td>PAM_SYMBOL_ERR</td>
<td>Symbol not found.</td>
</tr>
</tbody>
</table>

Related Information

- "pam_get_data Subroutine" on page 942
- "pam_getenv Subroutine" on page 945
- "pam_get_user Subroutine" on page 945
- "pam_getenvlist Subroutine" on page 946
- "pam_set_item Subroutine" on page 950

---

**pam_get_user Subroutine**

**Purpose**

Gets the user’s name from the PAM handle or through prompting for input.

**Library**

PAM Library (`libpam.a`)
Syntax

```c
#include <security/pam Appl.h>

int pam_get_user(PAMHandle, User, PromptMsg);

int PAMHandle;
int User;
void **PromptMsg;
```

Description

The `pam_get_user` subroutine returns the user name currently stored in the PAM handle, `PAMHandle`. If the user name has not already been set through `pam_start()` or `pam_set_item()`, the subroutine displays the string specified by `PromptMsg`, to prompt for the user name through the conversation function. If `PromptMsg` is NULL, the value of `PAM_USER_PROMPT` set through a call to `pam_set_item()` is used. If both `PromptMsg` and `PAM_USER_PROMPT` are NULL, PAM defaults to use the following string:

```
Please enter user name:
```

After the user name has been retrieved, it is set in the PAM handle and is also returned to the caller in the `User` argument. The caller should not change or free `User`, as cleanup will be handled by `pam_end()`.

Parameters

- **PAMhandle**
  - The PAM handle representing the current user authentication session. This handle is obtained by a call to `pam_start()`.
- **User**
  - The user name retrieved from the PAM handle or provided by the user.
- **PromptMsg**
  - The prompt to be displayed if a user name is required and has not been already set.

Return Values

Upon successful completion, `pam_get_user` returns `PAM_SUCCESS`. Also, a reference to the user name is returned in `User`. If the routine fails, either `PAM_SYSTEM_ERR`, `PAM_BUF_ERR`, or `PAM_CONV_ERR` is returned, depending on what the actual error was, and a NULL reference in `User` is returned.

Error Codes

- **PAM_SYSTEM_ERR**: A system error occurred.
- **PAM_BUF_ERR**: A memory error occurred.
- **PAM_CONV_ERR**: A conversation error or failure.

Related Information

- [“pam_end Subroutine” on page 941](#)
- [“pam_get_item Subroutine” on page 943](#)
- [“pam_set_item Subroutine” on page 950](#)

---

**pam_getenv Subroutine**

Purpose

Returns the value of a defined PAM environment variable.

Library

PAM Library (libpam.a)
# Syntax

```c
#include <security/pam_appl.h>

char *pam_getenv(PAMHandle, VarName);
pam_handle_t *PAMHandle;
char *VarName;
```

## Description

The **pam_getenv** subroutine retrieves the value of the PAM environment variable *VarName* stored in the PAM handle *PAMHandle*. Environment variables can be defined through the `pam_putenv()` call. If *VarName* is defined, its value is returned in memory allocated by the library; it is the caller’s responsibility to free this memory. Otherwise, a NULL pointer is returned.

### Parameters

- **PAMHandle**
  - The PAM handle representing the current user authentication session. This handle is obtained by a call to `pam_start()`.

- **VarName**
  - The name of the PAM environment variable to get the value for.

### Return Values

Upon successful completion, **pam_getenv** returns the value of the *VarName* PAM environment variable. If the routine fails or *VarName* is not defined, NULL is returned.

### Related Information

- "**pam_getenvlist Subroutine,**" "**pam_putenv Subroutine**" on page 948

---

## pam_getenvlist Subroutine

### Purpose

Returns a list of all of the defined PAM environment variables and their values.

### Library

PAM Library (*libpam.a*)

### Syntax

```c
#include <security/pam_appl.h>

char **pam_getenvlist(PAMHandle);
pam_handle_t *PAMHandle;
```

### Description

The **pam_getenvlist** subroutine returns a pointer to a list of the currently defined environment variables in the PAM handle, *PAMHandle*. Environment variables can be set through calls to the `pam_putenv()` subroutine. The library returns the environment in an allocated array in which the last entry of the array is NULL. The caller is responsible for freeing the memory of the returned list.

### Parameters

- **PAMHandle**
  - The PAM handle representing the current user authentication session. This handle is obtained by a call to `pam_start()`.
Return Values
Upon successful completion, `pam_getenvlist` returns a pointer to a list of strings, one for each currently defined PAM environment variable. Each string is of the form `VARIABLE=VALUE`, where `VARIABLE` is the name of the variable and `VALUE` is its value. This list is terminated with a NULL entry. If the routine fails or there are no PAM environment variables defined, a NULL reference is returned. The caller is responsible for freeing the memory of the returned value.

Related Information
"pam_getenv Subroutine" on page 945, "pam_putenv Subroutine" on page 948

pam_open_session Subroutine

Purpose
Opens a new PAM user session.

Library
PAM Library (libpam.a)

Syntax
```
#include <security/pam_appl.h>

int pam_open_session (PAMHandle, Flags);

pam_handle_t *PAMHandle;
int Flags;
```

Description
The `pam_open_session` subroutine opens a new user session for an authenticated PAM user. A call to `pam_authenticate()` is typically made prior to invoking this subroutine. Applications that open a user session should subsequently close the session with `pam_close_session()` when the session has ended.

Parameters

- **PAMhandle**
  The PAM handle representing the current user authentication session. This handle is obtained by a call to `pam_start()`.

- **Flags**
  The flags are used to set pam_acct_mgmt options. The recognized flags are:
  - `PAM_SILENT`
    - No messages should be displayed

Return Values
Upon successful completion, `pam_open_session` returns `PAM_SUCCESS`. If the routine fails, a different error is returned, depending on the actual error.

Error Codes

- **PAM_SESSION_ERR**
  An error occurred while creating/removing an entry for the new session.

- **PAM_USER_UNKNOWN**
  The user is not known.

- **PAM_OPEN_ERR**
  One of the PAM authentication modules could not be loaded.
PAM_SYMBOL_ERR  A necessary item is not available to a PAM module.
PAM_SERVICE_ERR  An error occurred in a PAM module.
PAM_SYSTEM_ERR  A system error occurred.
PAM_BUF_ERR  A memory error occurred.
PAM_CONV_ERR  A conversation error occurred.
PAM_PERM_DENIED  Access permission was denied to the user.

Related Information
“pam_authenticate Subroutine” on page 937, “pam_close_session Subroutine” on page 940, “pam_sm_open_session Subroutine” on page 958, “pam_start Subroutine” on page 960

pam_putenv Subroutine

Purpose
Defines a PAM environment variable.

Library
PAM Library (libpam.a)

Syntax
#include <security/pam_appl.h>

int pam_putenv (PAMHandle, NameValue);

pam_handle_t *PAMHandle;
const char *NameValue;

Description
The pam_putenv subroutine sets and deletes environment variables in the PAM handle, PAMHandle. Applications can retrieve the defined variables by calling pam_getenv() or pam_getenvlist() and add them to the user’s session. If a variable with the same name is already defined, the old value is replaced by the new value.

Parameters
PAMHandle  The PAM authentication handle, obtained from a previous call to pam_start().
NameValue  A string of the form name=value to be stored in the environment section of the PAM handle. The following behavior is exhibited with regards to the format of the passed-in string:

NAME=VALUE  Creates or overwrites the value for the variable in the environment.
NAME=  Sets the variable to the empty string.
NAME  Deletes the variable from the environment, if it is currently defined.

Return Values
Upon successful completion, pam_putenv returns PAM_SUCCESS. If the routine fails, either PAM_SYSTEM_ERR or PAM_BUF_ERR is returned, depending on the actual error.
Error Codes

PAM_SYSTEM_ERR  A system error occurred.
PAM_BUF_ERR    A memory error occurred.

Related Information
“pam_getenv Subroutine” on page 945, “pam_getenvlist Subroutine” on page 946, “pam_start Subroutine” on page 960

pam_set_data Subroutine

Purpose
Sets information for a specific PAM module for the active PAM session.

Library
PAM Library (libpam.a)

Syntax
#include <security/pam_appl.h>

int pam_set_data (PAMHandle PAMHandle, ModuleDataName ModuleDataName, Data *Data, *cleanup(pam_handle_t *pamh, void *data, int pam_end_status))

Parameters

PAMHandle  The PAM handle representing the current user authentication session. This handle is obtained by a call to pam_start().

ModuleDataName  A unique identifier for Data.

Data  A reference to the data denoted by ModuleDataName.

cleanup  A function pointer that is called by pam_end() to clean up all allocated memory used by Data.

Description
The pam_set_data subroutine allows for the setting and updating of module-specific data within the PAM handle, PAMHandle. The ModuleDataName argument serves to uniquely identify the data, Data. Stored information can be retrieved by specifying ModuleDataName and passing it, along with the appropriate PAM handle, to pam_get_data(). The cleanup argument is a pointer to a function that is called to free allocated memory used by the Data when pam_end() is invoked. If data is already associated with ModuleDataName, PAM does a cleanup of the old data, overwrites it with Data, and replaces the old cleanup function. If the information being set is of a known PAM item type, use the pam_putenv subroutine instead.

Parameters

PAMHandle

ModuleDataName

Data

cleanup

Return Values
Upon successful completion, pam_set_data_ returns PAM_SUCCESS. If the routine fails, either PAM_SYSTEM_ERR or PAM_BUF_ERR is returned, depending on the actual error.
Error Codes

PAM_SYSTEM_ERR A system error occurred.
PAM_BUF_ERR A memory error occurred.

Related Information

"pam_end Subroutine" on page 941, "pam_get_data Subroutine" on page 942, "pam_get_item Subroutine" on page 943, "pam_set_item Subroutine"

pam_set_item Subroutine

Purpose

Sets the value of an item for this PAM session.

Library

PAM Library (libpam.a)

Syntax

#include <security/pam_appl.h>

int pam_set_item (PAMHandle, ItemType, Item)

pam_handle_t *PAMHandle;
int ItemType;
void **Item;

Description

The pam_set_item subroutine allows for the setting and updating of a set of known PAM items. The item value is stored within the PAM handle, PAMHandle. If a previous value exists for the item type, ItemType, then the old value is overwritten with the new value, Item.

Parameters

PAMHandle The PAM handle representing the current user authentication session. This handle is obtained by a call to pam_start().
**ItemType**

The type of item that is being requested. The following values are valid item types:

- **PAM_SERVICE**
  - The service name requesting this PAM session.
- **PAM_USER**
  - The user name of the user being authenticated.
- **PAM_AUTHTOK**
  - The user’s current authentication token. Interpreted as the new authentication token by password modules.
- **PAM_OLDAUTHTOK**
  - The user’s old authentication token. Interpreted as the current authentication token by password modules.
- **PAM_TTY**
  - The terminal name.
- **PAM_RHOST**
  - The name of the remote host.
- **PAM_RUSER**
  - The name of the remote user.
- **PAM_CONV**
  - The pam_conv structure for conversing with the user.
- **PAM_USER_PROMPT**
  - The default prompt for the user (used by pam_get_user()).

For security, **PAM_AUTHTOK** and **PAM_OLDAUTHTOK** are only available to PAM modules.

**Item**
The value that the **ItemType** is set to.

**Return Values**

Upon successful completion, **pam_set_item** returns **PAM_SUCCESS**. If the routine fails, either **PAM_SYSTEM_ERR** or **PAM_BUF_ERR** is returned, depending on what the actual error was.

**Error Codes**

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>PAM_SYSTEM_ERR</td>
<td>A system error occurred.</td>
</tr>
<tr>
<td>PAM_BUF_ERR</td>
<td>A memory error occurred.</td>
</tr>
<tr>
<td>PAM_SYMBOL_ERR</td>
<td>Symbol not found.</td>
</tr>
</tbody>
</table>

**Related Information**

“pam_get_item Subroutine” on page 943, “pam_get_user Subroutine” on page 944

---

**pam_setcred Subroutine**

**Purpose**

Establishes, changes, or removes user credentials for authentication.

**Library**

PAM Library (**libpam.a**)
Syntax

```c
#include <security/pam_appl.h>

int pam_setcred (PAMHandle, Flags);

```

Description
The `pam_setcred` subroutine allows for the credentials of the PAM user for the current PAM session to be modified. Functions such as establishing, deleting, renewing, and refreshing credentials are defined.

Parameters

- **PAMhandle**
  The PAM handle representing the current user authentication session. This handle is obtained by a call to `pam_start()`.

- **Flags**
  The flags are used to set `pam_setcred` options. The recognized flags are:

  - **PAM_SILENT**
    - No messages should be displayed.

  - **PAM_ESTABLISH_CRED**
    - Sets the user’s credentials. This is the default.

  - **PAM_DELETE_CRED**
    - Removes the user credentials.

  - **PAM_REINITIALIZE_CRED**
    - Renews the user credentials.

  - **PAM_REFRESH_CRED**
    - Refresh the user credentials, extending their lifetime.

*Mutually exclusive but may be logically OR’d with `PAM_SILENT`. If one of them is not set, `PAM_ESTABLISH_CRED` is assumed.

Return Values
Upon successful completion, `pam_setcred` returns `PAM_SUCCESS`. If the routine fails, a different error is returned, depending on the actual error.

Error Codes

- **PAM_CRED_UNAVAIL**
  The user credentials cannot be found.

- **PAM_CRED_EXPIRED**
  The user’s credentials have expired.

- **PAM_CRED_ERR**
  A failure occurred while setting user credentials.

- **PAM_USER_UNKNOWN**
  The user is not known.

- **PAM_OPEN_ERR**
  One of the PAM authentication modules could not be loaded.

- **PAM_SYMBOL_ERR**
  A necessary item is not available to a PAM module.

- **PAM_SERVICE_ERR**
  An error occurred in a PAM module.

- **PAM_SYSTEM_ERR**
  A system error occurred.

- **PAM_BUF_ERR**
  A memory error occurred.

- **PAM_CONV_ERR**
  A conversation error occurred.

- **PAM_PERM_DENIED**
  Access permission was denied to the user.
pam_sm_acct_mgmt Subroutine

Purpose
PAM module implementation for pam_acct_mgmt().

Library
PAM Library (libpam.a)

Syntax
```c
#include <security/pam_appl.h>
#include <security/pam_modules.h>

int pam_sm_acct_mgmt (PAMHandle, Flags, Argc, Argv)

{PAMHandle, Flags, Argc, Argv}

```  

Description
The pam_sm_acct_mgmt subroutine is invoked by the PAM library in response to a call to pam_acct_mgmt. The pam_sm_acct_mgmt subroutine performs the account and password validation for a user and is associated with the "account" service in the PAM configuration file. It is up to the module writers to implement their own service-dependent version of pam_sm_acct_mgmt, if the module requires this feature. Actual checks performed are at the discretion of the module writer but typically include checks such as password expiration and login time validation.

Parameters
- **PAMhandle**
  The PAM handle representing the current user authentication session. This handle is obtained by a call to pam_start().
- **Flags**
  The Flags argument can be a logically OR'd combination of the following:
  - **PAM_SILENT**
    - No messages should be displayed.
  - **PAM_DISALLOW_NULL_AUTHTOK**
    - Do not authenticate a user with a NULL authentication token.
- **Argc**
  The number of module options specified in the PAM configuration file.
- **Argv**
  The module options specified in the PAM configuration file. These options are module-dependent. Any modules receiving invalid options should ignore them.

Return Values
Upon successful completion, pam_sm_acct_mgmt returns PAM_SUCCESS. If the routine fails, a different error is returned, depending on the actual error.

Error Codes
- **PAM_ACCT_EXPIRED**
  The user's account has expired.
The user’s password needs to be changed. This is usually due to password aging or because it was last set by the system administrator. At this stage, most users can still change their passwords. Applications should call `pam_chauthtok()` and have the users change their password.

The user’s password has expired. Unlike `PAM_NEW_AUTHTOKEN_REQD`, the password cannot be changed by the user.

The user is not known.

One of the PAM authentication modules could not be loaded.

A necessary item is not available to a PAM module.

An error occurred in a PAM module.

A system error occurred.

A memory error occurred.

A conversation error occurred.

Access permission was denied to the user.

Related Information

“pam_acct_mgmt Subroutine” on page 936, “pam_authenticate Subroutine” on page 937, “pam_start Subroutine” on page 960

pam_sm_authenticate Subroutine

Purpose

PAM module-specific implementation of `pam_authenticate()`.

Library

PAM Library (`libpam.a`)

Syntax

```c
#include <security/pam_appl.h>
#include <security/pam_modules.h>

int pam_sm_authenticate(PAMHandle, Flags, Argc,_argv);
```

Description

When an application invokes `pam_authenticate()`, the PAM Framework calls `pam_sm_authenticate` for each module in the authentication module stack. This allows all the PAM module authors to implement their own authenticate routine. `pam_authenticate` and `pam_sm_authenticate` provide an authentication service to verify that the user is allowed access.

Parameters

- **PAMhandle**: The PAM handle representing the current user authentication session. This handle is obtained by a call to `pam_start()`. 
Flags

The flags are used to set `pam_acct_mgmt` options. The recognized flags are:

- **PAM_SILENT**
  - No messages should be displayed.
- **PAM_DISALLOW_NULL_AUTHTOK**
  - Do not authenticate a user with a NULL authentication token.

Argc

The number of module options defined.

Argv

The module options. These options are module-dependent. Any modules receiving invalid options should ignore them.

Return Values

Upon successful completion, `pam_sm_authenticate` returns `PAM_SUCCESS`. If the routine fails, a different error is returned, depending on the actual error.

Error Codes

- **PAM_AUTH_ERR**
  - An error occurred in authentication, usually because of an invalid authentication token.
- **PAM_CRED_INSUFFICIENT**
  - The user has insufficient credentials to access the authentication data.
- **PAM_AUTHINFO_UNAVAIL**
  - The authentication information cannot be retrieved.
- **PAM_USER_UNKNOWN**
  - The user is not known.
- **PAM_MAXTRIES**
  - The maximum number of authentication retries has been reached.
- **PAM_OPEN_ERR**
  - One of the PAM authentication modules could not be loaded.
- **PAM_SYMBOL_ERR**
  - A necessary item is not available to a PAM module.
- **PAM_SERVICE_ERR**
  - An error occurred in a PAM module.
- **PAM_SYSTEM_ERR**
  - A system error occurred.
- **PAM_BUF_ERR**
  - A memory error occurred.
- **PAM_CONV_ERR**
  - A conversation error occurred.
- **PAM_PERM_DENIED**
  - Access permission was denied to the user.

Related Information

[pam_authenticate Subroutine” on page 937](#)

pam_sm_chauthtok Subroutine

Purpose

PAM module-specific implementation of `pam_chauthtok()`.

Library

PAM Library (`libpam.a`)

Syntax

```c
#include <security/pam_appl.h>
#include <security/pam_modules.h>

int pam_sm_chauthtok (PAMHandle, Flags, Argc, Argv)
    pam_handle_t *PAMHandle;
```
int Flags;
int argc;
const char **argv;

Description
When an application invokes pam_chauthtok(), the PAM Framework calls pam_sm_chauthtok for each module in the password module stack. The pam_sm_chauthtok module interface is intended to change the user’s password or authentication token. Before any password is changed, pam_sm_chauthtok performs preliminary tests to ensure necessary hosts and information, depending on the password service, are there. If PAM_PRELIM_CHECK is specified, only these preliminary checks are done. If successful, the authentication token is ready to be changed. If the PAM_UPDATE_AUTHTOK flag is passed in, pam_sm_chauthtok should take the next step and change the user’s authentication token. If the PAM_CHANGE_EXPIRED_AUTHTOK flag is set, the module should check the authentication token for aging and expiration. If the user’s authentication token is aged or expired, the module should store that information by passing it to pam_set_data(). Otherwise, the module should exit and return PAM_IGNORE. Required information is obtained through the PAM handle or by prompting the user by way of pam_conv.

Parameters

PAMhandle
The PAM handle representing the current user authentication session. This handle is obtained by a call to pam_start().

Flags
The flags are used to set pam_acct_mgm options. The recognized flags are:

- PAM_SILENT
  - No messages should be displayed.

- PAM_CHANGE_EXPIRED_AUTHTOK
  - Only expired passwords should be changed. If this flag is not included, all users using the related password service are forced to update their passwords.

- PAM_PRELIM_CHECK*
  - Only perform preliminary checks to see if the password can be changed, but do not change it.

- PAM_UPDATE_AUTHTOK*
  - Perform all necessary checks, and if possible, change the user’s password/ authentication token.

* PAM_PRELIM_CHECK and PAM_UPDATE_AUTHTOK are mutually exclusive.

Argc
The number of module options defined.

Argv
The module options. These options are module-dependent. Any modules receiving invalid options should ignore them.

Return Values
Upon successful completion, pam_sm_chauthtok returns PAM_SUCCESS. If the routine fails, a different error is returned, depending on the actual error.

Error Codes

PAM_AUTHTOK_ERR
A failure occurred while updating the authentication token.

PAM_TRY_AGAIN
Preliminary checks for changing the password have failed. Try again later.

PAM_AUTHTOK_RECOVERY_ERR
An error occurred while trying to recover the authentication information.
PAM_AUTHTOK_LOCK_BUSY

Cannot get the authentication token lock. Try again later.

PAM_AUTHTOK_DISABLE_AGING

Authentication token aging checks are disabled and were not performed.

PAM_USER_UNKNOWN

The user is not known.

PAM_OPEN_ERR

One of the PAM authentication modules could not be loaded.

PAM_SYMBOL_ERR

A necessary item is not available to a PAM module.

PAM_SERVICE_ERR

An error occurred in a PAM module.

PAM_SYSTEM_ERR

A system error occurred.

PAM_BUF_ERR

A memory error occurred.

PAM_CONV_ERR

A conversation error occurred.

PAM_PERM_DENIED

Access permission was denied to the user.

Related Information

"pam_chauthtok Subroutine" on page 939

pam_sm_close_session Subroutine

Purpose

PAM module-specific implementation to close a session previously opened by `pam_sm_open_session()`.

Library

PAM Library (libpam.a)

Syntax

```c
#include <security/pam_appl.h>
#include <security/pam_modules.h>

int pam_sm_close_session (PAMHandle, Flags, Argc, Argv)
    pam_handle_t *PAMHandle;
    int Flags;
    int Argc;
    const char **Argv;
```

Description

When an application invokes `pam_close_session()`, the PAM Framework calls `pam_sm_close_session` for each module in the session module stack. The `pam_sm_close_session` module interface is intended to clean up and terminate any user session started by `pam_sm_open_session()`.

Parameters

- **PAMhandle**
  - The PAM handle representing the current user authentication session. This handle is obtained by a call to `pam_start()`.
- **Flags**
  - The flags are used to set `pam_acct_mgmt` options. The recognized flag is:
    - PAM_SILENT
      - No messages should be displayed.
- **Argc**
  - The number of module options defined.
- **Argv**
  - The module options. These options are module-dependent. Any modules receiving invalid options should ignore them.
Return Values
Upon successful completion, `pam_sm_close_session` returns `PAM_SUCCESS`. If the routine fails, a different error is returned, depending on the actual error.

Error Codes

<table>
<thead>
<tr>
<th>Error Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>PAM_SESSION_ERR</td>
<td>An error occurred while creating or removing an entry for the new session.</td>
</tr>
<tr>
<td>PAM_USER_UNKNOWN</td>
<td>The user is not known.</td>
</tr>
<tr>
<td>PAM_OPEN_ERR</td>
<td>One of the PAM authentication modules could not be loaded.</td>
</tr>
<tr>
<td>PAM_SYMBOL_ERR</td>
<td>A necessary item is not available to a PAM module.</td>
</tr>
<tr>
<td>PAM_SERVICE_ERR</td>
<td>An error occurred in a PAM module.</td>
</tr>
<tr>
<td>PAM_SYSTEM_ERR</td>
<td>A system error occurred.</td>
</tr>
<tr>
<td>PAM_BUF_ERR</td>
<td>A memory error occurred.</td>
</tr>
<tr>
<td>PAM_CONV_ERR</td>
<td>A conversation error occurred.</td>
</tr>
<tr>
<td>PAM_PERM_DENIED</td>
<td>Access permission was denied to the user.</td>
</tr>
</tbody>
</table>

Related Information

- "pam_close_session Subroutine" on page 940
- "pam_sm_open_session Subroutine"

pam_sm_open_session Subroutine

Purpose
PAM module-specific implementation of `pam_open_session`.

Library
PAM Library (`libpam.a`)

Syntax

```c
#include <security/pam_appl.h>
#include <security/pam_modules.h>

int pam_sm_open_session(PAMHandle PAMHandle, Flags Flags, Argc Argv, Argv)

pam_handle_t *PAMHandle;
int Flags;
int Argc;
const char **Argv;
```

Description
When an application invokes `pam_open_session()`, the PAM Framework calls `pam_sm_open_session` for each module in the session module stack. The `pam_sm_open_session` module interface starts a new user session for an authenticated PAM user. All session-specific information and memory used by opening a session should be cleaned up by `pam_sm_close_session()`.

Parameters

- **PAMhandle**
  The PAM handle representing the current user authentication session. This handle is obtained by a call to `pam_start()`.
**Flags**
The flags are used to set `pam_acct_mgmt` options. The recognized flag is:

- **PAM_SILENT**
  - No messages should be displayed.

**Argc**
The number of module options defined.

**Argv**
The module options. These options are module-dependent. Any modules receiving invalid options should ignore them.

## Return Values
Upon successful completion, `pam_sm_open_session` returns `PAM_SUCCESS`. If the routine fails, a different error is returned, depending on the actual error.

## Error Codes
<table>
<thead>
<tr>
<th>Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>PAM_SESSION_ERR</td>
<td>An error occurred while creating or removing an entry for the new session.</td>
</tr>
<tr>
<td>PAM_USER_UNKNOWN</td>
<td>The user is not known.</td>
</tr>
<tr>
<td>PAM_OPEN_ERR</td>
<td>One of the PAM authentication modules could not be loaded.</td>
</tr>
<tr>
<td>PAM_SYMBOL_ERR</td>
<td>A necessary item is not available to a PAM module.</td>
</tr>
<tr>
<td>PAM_SERVICE_ERR</td>
<td>An error occurred in a PAM module.</td>
</tr>
<tr>
<td>PAM_SYSTEM_ERR</td>
<td>A system error occurred.</td>
</tr>
<tr>
<td>PAM_BUF_ERR</td>
<td>A memory error occurred.</td>
</tr>
<tr>
<td>PAM_CONV_ERR</td>
<td>A conversation error occurred.</td>
</tr>
<tr>
<td>PAM_PERM_DENIED</td>
<td>Access permission was denied to the user.</td>
</tr>
</tbody>
</table>

## Related Information
- ["pam_open_session Subroutine" on page 947](#)
- ["pam_sm_close_session Subroutine" on page 957](#)

### pam_sm_setcred Subroutine

#### Purpose
PAM module-specific implementation of `pam_setcred`.

#### Library
PAM Library (libpam.a)

#### Syntax
```c
#include <security/pam_appl.h>
#include <security/pam_modules.h>

int pam_sm_setcred(PAMHandle, Flags, Argc, Argv)

pam_handle_t *PAMHandle;
int Flags;
int Argc;
const char **Argv;
```

#### Description
When an application invokes `pam_setcred()`, the PAM Framework calls `pam_sm_setcred` for each module in the authentication module stack. The `pam_sm_setcred` module interface allows for the setting of module-specific credentials in the PAM handle. The user’s credentials should be set based upon the user’s authentication state. This information can usually be retrieved with a call to `pam_get_data()`. 
Parameters

PAMhandle
The PAM handle representing the current user authentication session. This handle is obtained by a call to pam_start().

Flags
The flags are used to set pam_setcred options. The recognized flags are:

- PAM_SILENT
  - No messages should be displayed.
- PAM_ESTABLISH_CRED*
  - Sets the user’s credentials. This is the default.
- PAM_DELETE_CRED*
  - Removes the user credentials.
- PAM_REINITIALIZE_CRED*
  - Renews the user credentials.
- PAM_REFRESH_CRED*
  - Refreshes the user credentials, extending their lifetime.

*Mutually exclusive. If one of them is not set, PAM_ESTABLISH_CRED is assumed.

Argc
The number of module options defined.

Argv
The module options. These options are module-dependent. Any modules receiving invalid options should ignore them.

Return Values
Upon successful completion, pam_sm_setcred returns PAM_SUCCESS. If the routine fails, a different error is returned, depending on the actual error.

Error Codes

- PAM_CRED_UNAVAIL
  - The user credentials cannot be found.
- PAM_CRED_EXPIRED
  - The user’s credentials have expired.
- PAM_CRED_ERR
  - A failure occurred while setting user credentials.
- PAM_USER_UNKNOWN
  - The user is not known.
- PAM_OPEN_ERR
  - One of the PAM authentication modules could not be loaded.
- PAM_SYMBOL_ERR
  - A necessary item is not available to a PAM module.
- PAM_SERVICE_ERR
  - An error occurred in a PAM module.
- PAM_SYSTEM_ERR
  - A system error occurred.
- PAM_BUF_ERR
  - A memory error occurred.
- PAM_CONV_ERR
  - A conversation error occurred.
- PAM_PERM_DENIED
  - Access permission was denied to the user.

Related Information

“pam_setcred Subroutine” on page 951

pam_start Subroutine

Purpose
Initiates a new PAM user authentication session.

Library
PAM Library (libpam.a)
Syntax

```
#include <security/pam_appl.h>

int pam_start (Service, User, Conversation, PAMHandle)
const char *Service;
const char *User;
const struct pam_conv *Conversation;
pam_handle_t **PAMHandle;
```

Description

The `pam_start` subroutine begins a new PAM session for authentication within one of the four realms of the PAM environment [authentication, account, session, password]. This routine is called only at the start of the session, not at the start of each module comprising the session. The PAM handle, `PAMHandle`, returned by this subroutine is subsequently used by other PAM routines. The handle must be cleaned up at the end of use, which can easily be done by passing it as an argument to `pam_end`.

Parameters

- **Service**: The name of the service initiating this PAM session.
- **User**: The user who is being authenticated.
Conversation

The PAM conversation struct enabling communication with the user. This structure, \texttt{pam\_conv}, consists of a pointer to a conversation function, as well as a pointer to application data.

\begin{verbatim}
struct pam_conv {
    int (**conv)();
    void (**appdata_ptr);
}
\end{verbatim}

The argument \texttt{conv} is defined as:

\begin{verbatim}
int conv( int num_msg, const struct pam_message **msg,
         const struct pam_response **resp, void *appdata );
\end{verbatim}

The conversation function, \texttt{conv}, allows PAM to send messages to, and get input from, a user. The arguments to the function have the following definition and behavior:

\begin{itemize}
  \item \texttt{num\_msg} The number of lines of messages to be displayed (all messages are returned in one-line fragments, each no longer than \texttt{PAM\_MAX\_MSG\_SIZE} characters and with no more lines than \texttt{PAM\_MAX\_NUM\_MSG})
  \item \texttt{msg} Contains the message text and its style.
    \begin{verbatim}
    struct pam_message {
        int style;  /* Message style */
        char *msg;   /* The message */
    }
    \end{verbatim}
    The message style, can be one of:
    \begin{itemize}
      \item \texttt{PAM\_PROMPT\_ECHO\_OFF} Prompts users with message and does not echo their responses; it is typically for use with requesting passwords and other sensitive information.
      \item \texttt{PAM\_PROMPT\_ECHO\_ON} Prompts users with message and echoes their responses back to them.
      \item \texttt{PAM\_ERROR\_MSG} Displays message as an error message.
      \item \texttt{PAM\_TEXT\_INFO} Displays general information, such as authentication failures.
    \end{itemize}
  \item \texttt{resp} Holds the user's response and a response code.
    \begin{verbatim}
    struct pam_response {
        char **resp;  /* Reference to the response */
        int resp_retcode; /* Not used, should be 0 */
    }
    \end{verbatim}
  \item \texttt{appdata, appdata\_ptr} Pointers to the application data that can be passed by the calling application to the PAM modules. Use these to allow PAM to send data back to the application.
\end{itemize}

\texttt{PAM\_Handle} The PAM handle representing the current user authentication session is returned upon successful completion.

Return Values

Upon successful completion, \texttt{pam\_start} returns \texttt{PAM\_SUCCESS}, and a reference to the pointer of a valid PAM handle is returned through \texttt{PAM\_Handle}. If the routine fails, a value different from \texttt{PAM\_SUCCESS} is returned, and the \texttt{PAM\_Handle} reference is NULL.
Error Codes

PAM_SERVICE_ERR: An error occurred in a PAM module.
PAM_SYSTEM_ERR: A system error occurred.
PAM_BUF_ERR: A memory error occurred.

Related Information
“pam_end Subroutine” on page 941, “pam_set_data Subroutine” on page 949, “pam_set_item Subroutine” on page 950

pam_strerror Subroutine

Purpose
Translates a PAM error code to a string message.

Library
PAM Library (libpam.a)

Syntax
#include <security/pam_appl.h>

const char *pam_strerror (PAMHandle, ErrorCode)
pam_handle_t *PAMHandle;
int ErrorCode;

Description
The pam_strerror subroutine uses the error number returned by the PAM routines and returns the PAM error message that is associated with that error number. If the error number is not known to pam_strerror, or there is no translation error message, then NULL is returned. The caller should not free or modify the returned string.

Parameters

PAMhandle: The PAM handle representing the current user authentication session. This handle is obtained by a call to pam_start().
ErrorCode: The PAM error code for which the PAM error message is to be retrieved.

Return Values
Upon successful completion, pam_strerror returns the PAM error message corresponding to the PAM error code, ErrorCode. A NULL pointer is returned if the routine fails, the error code is not known, or no error message exists for that error code.

passwdexpired Subroutine

Purpose
Checks the user’s password to determine if it has expired.
Syntax

```c
passwdexpired (UserName, Message);
char *UserName;
char **Message;
```

Description

The `passwdexpired` subroutine checks a user’s password to determine if it has expired. The subroutine checks the `registry` variable in the `/etc/security/user` file to ascertain where the user is administered. If the `registry` variable is not defined, the `passwdexpired` subroutine checks the local, NIS, and DCE databases for the user definition and expiration time.

The `passwdexpired` subroutine may pass back informational messages, such as how many days remain until password expiration.

Parameters

- **UserName**
  Specifies the user's name whose password is to be checked.
- **Message**
  Points to a pointer that the `passwdexpired` subroutine allocates memory for and fills in. This string is suitable for printing and issues messages, such as in how many days the password will expire.

Return Values

Upon successful completion, the `passwdexpired` subroutine returns a value of 0. If this subroutine fails, it returns one of the following values:

- **1** Indicates that the password is expired, and the user must change it.
- **2** Indicates that the password is expired, and only a system administrator may change it.
- **-1** Indicates that an internal error has occurred, such as a memory allocation (malloc) failure or database corruption.

Error Codes

The `passwdexpired` subroutine fails if one or more of the following values is true:

- **ENOENT** Indicates that the user could not be found.
- **EACCES** Indicates that the user did not have permission to check password expiration.
- **ENOMEM** Indicates that memory allocation (malloc) failed.
- **EINVAL** Indicates that the parameters are not valid.

Related Information

The [authenticate Subroutine](#) subroutine.

The `login` command.

**passwdexpiredx Subroutine**

**Purpose**

Checks the user’s password to determine if it has expired, in multiple methods.
Syntax

```c
passwdexpiredx (UserName, Message, State)
```

**Description**

The `passwdexpiredx` subroutine checks a user's password to determine if it has expired. The subroutine uses the user’s `SYSTEM` attribute to ascertain which administrative domains are used for password authentication.

The `passwdexpiredx` subroutine can pass back informational messages, such as how many days remain until password expiration.

The `State` parameter can contain information about the results of the authentication process. The `State` parameter from an earlier call to the `authenticatex` subroutine can be used to control how password expiration checking is performed. Authentication mechanisms that were not used to authenticate a user are not examined for expired passwords. The `State` parameter must be initialized to reference a null pointer if the `State` parameter from an earlier call to the `authenticatex` subroutine is not used.

**Parameters**

- **UserName**
  Specifies the user’s name whose password is to be checked.
- **Message**
  Points to a pointer that the `passwdexpiredx` subroutine allocates memory for and fills in. This string is suitable for printing, and it issues messages, such as an alert that indicates how many days are left before the password expires.
- **State**
  Points to a pointer that the `passwdexpiredx` subroutine allocates memory for and fills in. The `State` parameter can also be the result of an earlier call to the `authenticatex` subroutine. The `State` parameter contains information about the results of the password expiration examination process for each term in the user’s `SYSTEM` attribute. The calling application is responsible for freeing this memory when it is no longer needed for a subsequent call to the `chpassx` subroutine.

**Return Values**

Upon successful completion, the `passwdexpiredx` subroutine returns a value of 0. If this subroutine fails, it returns one of the following values:

- `-1`
  Indicates that an internal error has occurred, such as a memory allocation (malloc) failure or database corruption.
- `1`
  Indicates that one or more passwords are expired, and the user must change it. None of the expired passwords require system administrator intervention to be changed.
- `2`
  Indicates that one or more passwords are expired, at least one of which must be changed by the user and at least one of which requires system administrator intervention to be changed.
- `3`
  Indicates that all expired passwords require system administrator intervention to be changed.

**Error Codes**

The `passwdexpiredx` subroutine fails if one or more of the following values is true:

- **EACCESS**
  The user did not have permission to access the password attributes required to check password expiration.
- **EINVAL**
  The parameters are not valid.
- **ENOENT**
  The user could not be found.
- **ENOMEM**
  Memory allocation (malloc) failed.
 Related Information

The “authenticate” Subroutine on page 115.

The login Command

passwdpolicy Subroutine

Purpose
Supports password strength policies on a per-user or per-named-policy basis.

Syntax

#include <pwdpolicy.h>

int passwdpolicy

(const char * name, int type,
 const char * old_password,
 const char * new_password,
 time64_t last_update);

Description

The passwdpolicy subroutine supports application use of password strength policies on a per-user or per-named-policy basis. The policies that are supported include password dictionaries, history list length, history list expiration, maximum lifetime of a password, minimum period of time between permitted password changes, maximum period after which an expired password can be changed, maximum number of repeated characters in a password, minimum number of alphabetic characters in a password, minimum number of nonalphabetic characters in a password, minimum length of a password, and a list of loadable modules that can be used to determine additional password strength rules.

The type parameter allows an application to select where the policy values are located. Privileged process can use either PWP_USERNAME or PWP_SYSTEMPOLICY. Unprivileged processes are limited to using PWP_LOCALPOLICY.

The following named attributes are used for each test:

dictionlist A SEC_LIST value that gives a list of dictionaries to be checked. If new_password is found in any of the named dictionaries, this test fails. If this test fails, the return value contains the PWP_IN_DICTIONARY bit.

histsize A SEC_INT value giving the permissible size of the named user’s password history. The named user’s password history is obtained by calling getuseridattr with the S_HISTLIST attribute. If this attribute does not exist, password history checks are disabled. A value of 0 disables password history tests. If this test fails, the return value contains the PWP_REUSE_PASSWORD bit.

histexpire A SEC_INT value that gives the number of weeks that must elapse before a password in the named user’s password history list can be reused. If this test fails the return value contains the PWP_REUSE_PASSWORD_TOO_SOON bit.

maxage A SEC_INT value that gives the number of weeks a password can be considered valid. A password that has not been modified more recently than maxage weeks is considered to have expired and is subject to the maxexpired test. A value less than or equal to 0 disables this test. This attribute is used to determine if maxexpired must be tested, and it does not generate a return value.

minage A SEC_INT value that gives the number of weeks before a password can be changed. A password that has been modified more recently than minage weeks fails this test. A value less than or equal to 0 disables this test. If this test fails, the return value contains the PWP_TOO_SOON bit.

maxexpired A SEC_INT value that gives the number of weeks after which an expired password cannot be changed. A value of 0 indicates that an expired password cannot be changed. A value of -1 indicates that an expired password can be changed after any length of time. If this test fails, the return value contains the PWP_EXPIRED bit.
maxrepeats A SEC_INT value that gives the maximum number of times any single character can appear in the new password. A value less than or equal to 0 disables this test. If this test fails, the return value contains the PWP_TOO_MANY_REPEATS bit.

mindiff A SEC_INT value that gives the maximum number of characters in the new password that must not be present in the old password. A value less than or equal to 0 disables this test. If this test fails, the return value contains the PWP_TOO_MANY_SAME bit.

minalpha A SEC_INT value that gives the minimum number of alphabetic characters that must be present in the password. A value less than or equal to 0 disables this test. If this test fails, the return value contains the PWP_TOO_FEW_ALPHA bit.

minother A SEC_INT value that gives the minimum number of nonalphabetic characters that must be present in the password. A value less than or equal to 0 disables this test. If this test fails, the return value contains the PWP_TOO_FEW_OTHER bit.

minlen A SEC_INT value that gives the minimum required length of a password. There is no maximum value for this attribute. A value less than or equal to 0 disables this test. If this test fails, the return value contains the PWP_TOO_SHORT bit.

pwdchecks A SEC_LIST value that gives a list of named loadable modules that must be executed to validate the password. If this test fails, the return value contains the PWP_FAILED_OTHER bit.

Parameters

name The name of either a specific user or named policy. User names have policy information determined by invoking the getuserattr subroutine. Policy names have policy information determined by invoking the getconfattr subroutine.

type One of three values:

PWP_USERNAME Policy values for PWP_USERNAME are stored in /etc/security/user. Password tests PWP_REUSED_PW and PWP_REUSED_TOO_SOON are only enabled for this value.

PWP_SYSTEMPOLICY Policy values for PWP_SYSTEMPOLICY are stored in /etc/security/passwd_policy.

PWP_LOCALPOLICY Policy values for PWP_LOCALPOLICY are stored in /usr/lib/security/passwd_policy.

old_password The current value of the password. This function does not verify that old_password is the correct current password. Invoking passwdpolicy with a NULL pointer for this parameter disables PWP_TOO_MANYSAME tests.

new_password The value of the new password. Invoking passwdpolicy with a NULL pointer for this parameter disables all tests except password age tests.

last_update The time the password was last changed, as a time64_t value, expressed in seconds since the UNIX epoch. A 0 value for this parameter disables password age tests regardless of the value of any other parameter.

Return Values

The return value is a bit-mapped representation of the tests that failed. A return value of 0 indicates that all password rules passed. A value of -1 indicates that some other error, other than a failed password test, has occurred. The errno variable indicates the cause of that error. Applications must compare a non-zero return value against -1 before checking any specific bits in the return value.

Files

The /usr/include/pwdpolicy.h header file.
Related Information

“passwdexpired Subroutine” on page 963, “passwdstrength Subroutine”

passwdstrength Subroutine

Purpose
Performs basic password age and construction tests.

Syntax

```c
#include <pwdpolicy.h>
int passwdstrength (const char *old_password, const char *new_password,
                    time64_t last_update, passwd_policy_t *policy, int checks);
```

Description

The passwdstrength subroutine performs basic password age and construction tests. Password history, reuse, and dictionary tests are not performed. The values contained in the policy parameter are used to validate the value of new_password.

The following fields are used by the passwdstrength subroutine.

- **pwp_version**: Specifies the version of the passwd_policy_t structure. The current structure version number is PWP_VERSION_1.
- **pwp_minage**: The number of seconds, as a time32_t, between the time a password is modified and the time the password can again be modified. This field is referenced if PWP_TOO_SOON is set in checks.
- **pwp_maxage**: The number of seconds, as a time32_t, after which a password that has been modified is considered to be expired. This field is referenced if PWP_EXPIRED is set in checks.
- **pwp_maxexpired**: The number of seconds, as a time32_t, since a password has expired after which it can no longer be modified. A value of 0 indicates that an expired password cannot be changed. A value of -1 indicates that an expired password can be changed after any length of time. This field is referenced if PWP_EXPIRED is set in checks.
- **pwp_minalpha**: The minimum number of characters in the password that must be alphabetic characters, as determined by invoking the isalpha() macro. A value less than or equal to 0 disables this test. This field is referenced if PWP_TOO_FEW_ALPHA is set in checks.
- **pwp_minother**: The minimum number of characters in the password that cannot be alphabetic characters, as determined by invoking the isalpha() macro. A value less than or equal to 0 disables this test. This field is referenced if PWP_TOO_FEW_OTHER is set in checks.
- **pwp_minlen**: The minimum total number of characters in the password. A value less than or equal to 0 disables this test. This field is referenced if PWP_TOO_SHORT is set in checks.
- **pwp_maxrepeats**: The maximum number of times an individual character can appear in the password. A value less than or equal to 0 disables this test. This field is referenced if PWP_TOO_MANY_REPEATS is set in checks.
- **pwp_mindiff**: The minimum number of characters that must be changed between the old password and the new password. A value less than or equal to 0 disables this test. This field is referenced if PWP_TOO_MANY_SAME is set in checks.

Parameters

- **old_password**: The value of the current password. This parameter must be non-NULL if PWP_TOO_MANY_SAME is set in checks or the results are undefined.
new_password: The value of the new password. This parameter must be non-NULL if any of PWP_TOO_SHORT, PWP_TOO_FEW_ALPHA, PWP_TOO_FEW_OTHER, PWP_TOO_MANY_SAME, or PWP_TOO_MANY_REPEATS are set in checks or the results are undefined.

last_update: The time the password was last changed, as a time64_t value, expressed in seconds since the UNIX epoch. A 0 value for this parameter indicates that the password has never been set. This might cause PWP_EXPIRED to be set in the return value if it is set in checks.

policy: A pointer to a passwd_policy_t containing the values for the password policy attributes.

checks: A bitmask value that indicates the set of password tests to be performed. The return value contains only those bits that are defined in checks.

Return Values
The return value is a bit-mapped representation of the tests that failed. A return value of 0 indicates that all password rules requested in the checks parameter passed. A value of -1 indicates that some other error, other than a password test, has occurred. The errno variable indicates the cause of that error. Applications must compare a non-zero return value against -1 before checking any specific bits in the return value.

Files
The /usr/include/pwdpolicy.h header file.

Related Information
“passwdexpired Subroutine” on page 963, “passwdpolicy Subroutine” on page 966

pathconf or fpathconf Subroutine

Purpose
Retrieves file-implementation characteristics.

Library
Standard C Library (libc.a)

Syntax
#include <unistd.h>

long pathconf (Path, Name);
const char *Path;
int Name;

long fpathconf(FileDescriptor, Name)
int FileDescriptor, Name;

Description
The pathconf subroutine allows an application to determine the characteristics of operations supported by the file system contained by the file named by the Path parameter. Read, write, or execute permission of the named file is not required, but all directories in the path leading to the file must be searchable.

The fpathconf subroutine allows an application to retrieve the same information for an open file.
Parameters

Path Specifies the path name.

FileDescriptor Specifies an open file descriptor.

Name Specifies the configuration attribute to be queried. If this attribute is not applicable to the file specified by the Path or FileDescriptor parameter, the pathconf subroutine returns an error. Symbolic values for the Name parameter are defined in the unistd.h file:

_PC_LINK_MAX Specifies the maximum number of links to the file.

_PC_MAX_CANON Specifies the maximum number of bytes in a canonical input line. This value is applicable only to terminal devices.

_PC_MAX_INPUT Specifies the maximum number of bytes allowed in an input queue. This value is applicable only to terminal devices.

_PC_NAME_MAX Specifies the maximum number of bytes in a file name, not including a terminating null character. This number can range from 14 through 255. This value is applicable only to a directory file.

_PC_PATH_MAX Specifies the maximum number of bytes in a path name, including a terminating null character.

_PC_PIPE_BUF Specifies the maximum number of bytes guaranteed to be written atomically. This value is applicable only to a first-in-first-out (FIFO).

_PC_CHOWN_RESTRICTED Returns 0 if the use of the chown subroutine is restricted to a process with appropriate privileges, and if the chown subroutine is restricted to changing the group ID of a file only to the effective group ID of the process or to one of its supplementary group IDs.

_PC_NO_TRUNC Returns 0 if long component names are truncated. This value is applicable only to a directory file.

_PC_VDISABLE This is always 0. No disabling character is defined. This value is applicable only to a terminal device.

_PC_AIX_DISK_PARTITION Determines the physical partition size of the disk.

Note: The _PC_AIX_DISK_PARTITION variable is available only to the root user.

_PC_AIX_DISK_SIZE Determines the disk size in megabytes.

Note: The _PC_AIX_DISK_SIZE variable is available only to the root user.

Note: The _PC_FILESIZEBITS and PC_SYNC_IO flags apply to AIX 4.3 and later releases.

_PC_FILESIZEBITS Returns the minimum number of bits required to hold the file system's maximum file size as a signed integer. The smallest value returned is 32.

_PC_SYNC_IO Returns -1 if the file system does not support the Synchronized Input and Output option. Any value other than -1 is returned if the file system supports the option.
Notes:
1. If the Name parameter has a value of _PC_LINK_MAX, and if the Path or FileDescriptor parameter refers to a directory, the value returned applies to the directory itself.
2. If the Name parameter has a value of _PC_NAME_MAX or _PC_NO_TRUNC, and if the Path or FileDescriptor parameter refers to a directory, the value returned applies to filenames within the directory.
3. If the Name parameter has a value if _PC_PATH_MAX, and if the Path or FileDescriptor parameter refers to a directory that is the working directory, the value returned is the maximum length of a relative pathname.
4. If the Name parameter has a value of _PC_PIPE_BUF, and if the Path parameter refers to a FIFO special file or the FileDescriptor parameter refers to a pipe or a FIFO special file, the value returned applies to the referenced object. If the Path or FileDescriptor parameter refers to a directory, the value returned applies to any FIFO special file that exists or can be created within the directory.
5. If the Name parameter has a value of _PC_CHOWN_RESTRICTED, and if the Path or FileDescriptor parameter refers to a directory, the value returned applies to any files, other than directories, that exist or can be created within the directory.

Return Values
If the pathconf or fpathconf subroutine is successful, the specified parameter is returned. Otherwise, a value of -1 is returned and the errno global variable is set to indicate the error. If the variable corresponding to the Name parameter has no limit for the Path parameter or the FileDescriptor parameter, both the pathconf and fpathconf subroutines return a value of -1 without changing the errno global variable.

Error Codes
The pathconf or fpathconf subroutine fails if the following error occurs:

EINVAL The name parameter specifies an unknown or inapplicable characteristic.

The pathconf subroutine can also fail if any of the following errors occur:

EACCES Search permission is denied for a component of the path prefix.
EINVAL The implementation does not support an association of the Name parameter with the specified file.
ENAMETOOLONG The length of the Path parameter string exceeds the PATH_MAX value.
ENAMETOOLONG Pathname resolution of a symbolic link produced an intermediate result whose length exceeds PATH_MAX.
ENOENT The named file does not exist or the Path parameter points to an empty string.
ENOTDIR A component of the path prefix is not a directory.
ELOOP Too many symbolic links were encountered in resolving path.

The fpathconf subroutine can fail if either of the following errors occur:

EBADF The File Descriptor parameter is not valid.
EINVAL The implementation does not support an association of the Name parameter with the specified file.

Related Information
The “chown, fchown, lchown, chownx, or fchownx Subroutine” on page 151, “confstr Subroutine” on page 181, sysconf subroutine.

Files, Directories, and File Systems for Programmers Subroutines Overview in AIX 5L Version 5.3 General Programming Concepts: Writing and Debugging Programs.
pause Subroutine

Purpose
Suspends a process until a signal is received.

Library
Standard C Library (libc.a)

Syntax
#include <unistd.h>
int pause (void)

Description
The pause subroutine suspends the calling process until it receives a signal. The signal must not be one that is ignored by the calling process. The pause subroutine does not affect the action taken upon the receipt of a signal.

Return Values
If the signal received causes the calling process to end, the pause subroutine does not return.

If the signal is caught by the calling process and control is returned from the signal-catching function, the calling process resumes execution from the point of suspension. The pause subroutine returns a value of -1 and sets the errno global variable to EINTR.

Related Information
The incinterval, alarm, or settimer subroutine, getinterval, incinterval, absinterval, resinc, resabs, alarm, ualarm, sigset timer Subroutine subroutine, kill or kil lpg subroutine, sigaction, sigvec, or signal subroutine, wait, waitpid, or wait3 subroutine.

pcap_close Subroutine

Purpose
Closes the open files related to the packet capture descriptor and frees the memory used by the packet capture descriptor.

Library
pcap Library (libpcap.a)

Syntax
#include <pcap.h>

void pcap_close(pcap_t * p);

Description
The pcap_close subroutine closes the files associated with the packet capture descriptor and deallocates resources. If the pcap_open_offline subroutine was previously called, the pcap_close subroutine closes the savefile, a previously saved packet capture data file. Or the pcap_close subroutine closes the packet capture device if the pcap_open_live subroutine was previously called.
Parameters

\( p \)

Points to a packet capture descriptor as returned by the \texttt{pcap_open_live} or the \texttt{pcap_open_offline} subroutine.

Related Information

The \texttt{pcap_open_live} \texttt{("pcap_open_live Subroutine" on page 985)} subroutine, \texttt{pcap_open_offline} \texttt{("pcap_open_offline Subroutine" on page 986)} subroutine.

\textbf{pcap_compile Subroutine}

\textbf{Purpose}

Compiles a filter expression into a filter program.

\textbf{Library}

pcap Library (libpcap.a)

\textbf{Syntax}

\begin{verbatim}
#include <pcap.h>

int pcap_compile(pcap_t *p, struct bpf_program *fp, char *str, int optimize, bpf_u_int32 netmask);
\end{verbatim}

\textbf{Description}

The \texttt{pcap_compile} subroutine is used to compile the string \texttt{str} into a filter program. This filter program will then be used to filter, or select, the desired packets.

\textbf{Parameters}

- \texttt{netmask}
  - Specifies the \texttt{netmask} of the network device. The \texttt{netmask} can be obtained from the \texttt{pcap_lookupnet} subroutine.

- \texttt{optimize}
  - Controls whether optimization on the resulting code is performed.

- \texttt{p}
  - Points to a packet capture descriptor returned from the \texttt{pcap_open_offline} or the \texttt{pcap_open_live} subroutine.

- \texttt{program}
  - Points to a \texttt{bpf_program} struct which will be filled in by the \texttt{pcap_compile} subroutine if the subroutine is successful.

- \texttt{str}
  - Contains the filter expression.

\textbf{Return Values}

Upon successful completion, the \texttt{pcap_compile} subroutine returns 0, and the program parameter will hold the filter program. If \texttt{pcap_compile} subroutine is unsuccessful, -1 is returned.

Related Information

The \texttt{pcap_geterr} \texttt{("pcap_geterr Subroutine" on page 979)} subroutine, \texttt{pcap_lookupnet} \texttt{("pcap_lookupnet Subroutine" on page 981)} subroutine, \texttt{pcap_open_live} \texttt{("pcap_open_live Subroutine" on page 985)} subroutine, \texttt{pcap_open_offline} \texttt{("pcap_open_offline Subroutine" on page 986)} subroutine, \texttt{pcap_perror} \texttt{("pcap_perror Subroutine" on page 987)} subroutine, \texttt{pcap_setfilter} \texttt{("pcap_setfilter Subroutine" on page 988)} subroutine.
pcap_datalink Subroutine

Purpose
Obtains the link layer type for the packet capture device.

Library
pcap Library (libpcap.a)

Syntax
```
#include <pcap.h>

int pcap_datalink(pcap_t *p);
```

Description
The `pcap_datalink` subroutine returns the link layer type of the packet capture device, for example, IFT_ETHER. This is useful in determining the size of the datalink header at the beginning of each packet that is read.

Parameters

\p
Points to the packet capture descriptor as returned by the \pcap_open_live or the \pcap_open_offline subroutine.

Return Values
The `pcap_datalink` subroutine returns the link layer type.

Note: Only call this subroutine after successful calls to either the \pcap_open_live or the \pcap_open_offline subroutine. Never call the `pcap_datalink` subroutine after a call to \pcap_close as unpredictable results will occur.

Related Information
The `pcap_close` subroutine, `pcap_open_live` subroutine, `pcap_open_offline` subroutine.

pcap_dispatch Subroutine

Purpose
Collects and processes packets.

Library
pcap Library (libpcap.a)

Syntax
```
#include <pcap.h>

int pcap_dispatch(pcap_t *p, int cnt, pcap_handler callback, u_char *user);
```

Related Information
The `pcap_close` subroutine, `pcap_open_live` subroutine, `pcap_open_offline` subroutine.
Description

The `pcap_dispatch` subroutine reads and processes packets. This subroutine can be called to read and process packets that are stored in a previously saved packet capture data file, known as the `savefile`. The subroutine can also read and process packets that are being captured live.

Notice that the third parameter, `callback`, is of the type `pcap_handler`. This is a pointer to a user-provided subroutine with three parameters. Define this user-provided subroutine as follows:

```c
void user_routine(u_char *user, struct pcap_pkthdr *phdr, u_char *pdata)
```

The parameter, `user`, is the `user` parameter that is passed into the `pcap_dispatch` subroutine. The parameter, `phdr`, is a pointer to the `pcap_pkthdr` structure which precedes each packet in the `savefile`. The parameter, `pdata`, points to the packet data. This allows users to define their own handling of packet capture data.

Parameters

- **callback**: Points to a user-provided routine that will be called for each packet read. The user is responsible for providing a valid pointer, and that unpredictable results can occur if an invalid pointer is supplied.

  **Note**: The `pcap_dump` subroutine can also be specified as the `callback` parameter. If this is done, the `pcap_dump_open` subroutine should be called first. The pointer to the `pcap_dumper_t` struct returned from the `pcap_dump_open` subroutine should be used as the `user` parameter to the `pcap_dispatch` subroutine. The following program fragment illustrates this use:

  ```c
  pcap_dumper_t *pd;
  pcap_t *p;
  int rc = 0;
  
  pd = pcap_dump_open(p, "/tmp/savefile");
  
  rc = pcap_dispatch(p, 0, pcap_dump, (u_char *) pd);
  ```

- **cnt**: Specifies the maximum number of packets to process before returning. A `cnt` of -1 processes all the packets received in one buffer. A `cnt` of 0 processes all packets until an error occurs, EOF is reached, or the read times out (when doing live reads and a non-zero read timeout is specified).

- **p**: Points to a packet capture descriptor returned from the `pcap_open_offline` or the `pcap_open_live` subroutine. This will be used to store packet data that is read in.

- **user**: Specifies the first argument to pass into the `callback` routine.

Return Values

Upon successful completion, the `pcap_dispatch` subroutine returns the number of packets read. If EOF is reached in a `savefile`, zero is returned. If the `pcap_dispatch` subroutine is unsuccessful, -1 is returned. In this case, the `pcap_geterr` or `pcap_perror` subroutine can be used to get the error text.

Related Information

The `pcap_dump` subroutine, `pcap_dump_close` subroutine, `pcap_dump_open` subroutine, `pcap_geterr` subroutine, `pcap_open_live` subroutine, `pcap_open_offline` subroutine, `pcap_perror` subroutine.
pcap_dump Subroutine

Purpose
Writes packet capture data to a binary file.

Library
pcap Library (libpcap.a)

Syntax
#include <pcap.h>

void pcap_dump(u_char *user, struct pcap_pkthdr *h, u_char *sp);

Description
The pcap_dump subroutine writes the packet capture data to a binary file. The packet header data, contained in h, will be written to the file pointed to by the user file pointer, followed by the packet data from sp. Up to h->caplen bytes of sp will be written.

The file that user points to (where the pcap_dump subroutine writes to) must be open. To open the file and retrieve its pointer, use the pcap_dump_open subroutine.

The calling arguments for the pcap_dump subroutine are suitable for use with pcap_dispatch subroutine and the pcap_loop subroutine. To retrieve this data, the pcap_open_offline subroutine can be invoked with the name of the file that user points to as its first parameter.

Parameters

h
Contains the packet header data that will be written to the packet capture data file, known as the savefile. This data will be written ahead of the rest of the packet data.

sp
Points to the packet data that is to be written to the savefile.

user
Specifies the savefile file pointer which is returned from the pcap_dump_open subroutine. It should be cast to a u_char * when passed in.

Related Information
The pcap_dispatch subroutine, pcap_dump_close subroutine, pcap_dump_open subroutine, pcap_loop subroutine, pcap_open_live subroutine, pcap_open_offline subroutine.

pcap_dump_close Subroutine

Purpose
Closes a packet capture data file, know as a savefile.

Library
pcap Library (libpcap.a)
Syntax

```c
#include <pcap.h>

void pcap_dump_close(pcap_dumper_t *p);
```

Description

The `pcap_dump_close` subroutine closes a packet capture data file, known as the `savefile`, that was opened using the `pcap_dump_open` subroutine.

Parameters

- `p` Points to a `pcap_dumper_t`, which is synonymous with a `FILE *`, the file pointer of a `savefile`.

Related Information

The `pcap_dump_open` subroutine.

---

**pcap_dump_open Subroutine**

Purpose

 Opens or creates a file for writing packet capture data.

Library

`pcap` Library (libpcap.a)

Syntax

```c
#include <pcap.h>

pcap_dumper_t *pcap_dump_open(pcap_t *p, char *fname);
```

Description

The `pcap_dump_open` subroutine opens or creates the packet capture data file, known as the `savefile`. This action is specified through the `fname` parameter. The subroutine then writes the required packet capture file header to the file. The `pcap_dump` subroutine can then be called to write the packet capture data associated with the packet capture descriptor, `p`, into this file. The `pcap_dump_open` subroutine must be called before calling the `pcap_dump` subroutine.

Parameters

- `fname` Specifies the name of the file to open. A "-" indicates that standard output should be used instead of a file.
- `p` Specifies a packet capture descriptor returned by the `pcap_open_offline` or the `pcap_open_live` subroutine.

Return Values

Upon successful completion, the `pcap_dump_open` subroutine returns a pointer to a the file that was opened or created. This pointer is a pointer to a `pcap_dumper_t`, which is synonymous with `FILE *`. See the `pcap_dump` subroutine or the `pcap_loop` subroutine for an example of how to
use pcap_dumper_t. If the pcap_dump_open subroutine is unsuccessful, Null is returned. Use the pcap_geterr subroutine to obtain the specific error text.

Related Information
The pcap_dispatch subroutine, pcap_dump subroutine, pcap_dump_close subroutine, pcap_geterr subroutine, pcap_loop subroutine, pcap_open_live subroutine, pcap_open_offline subroutine.

pcap_file Subroutine

Purpose
Obtains the file pointer to the savefile, a previously saved packed capture data file.

Library
pcap Library (libpcap.a)

Syntax
#include <pcap.h>

FILE *pcap_file(pcap_t *p);

Description
The pcap_file subroutine returns the file pointer to the savefile. If there is no open savefile, 0 is returned. This subroutine should be called after a successful call to the pcap_open_offline subroutine and before any calls to the pcap_close subroutine.

Parameters

p Points to a packet capture descriptor as returned by the pcap_open_offline subroutine.

Return Values
The pcap_file subroutine returns the file pointer to the savefile.

Related Information
The pcap_close subroutine, pcap_open_offline subroutine.

pcap_fileno Subroutine

Purpose
Obtains the descriptor for the packet capture device.

Library
pcap Library (libpcap.a)
Syntax
#include <pcap.h>

int pcap_fileno(pcap_t *p);

Description
The `pcap_fileno` subroutine returns the descriptor for the packet capture device. This subroutine should be called only after a successful call to the `pcap_open_live` subroutine and before any calls to the `pcap_close` subroutine.

Parameters

`p` Points to a packet capture descriptor as returned by the `pcap_open_live` subroutine.

Return Values
The `pcap_fileno` subroutine returns the descriptor for the packet capture device.

Related Information
The `pcap_close` subroutine, `pcap_open_live` subroutine.

pcap_geterr Subroutine

Purpose
Obtains the most recent pcap error message.

Library
pcap Library (libpcap.a)

Syntax
#include <pcap.h>

char *pcap_geterr(pcap_t *p);

Description
The `pcap_geterr` subroutine returns the error text pertaining to the last pcap library error. This subroutine is useful in obtaining error text from those subroutines that do not return an error string. Since the pointer returned points to a memory space that will be reused by the pcap library subroutines, it is important to copy this message into a new buffer if the error text needs to be saved.

Parameters

`p` Points to a packet capture descriptor as returned by the `pcap_open_live` or the `pcap_open_offline` subroutine.

Return Values
The `pcap_geterr` subroutine returns a pointer to the most recent error message from a pcap library subroutine. If there were no previous error messages, a string with 0 as the first byte is returned.
Related Information
The `pcap_open_live` subroutine, `pcap_open_offline` subroutine, `pcap_perror` subroutine, `pcap_strerror` subroutine.

pcap_is_swapped Subroutine

Purpose
Reports if the byte order of the previously saved packet capture data file, known as the `savefile` was swapped.

Library
`pcap` Library (`libpcap.a`)

Syntax
```c
#include <pcap.h>

int pcap_is_swapped(pcap_t *p);
```

Description
The `pcap_is_swapped` subroutine returns 1 (True) if the current `savefile` uses a different byte order than the current system. This subroutine should be called after a successful call to the `pcap_open_offline` subroutine and before any calls to the `pcap_close` subroutine.

Parameters

`p`
Points to a packet capture descriptor as returned from the `pcap_open_offline` subroutine.

Return Values

1
If the byte order of the `savefile` is different from that of the current system.

0
If the byte order of the `savefile` is the same as that of the current system.

Related Information
The `pcap_close` subroutine, `pcap_open_offline` subroutine.

pcap_lookupdev Subroutine

Purpose
Obtains the name of a network device on the system.

Library
`pcap` Library (`libpcap.a`)
**Syntax**

```
#include <pcap.h>

char *pcap_lookupdev(char *errbuf);
```

**Description**

The `pcap_lookupdev` subroutine gets a network device suitable for use with the `pcap_open_live` and the `pcap_lookupnet` subroutines. If no interface can be found, or none are configured to be up, `Null` is returned. In the case of multiple network devices attached to the system, the `pcap_lookupdev` subroutine returns the first one it finds to be up, other than the loopback interface. (Loopback is always ignored.)

**Parameters**

- **errbuf**
  
  Returns error text and is only set when the `pcap_lookupdev` subroutine fails.

**Return Values**

Upon successful completion, the `pcap_lookupdev` subroutine returns a pointer to the name of a network device attached to the system. If `pcap_lookupdev` subroutine is unsuccessful, `Null` is returned, and text indicating the specific error is written to `errbuf`.

**Related Information**

The `pcap_geterr` ([“pcap_geterr Subroutine” on page 979](#)) subroutine, `pcap_lookupnet` ([“pcap_lookupnet Subroutine” on page 980](#)) subroutine, `pcap_open_live` ([“pcap_open_live Subroutine” on page 985](#)) subroutine, `pcap_perror` ([“pcap_perror Subroutine” on page 987](#)) subroutine.

---

**pcap_lookupnet Subroutine**

**Purpose**

Returns the network address and subnet mask for a network device.

**Library**

pcap Library (`libpcap.a`)

**Syntax**

```
#include <pcap.h>

int pcap_lookupnet(char *device, bpf_u_int32 *netp, bpf_u_int32 *maskp, char *errbuf);
```

**Description**

Use the `pcap_lookupnet` subroutine to determine the network address and subnet mask for the network device, `device`.

**Parameters**

- **device**
  
  Specifies the name of the network device to use for the network lookup, for example, en0.

- **errbuf**
  
  Returns error text and is only set when the `pcap_lookupnet` subroutine fails.
maskp

Holds the subnet mask associated with device.

netp

Holds the network address for the device.

Return Values

Upon successful completion, the \texttt{pcap_lookupnet} subroutine returns 0. If the \texttt{pcap_lookupnet} subroutine is unsuccessful, -1 is returned, and \texttt{errbuf} is filled in with an appropriate error message.

Related Information

The \texttt{pcap_compile} ("pcap_compile Subroutine" on page 973) subroutine, \texttt{pcap_geterr} ("pcap_geterr Subroutine" on page 979) subroutine, \texttt{pcap_lookupdev} ("pcap_lookupdev Subroutine" on page 980) subroutine, \texttt{pcap_perror} ("pcap_perror Subroutine" on page 987) subroutine.

\textbf{pcap_loop Subroutine}

\textbf{Purpose}

Collects and processes packets.

\textbf{Library}

pcap Library (libpcap.a)

\textbf{Syntax}

\begin{verbatim}
#include <pcap.h>

int pcap_loop(pcap_t * p, int cnt, pcap_handler callback, u_char * user);
\end{verbatim}

\textbf{Description}

The \texttt{pcap_loop} subroutine reads and processes packets. This subroutine can be called to read and process packets that are stored in a previously saved packet capture data file, known as the \texttt{savefile}. The subroutine can also read and process packets that are being captured live.

This subroutine is similar to \texttt{pcap_dispatch} subroutine except it continues to read packets until \texttt{cnt} packets have been processed, EOF is reached (in the case of offline reading), or an error occurs. It does not return when live read timeouts occur. That is, specifying a non-zero read timeout to the \texttt{pcap_open_live} subroutine and then calling the \texttt{pcap_loop} subroutine allows the reception and processing of any packets that arrive when the timeout occurs.

Notice that the third parameter, \texttt{callback}, is of the type \texttt{pcap_handler}. This is a pointer to a user-provided subroutine with three parameters. Define this user-provided subroutine as follows:

\begin{verbatim}
void user_routine(u_char *user, struct pcap_pkthdr *phrd, u_char *pdata)
\end{verbatim}

The parameter, \texttt{user}, will be the user parameter that was passed into the \texttt{pcap_dispatch} subroutine. The parameter, \texttt{phdr}, is a pointer to the \texttt{pcap_pkthdr} structure, which precedes each packet in the \texttt{savefile}. The parameter, \texttt{pdata}, points to the packet data. This allows users to define their own handling of their filtered packets.
Parameters

*callback*

Points to a user-provided routine that will be called for each packet read. The user is responsible for providing a valid pointer, and that unpredictable results can occur if an invalid pointer is supplied.

**Note:** The `pcap_dump` subroutine can also be specified as the callback parameter. If this is done, call the `pcap_dump_open` subroutine first. Then use the pointer to the `pcap_dumper_t` struct returned from the `pcap_dump_open` subroutine as the user parameter to the `pcap_dispatch` subroutine. The following program fragment illustrates this use:

```c
pcap_dumper_t *pd
pcap_t *p;
int rc = 0;

pd = pcap_dump_open(p, "/tmp/savefile");
rc = pcap_dispatch(p, 0, pcap_dump, (u_char *)pd);
```

*cnt*

Specifies the maximum number of packets to process before returning. A negative value causes the `pcap_loop` subroutine to loop forever, or until EOF is reached or an error occurs. A `cnt` of 0 processes all packets until an error occurs or EOF is reached.

*p*

Points to a packet capture descriptor returned from the `pcap_open_offline` or the `pcap_open_live` subroutine. This will be used to store packet data that is read in.

*user*

Specifies the first argument to pass into the *callback* routine.

Return Values

Upon successful completion, the `pcap_loop` subroutine returns 0. 0 is also returned if EOF has been reached in a *savefile*. If the `pcap_loop` subroutine is unsuccessful, -1 is returned. In this case, the `pcap_geterr` subroutine or the `pcap_perror` subroutine can be used to get the error text.

Related Information

The `pcap_dispatch` subroutine, `pcap_dump` subroutine, `pcap_dump_close` subroutine, `pcap_dump_open` subroutine, `pcap_geterr` subroutine, `pcap_open_live` subroutine, `pcap_open_offline` subroutine, `pcap_perror` subroutine.

**pcap_major_version Subroutine**

**Purpose**

Obtains the major version number of the packet capture format used to write the *savefile*, a previously saved packet capture data file.

**Library**

pcap Library (libpcap.a)

**Syntax**

```c
#include <pcap.h>

int pcap_major_version(pcap_t *p);
```
Description
The \texttt{pcap\_major\_version} subroutine returns the major version number of the packet capture format used to write the \textit{savefile}. If there is no open \textit{savefile}, 0 is returned.

\textbf{Note:} This subroutine should be called only after a successful call to \texttt{pcap\_open\_offline} subroutine and before any calls to the \texttt{pcap\_close} subroutine.

\textbf{Parameters}
\begin{itemize}
  \item \textit{p} \hspace{1cm} Points to a packet capture descriptor as returned from \texttt{pcap\_open\_offline} subroutine.
\end{itemize}

\textbf{Return Values}
The major version number of the packet capture format used to write the \textit{savefile}.

\textbf{Related Information}
The \texttt{pcap\_close} ("\textit{pcap\_close Subroutine" on page 972) subroutine, \texttt{pcap\_open\_offline} ("\textit{pcap\_open\_offline Subroutine" on page 986) subroutine.

---

\textbf{pcap\_minor\_version Subroutine}

\textbf{Purpose}
Obtains the minor version number of the packet capture format used to write the \textit{savefile}.

\textbf{Library}
\texttt{pcap Library (libpcap.a)}

\textbf{Syntax}
\begin{verbatim}
#include <pcap.h>

int pcap_minor_version(pcap_t *p);
\end{verbatim}

\textbf{Description}
The \texttt{pcap\_minor\_version} subroutine returns the minor version number of the packet capture format used to write the \textit{savefile}. This subroutine should only be called after a successful call to the \texttt{pcap\_open\_offline} subroutine and before any calls to the \texttt{pcap\_close} subroutine.

\textbf{Parameters}
\begin{itemize}
  \item \textit{p} \hspace{1cm} Points to a packet capture descriptor as returned from the \texttt{pcap\_open\_offline} subroutine.
\end{itemize}

\textbf{Return Values}
The minor version number of the packet capture format used to write the \textit{savefile}.

\textbf{Related Information}
The \texttt{pcap\_close} ("\textit{pcap\_close Subroutine" on page 972) subroutine, \texttt{pcap\_open\_offline} ("\textit{pcap\_open\_offline Subroutine" on page 986) subroutine.
pcap_next Subroutine

Purpose
Obtains the next packet from the packet capture device.

Library
pcap Library (libpcap.a)

Syntax
#include <pcap.h>

u_char *pcap_next(pcap_t *p, struct pcap_pkthdr *h);

Description
The pcap_next subroutine returns a u_char pointer to the next packet from the packet capture device. The packet capture device can be a network device or a savefile that contains packet capture data. The data has the same format as used by tcpdump.

Parameters
h
Points to the packet header of the packet that is returned. This is filled in upon return by this routine.

p
Points to the packet capture descriptor to use as returned by the pcap_open_live or the pcap_open_offline subroutine.

Return Values
Upon successful completion, the pcap_next subroutine returns a pointer to a buffer containing the next packet and fills in the h, which points to the packet header of the returned packet. If the pcap_next subroutine is unsuccessful, Null is returned.

Related Information
The pcap_dispatch subroutine, pcap_dump subroutine, pcap_dump_close subroutine, pcap_dump_open subroutine, pcap_loop subroutine, pcap_open_live subroutine, pcap_open_offline subroutine, pcap_open_live Subroutine, pcap_open_offline Subroutine.

The tcpdump command.

pcap_open_live Subroutine

Purpose
Opens a network device for packet capture.

Library
pcap Library (libpcap.a)
#include <pcap.h>

pcap_t *pcap_open_live(char *device, int snaplen, int promisc, int to_ms, char *ebuf);

## Description

The `pcap_open_live` subroutine opens the specified network device for packet capture. The term "live" is to indicate that a network device is being opened, as opposed to a file that contains packet capture data. This subroutine must be called before any packet capturing can occur. All other routines dealing with packet capture require the packet capture descriptor that is created and initialized with this routine. See the `pcap_open_offline` subroutine for more details on opening a previously saved file that contains packet capture data.

### Parameters

- **device**
  Specifies a string that contains the name of the network device to open for packet capture, for example, en0.
- **ebuf**
  Returns error text and is only set when the `pcap_open_live` subroutine fails.
- **promisc**
  Specifies that the device is to be put into promiscuous mode. A value of 1 (True) turns promiscuous mode on. If this parameter is 0 (False), the device will remain unchanged. In this case, if it has already been set to promiscuous mode (for some other reason), it will remain in this mode.
- **snaplen**
  Specifies the maximum number of bytes to capture per packet.
- **to_ms**
  Specifies the read timeout in milliseconds.

### Return Values

Upon successful completion, the `pcap_open_live` subroutine will return a pointer to the packet capture descriptor that was created. If the `pcap_open_live` subroutine is unsuccessful, Null is returned, and text indicating the specific error is written into the `ebuf` buffer.

### Related Information

The `pcap_close` subroutine, `pcap_compile` subroutine, `pcap_datalink` subroutine, `pcap_dispatch` subroutine, `pcap_dump` subroutine, `pcap_dump_open` subroutine, `pcap_geterr` subroutine, `pcap_getfilter` subroutine, `pcap_perror` subroutine, `pcap_stats` subroutine, `pcap_setfilter` subroutine, `pcap_snapshot` subroutine.

## pcap_open_offline Subroutine

### Purpose

Opens a previously saved file containing packet capture data.
Library
pcap Library (libpcap.a)

Syntax
#include <pcap.h>

pcap_t *pcap_open_offline(char *fname, char *ebuf);

Description
The pcap_open_offline subroutine opens a previously saved packet capture data file, known as the savefile. This subroutine creates and initializes a packet capture (pcap) descriptor and opens the specified savefile containing the packet capture data for reading.

This subroutine should be called before any other related routines that require a packet capture descriptor for offline packet processing. See the pcap_open_live subroutine for more details on live packet capture.

Note: The format of the savefile is expected to be the same as the format used by the tcpdump command.

Parameters

- ebuf: Returns error text and is only set when the pcap_open_offline subroutine fails.

- fname: Specifies the name of the file to open. A hyphen (-) passed as the fname parameter indicates that stdin should be used as the file to open.

Return Values
Upon successful completion, the pcap_open_offline subroutine will return a pointer to the newly created packet capture descriptor. If the pcap_open_offline subroutine is unsuccessful, Null is returned, and text indicating the specific error is written into the ebuf buffer.

Related Information
The pcap_close subroutine, pcap_dispatch subroutine, pcap_file subroutine, pcap_fileno subroutine, pcap_geterr subroutine, pcap_is_swapped subroutine, pcap_loop subroutine, pcap_major_version subroutine, pcap_minor_version subroutine, pcap_next subroutine, pcap_open_live subroutine.

The tcpdump command.

pcap_perror Subroutine

Purpose
Prints the passed-in prefix, followed by the most recent error text.
Library
pcap Library (libpcap.a)

Syntax
#include <pcap.h>

void pcap_perror(pcap_t * p, char * prefix);

Description
The **pcap_perror** subroutine prints the text of the last pcap library error to stderr, prefixed by *prefix*. If there were no previous errors, only *prefix* is printed.

Parameters
- **p**: Points to a packet capture descriptor as returned by the **pcap_open_live** subroutine or the **pcap_open_offline** subroutine.
- **prefix**: Specifies the string that is to be printed before the stored error message.

Related Information
The **pcap_geterr** subroutine, **pcap_open_live** subroutine, **pcap_open_offline** subroutine, **pcap_strerror** subroutine.

pcap_setfilter Subroutine

Purpose
Loads a filter program into a packet capture device.

Library
pcap Library (libpcap.a)

Syntax
#include <pcap.h>

int pcap_setfilter(pcap_t * p, struct bpf_program * fp);

Description
The **pcap_setfilter** subroutine is used to load a filter program into the packet capture device. This causes the capture of the packets defined by the filter to begin.

Parameters
- **fp**: Points to a filter program as returned from the **pcap_compile** subroutine.
- **p**: Points to a packet capture descriptor returned from the **pcap_open_offline** or the **pcap_open_live** subroutine.
Return Values

Upon successful completion, the `pcap_setfilter` subroutine returns 0. If the `pcap_setfilter` subroutine is unsuccessful, -1 is returned. In this case, the `pcap_geterr` subroutine can be used to get the error text, and the `pcap_perror` subroutine can be used to display the text.

Related Information

The `pcap_compile` subroutine, `pcap_geterr` subroutine, `pcap_open_live` subroutine, `pcap_open_offline` subroutine, `pcap_perror` subroutine.

pcap_snapshot Subroutine

Purpose

Obtains the number of bytes that will be saved for each packet captured.

Library

pcap Library (libpcap.a)

Syntax

```c
#include <pcap.h>

int pcap_snapshot( pcap_t *p);
```

Description

The `pcap_snapshot` subroutine returns the snapshot length, which is the number of bytes to save for each packet captured.

Note: This subroutine should only be called after successful calls to either the `pcap_open_live` subroutine or `pcap_open_offline` subroutine. It should not be called after a call to the `pcap_close` subroutine.

Parameters

`p` Points to the packet capture descriptor as returned by the `pcap_open_live` or the `pcap_open_offline` subroutine.

Return Values

The `pcap_snapshot` subroutine returns the snapshot length.

Related Information

The `pcap_close` subroutine, `pcap_open_live` subroutine, `pcap_open_offline` subroutine.

pcap_stats Subroutine

Purpose

Obtains packet capture statistics.
Library
pcap Library (libpcap.a)

Syntax
#include <pcap.h>

int pcap_stats (pcap_t *p, struct pcap_stat *ps);

Description
The pcap_stats subroutine fills in a pcap_stat struct. The values represent packet statistics from the start of the run to the time of the call. Statistics for both packets that are received by the filter and packets that are dropped are stored inside a pcap_stat struct. This subroutine is for use when a packet capture device is opened using the pcap_open_live subroutine.

Parameters
p Points to a packet capture descriptor as returned by the pcap_open_live subroutine.
ps Points to the pcap_stat struct that will be filled in with the packet capture statistics.

Return Values
On successful completion, the pcap_stats subroutine fills in ps and returns 0. If the pcap_stats subroutine is unsuccessful, -1 is returned. In this case, the error text can be obtained with the pcap_perror subroutine or the pcap_geterr subroutine.

Related Information
The pcap_geterr subroutine, pcap_open_live subroutine, pcap_perror subroutine.

pcap_strerror Subroutine

Purpose
Obtains the error message indexed by error.

Library
pcap Library (libpcap.a)

Syntax
#include <pcap.h>

char *pcap_strerror(int error);

Description
Lookup the error message indexed by error. The possible values of error correspond to the values of the errno global variable. This function is equivalent to the strerror subroutine.
Parameters

error

Specifies the key to use in obtaining the corresponding error message. The error message is taken from the system's `sys_errlist`.

Return Values

The `pcap_strerror` subroutine returns the appropriate error message from the system error list.

Related Information

The `pcap_geterr` subroutine, `pcap_perror` subroutine, `strerror` subroutine.

pclose Subroutine

Purpose

Closes a pipe to a process.

Library

Standard C Library (`libc.a`)

Syntax

```c
#include <stdio.h>
int pclose (FILE *Stream);
```

Description

The `pclose` subroutine closes a pipe between the calling program and a shell command to be executed. Use the `pclose` subroutine to close any stream you opened with the `popen` subroutine. The `pclose` subroutine waits for the associated process to end, and then returns the exit status of the command.

Attention: If the original processes and the `popen` process are reading or writing a common file, neither the `popen` subroutine nor the `pclose` subroutine should use buffered I/O. If they do, the results are unpredictable.

Avoid problems with an output filter by flushing the buffer with the `fflush` subroutine.

Parameter

`Stream` Specifies the `FILE` pointer of an opened pipe.

Return Values

The `pclose` subroutine returns a value of -1 if the `Stream` parameter is not associated with a `popen` command or if the status of the child process could not be obtained. Otherwise, the value of the termination status of the command language interpreter is returned; this will be 127 if the command language interpreter cannot be executed.
**Error Codes**

If the application has:
- Called the **wait** subroutine,
- Called the **waitpid** subroutine with a process ID less than or equal to zero or equal to the process ID of the command line interpreter,
- Masked the SIGCHIL signal, or
- Called any other function that could perform one of the steps above, and one of these calls caused the termination status to be unavailable to the **pclose** subroutine, a value of -1 is returned and the **errno** global variable is set to ECHILD.

**Related Information**

The related **fclose** or **fflush** subroutine, **fopen**, **freopen**, or **fdopen** subroutine, **fopen**, **fopen64**, **freopen**, **freopen64** or **fdopen** subroutine, **pipe** subroutine, **popen** subroutine, **wait**, **waitpid**, or **wait3** subroutine.

**Files, Directories, and File Systems for Programmers** in AIX 5L Version 5.3 General Programming Concepts: Writing and Debugging Programs.

---

**perfstat_cpu Subroutine**

**Purpose**
Retrieves individual logical processor usage statistics.

**Library**
perfstat library (libperfstat.a)

**Syntax**

```c
#include <libperfstat.h>

int perfstat_cpu (name, userbuff, sizeof_struct, desired_number);
```

- `name`: `perfstat_id_t *`
- `userbuff`: `perfstat_cpu_t *`
- `sizeof_struct`: `size_t`
- `desired_number`: `int`

**Description**

The **perfstat_cpu** subroutine retrieves one or more individual processor usage statistics. The same function can be used to retrieve the number of available sets of logical processor statistics.

To get one or more sets of processor usage metrics, set the `name` parameter to the name of the first processor for which statistics are desired, and set the `desired_number` parameter. To start from the first processor, set the `name` parameter to "". The `userbuff` parameter must always point to a memory area big enough to contain the desired number of `perfstat_cpu_t` structures that will be copied by this function. Upon return, the `name` parameter will be set to either the name of the next CPU, or to "" after all structures have been copied.

To retrieve the number of available sets of processor usage metrics, set the `name` and `userbuff` parameters to NULL, and the `desired_number` parameter to 0. The returned value will be the number of available sets.
This number represents the number of logical processors for which statistics are available. In a dynamic LPAR environment, this number is the highest logical index of an online processor since the last reboot. See the Perfstat API article in Performance Tools and APIs Technical Reference for more information on the `perfstat_cpu` subroutine and DLPAR.

In AIX 5.3 and later, SPLPAR environments virtualize physical processors. To help accurately measure the resource use in a virtualized environment, the POWER5™ family of processors implements a register PURR (Processor Utilization Resource Register) for each core. The PURR is a 64-bit counter with the same units as the timebase register and tracks the real physical processor resource used on a per-thread or per-partition level. The PURR registers are not compatible with previous global counters (user, system, idle and wait fields) returned by the `perfstat_cpu` and the `perfstat_cpu_total` subroutines. All data consumers requiring processor utilization must be modified to support PURR-based computations as shown in the example for `perfstat_partition_total()` interface under Perfstat API programming.

**Parameters**

- **name**
  - Contains either "", FIRST_CPU, or a name identifying the first logical processor for which statistics are desired. Logical processor names are: `cpu0`, `cpu1`, ...
  - To provide binary compatibility with previous versions of the library, names like `proc0`, `proc1`, ... will still be accepted. These names will be treated as if their corresponding `cpuN` name was used, but the names returned in the structures will always be names starting with `cpu`.

- **userbuff**
  - Points to the memory area that is to be filled with one or more `perfstat_cpu_t` structures.

- **sizeof_struct**
  - Specifies the size of the `perfstat_cpu_t` structure: `sizeof(perfstat_cpu_t)`.

- **desired_number**
  - Specifies the number of `perfstat_cpu_t` structures to copy to `userbuff`.

**Return Values**

Unless the `perfstat_cpu` subroutine is used to retrieve the number of available structures, the number of structures filled is returned upon successful completion. If unsuccessful, a value of -1 is returned and the `errno` global variable is set.

**Error Codes**

The `perfstat_cpu` subroutine is unsuccessful if the following is true:

- **EINVAL**
  - One of the parameters is not valid.

**Files**

The `libperfstat.h` file defines standard macros, data types, and subroutines.

**Related Information**

The `perfstat_netbuffer Subroutine` on page 1003, `perfstat_cpu_total Subroutine` on page 994, `perfstat_disk Subroutine` on page 995, `perfstat_diskadapter Subroutine` on page 997, `perfstat_diskpath Subroutine` on page 998, `perfstat_disk_total Subroutine` on page 1000, `perfstat_memory_total Subroutine` on page 1002, `perfstat_netinterface Subroutine` on page 1004, `perfstat_netinterface_total Subroutine` on page 1006, `perfstat_pagingspace Subroutine` on page 1007, `perfstat_partial_reset Subroutine` on page 1008, `perfstat_protocol Subroutine` on page 1011, and `perfstat_reset Subroutine` on page 1013.

Perfstat API in Performance Tools and APIs Technical Reference.
perfstat_cpu_total Subroutine

Purpose
Retrieves global processor usage statistics.

Library
Perfstat library (libperfstat.a)

Syntax
#include <libperfstat.h>

int perfstat_cpu_total (name, userbuff, sizeof_struct, desired_number);

perfstat_id_t *name;
perfstat_cpu_total_t *userbuff;
size_t sizeof_struct;
int desired_number;

Description
The perfstat_cpu_total subroutine returns global processor usage statistics in a perfstat_cpu_total_t structure.

To get statistics that are global to the whole system, the name parameter must be set to NULL, the userbuff parameter must be allocated, and the desired_number parameter must be set to 1.

The perfstat_cpu_total subroutine retrieves information from the ODM database. This information is automatically cached into a dictionary which is assumed to be frozen once loaded. The perfstat_reset subroutine must be called to flush the dictionary whenever the machine configuration has changed.

In AIX 5.3 and later, SPLPAR environments virtualize physical processors. To help accurately measure the resource used in a virtualized environment, the POWER5 family of processors implements a register PURR (Processor Utilization Resource Register) for each core. The PURR is a 64-bit counter with the same units as the timebase register and tracks the real physical processor resource used on a per-thread or per-partition level. The PURR registers are not compatible with previous global counters (user, system, idle and wait fields) returned by the perfstat_cpu and the perfstat_cpu_total subroutines. All data consumers requiring processor use must be modified to support PURR-based computations as shown in the example for the perfstat_partition_total interface under Perfstat API programming.

Parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>name</td>
<td>Must set to NULL.</td>
</tr>
<tr>
<td>userbuff</td>
<td>Points to the memory area that is to be filled with the perfstat_cpu_total_t structure.</td>
</tr>
<tr>
<td>sizeof_struct</td>
<td>Specifies the size of the perfstat_cpu_total_t structure: sizeof(perfstat_cpu_total_t).</td>
</tr>
<tr>
<td>desired_number</td>
<td>Must set to 1.</td>
</tr>
</tbody>
</table>

Return Values
Upon successful completion, the number of structures filled is returned. If unsuccessful, a value of -1 is returned and the errno global variable is set.

Error Codes
The perfstat_cpu_total subroutine is unsuccessful if one of the following is true:
One of the parameters is not valid.
Insufficient memory.
The string default length is too short.

Files
The libperfstat.h file defines standard macros, data types, and subroutines.

Related Information

Perfstat API in Performance Tools and APIs Technical Reference.

perfstat_disk Subroutine

Purpose
Retrieves individual disk usage statistics.

Library
Perfstat library (libperfstat.a)

Syntax
```
#include <libperfstat.h>

int perfstat_disk (name, userbuff, sizeof_struct, desired_number)

perfsat_id_t *name;
perfstat_disk_t *userbuff;
size_t sizeof_struct;
int desired_number;
```

Description
The perfstat_disk subroutine retrieves one or more individual disk usage statistics. The same function can also be used to retrieve the number of available sets of disk statistics.

To get one or more sets of disk usage metrics, set the name parameter to the name of the first disk for which statistics are desired, and set the desired_number parameter. To start from the first disk, specify "" or FIRST_DISK as the name. The userbuff parameter must always point to a memory area big enough to contain the desired number of perfstat_disk_t structures that will be copied by this function. Upon return, the name parameter will be set to either the name of the next disk, or to "" after all structures have been copied.

To retrieve the number of available sets of disk usage metrics, set the name and userbuff parameters to NULL, and the desired_number parameter to 0. The returned value will be the number of available sets.

The perfstat_disk subroutine retrieves information from the ODM database. This information is automatically cached into a dictionary which is assumed to be frozen once loaded. The perfstat_reset subroutine must be called to flush the dictionary whenever the machine configuration has changed.
To improve system performance, the collection of disk input and output statistics is disabled by default in current releases of AIX.

To enable the collection of this data, run:
```
chdev -l sys0 -a iostat=true
```
To display the current setting, run:
```
lsattr -E -l sys0 -a iostat
```

Another way to enable the collection of the disk input and output statistics is to use the `sys_parm` API and the `SYSP_V_IOSTRUN` flag:

To get the current status of the flag, run the following:
```
struct vario var;
sys_parm(SYSP_GET,SYSP_V_IOSTRUN, &var);
```
To set the flag, run the following:
```
struct vario var;
var.v.v_iostrun.value=1; /* 1 to set & 0 to unset */
sys_parm(SYSP_SET,SYSP_V_IOSTRUN, &var);
```

**Parameters**

- **name**
  - Contains either "", FIRST_DISK, or a name identifying the first disk for which statistics are desired. For example:
    ```
    hdisk0, hdisk1, ...
    ```
- **userbuff**
  - Points to the memory area to be filled with one or more `perfstat_disk_t` structures.
- **sizeof_struct**
  - Specifies the size of the `perfstat_disk_t` structure: `sizeof(perfstat_disk_t)`
- **desired_number**
  - Specifies the number of `perfstat_disk_t` structures to copy to `userbuff`.

**Return Values**

Unless the function is used to retrieve the number of available structures, the number of structures filled is returned upon successful completion. If unsuccessful, a value of -1 is returned and the `errno` global variable is set.

**Error Codes**

The `perfstat_disk` subroutine is unsuccessful if one of the following is true:

- **EINVAL**
  - One of the parameters is not valid.
- **EFAULT**
  - Insufficient memory.
- **ENOMEM**
  - The string default length is too short.
- **ENOMSG**
  - Cannot access the dictionary.

**Files**

The `libperfstat.h` file defines standard macros, data types, and subroutines.

**Related Information**

- "perfstat_netbuffer Subroutine" on page 1003
- "perfstat_cpu Subroutine" on page 992
- "perfstat_cpu_total Subroutine" on page 994
- "perfstat_diskadapter Subroutine" on page 997
- "perfstat_diskpath Subroutine" on page 998
- "perfstat_disk_total Subroutine" on page 1000
- "perfstat_memory_total Subroutine" on page 1002
- "perfstat_netinterface Subroutine" on page 1004
- "perfstat_netinterface_total Subroutine" on page 1006
Perfstat API in Performance Tools and APIs Technical Reference.

### perfstat_diskadapter Subroutine

#### Purpose
Retrieves individual disk adapter usage statistics.

#### Library
Perfstat Library (libperfstat.a)

#### Syntax
```c
#include <libperfstat.h>

int perfstat_diskadapter (name, userbuff, sizeof_struct, desired_number)
perfstat_id_t *name;
perfstat_diskadapter_t *userbuff;
size_t sizeof_struct;
int desired_number;
```

#### Description
The `perfstat_diskadapter` subroutine retrieves one or more individual disk adapter usage statistics. The same function can be used to retrieve the number of available sets of adapter statistics.

To get one or more sets of disk adapter usage metrics, set the `name` parameter to the name of the first disk adapter for which statistics are desired, and set the `desired_number` parameter. To start from the first disk adapter, set the `name` parameter to "" or FIRST_DISKADAPTER. The `userbuff` parameter must point to a memory area big enough to contain the desired number of `perfstat_diskadapter_t` structures which will be copied by this function. Upon return, the `name` parameter will be set to either the name of the next disk adapter, or to "" if all structures have been copied.

To retrieve the number of available sets of disk adapter usage metrics, set the `name` and `userbuff` parameters to NULL, and the `desired_number` parameter to 0. The returned value will be the number of available sets.

The `perfstat_diskadapter` subroutine retrieves information from the ODM database. This information is automatically cached into a dictionary which is assumed to be frozen once loaded. The `perfstat_reset` subroutine must be called to flush the dictionary whenever the machine configuration has changed.

To improve system performance, the collection of disk input/output statistics is disabled by default in current releases of AIX.

To enable the collection of this data, use:
```
chdev -l sys0 -a iostat=true
```

To display the current setting, use:
```
lsattr -E -l sys0 -a iostat
```

Another way to enable the collection of the disk input/output statistics is to use the `sys_parm` API and the `SYSP_V_IOSTRUN` flag:
To get the current status of the flag:

```c
struct vario var;
sys_parm(SYSP_GET, SYSP_V_IOSTRUN, &var);
```

To set the flag:

```c
struct vario var;
var.v.v_iostrun.value = 1; /* 1 to set & 0 to unset */
sys_parm(SYSP_SET, SYSP_V_IOSTRUN, &var);
```

**Parameters**

- **name**: Contains either "", FIRST_DISKADAPTER, or a name identifying the first disk adapter for which statistics are desired. For example:
  ```c
  scsi0, scsi1, ...
  ```
- **userbuff**: Points to the memory area to be filled with one or more `perfstat_diskadapter_t` structures.
- **sizeof_struct**: Specifies the size of the `perfstat_diskadapter_t` structure:
  ```c
  sizeof(perfstat_diskadapter_t)
  ```
- **desired_number**: Specifies the number of `perfstat_diskadapter_t` structures to copy to `userbuff`.

**Return Values**

Unless the function is used to retrieve the number of available structures, the number of structures filled is returned upon successful completion. If unsuccessful, a value of -1 is returned and the `errno` global variable is set.

**Error Codes**

The `perfstat_diskadapter` subroutine is unsuccessful if one of the following is true:

- **EINVAL**: One of the parameters is not valid.
- **EFAULT**: Insufficient memory.
- **ENOMEM**: The string default length is too short.
- **ENOMSG**: Cannot access the dictionary.

**Files**

The `libperfstat.h` file defines standard macros, data types, and subroutines.

**Related Information**

- "perfstat_netbuffer Subroutine" on page 1003
- "perfstat_cpu Subroutine" on page 992
- "perfstat_cpu_total Subroutine" on page 994
- "perfstat_disk Subroutine" on page 995
- "perfstat_diskpath Subroutine" on page 1000
- "perfstat_disk_total Subroutine" on page 1002
- "perfstat_netinterface Subroutine" on page 1004
- "perfstat_netinterface_total Subroutine" on page 1006
- "perfstat_pagingspace Subroutine" on page 1007
- "perfstat_partial_reset Subroutine" on page 1008
- "perfstat_protocol Subroutine" on page 1011
- "perfstat_reset Subroutine" on page 1013

**perfstat_diskpath Subroutine**

**Purpose**

Retrieves individual disk path usage statistics.
Library
Perfstat library (libperfstat.a)

Syntax
#include <libperfstat.h>

int perfstat_diskpath (name, userbuff, sizeof_struct, desired_number);

perfstat_id_t *name;
perfstat_diskpath_t *userbuff
size_t sizeof_struct;
int desired_number;

Description
The perfstat_diskpath subroutine retrieves one or more individual disk path usage statistics. The same function can also be used to retrieve the number of available sets of disk path statistics.

To get one or more sets of disk path usage metrics, set the name parameter to the name of the first disk path for which statistics are desired, and set the desired_number parameter. To start from the first disk path, specify "" or FIRST_DISKPATH as the name parameter. To start from the first path of a specific disk, set the name parameter to the diskname. The userbuff parameter must always point to a memory area big enough to contain the desired number of perfstat_diskpath_t structures that will be copied by this function. Upon return, the name parameter will be set to either the name of the next disk path, or to "" after all structures have been copied.

To retrieve the number of available sets of disk path usage metrics, set the name and userbuff parameters to NULL, and the desired_number parameter to 0. The number of available sets is returned.

The perfstat_diskpath subroutine retrieves information from the ODM database. This information is automatically cached into a dictionary which is assumed to be frozen once loaded. The perfstat_reset subroutine must be called to flush the dictionary whenever the machine configuration has changed.

To improve system performance, the collection of disk input and output statistics is disabled by default in current releases of AIX.

To enable the collection of this data, run:
chdev -l sys0 -a iostat=true

To display the current setting, run:
lsattr -E -l sys0 -a iostat

Another way to enable the collection of the disk input and output statistics is to use the sys_parm API and the SYSP_V_IOSTRUN flag:

To get the current status of the flag, run the following:
struct vario var;
sys_parm(SYSP_GET,SYSP_V_IOSTRUN, &var);

To set the flag, run the following:
struct vario var;
var.v.v_iostrun.value=1; /* 1 to set & 0 to unset */
sys_parm(SYSP_SET,SYSP_V_IOSTRUN, &var);
Parameters

name
Contains either "", FIRST_DISKPATH, a name identifying the first disk path for which statistics are desired, or a name identifying a disk for which path statistics are desired. For example: hdisk0_Path2, hdisk1_Path0, ... or hdisk5 (equivalent to hdisk5_Pathfirstpath)

userbuff
Points to the memory area to be filled with one or more perfstat_diskpath_t structures.

sizeof_struct
Specifies the size of the perfstat_diskpath_t structure: sizeof(perfstat_diskpath_t)

desired_number
Specifies the number of perfstat_diskpath_t structures to copy to userbuff.

Return Values

Unless the function is used to retrieve the number of available structures, the number of structures filled is returned upon successful completion. If unsuccessful, a value of -1 is returned and the errno global variable is set.

Error Codes

The perfstat_diskpath subroutine is unsuccessful if one of the following is true:

EINVAL
One of the parameters is not valid.

EFAULT
Insufficient memory.

ENOMEM
The string default length is too short.

ENOMSG
Cannot access the dictionary.

Files

The libperfstat.h file defines standard macros, data types, and subroutines.

Related Information

"perfstat_netbuffer Subroutine" on page 1003, "perfstat_cpu Subroutine" on page 992, "perfstat_cpu_total Subroutine" on page 994, "perfstat_disk Subroutine" on page 995, "perfstat_diskadapter Subroutine" on page 997, "perfstat_diskpath Subroutine" on page 998, "perfstat_disk_total Subroutine", "perfstat_memory_total Subroutine" on page 1002, "perfstat_netinterface Subroutine" on page 1004, "perfstat_netinterface_total Subroutine" on page 1006, "perfstat_pagingspace Subroutine" on page 1007, "perfstat_partial_reset Subroutine" on page 1008, "perfstat_protocol Subroutine" on page 1011, and "perfstat_reset Subroutine" on page 1013.

Perfstat API in Performance Tools and APIs Technical Reference.

perfstat_disk_total Subroutine

Purpose

Retrieves global disk usage statistics.

Library

Perfstat library (libperfstat.a)

Syntax

#include <libperfstat.h>

int perfstat_disk_total (name, userbuff, sizeof_struct, desired_number)
perfstat_id_t *name;

1000    Technical Reference, Volume 1: Base Operating System and Extensions
perfstat_disk_total_t *userbuff;
size_t sizeof_struct;
int desired_number;

Description
The `perfstat_disk_total` subroutine returns global disk usage statistics in a `perfstat_disk_total_t` structure.

To get statistics that are global to the whole system, the `name` parameter must be set to NULL, the `userbuff` parameter must be allocated, and the `desired_number` parameter must be set to 1.

The `perfstat_disk_total` subroutine retrieves information from the ODM database. This information is automatically cached into a dictionary which is assumed to be frozen once loaded. The `perfstat_reset` subroutine must be called to flush the dictionary whenever the machine configuration has changed.

To improve system performance, the collection of disk input and output statistics is disabled by default in current releases of AIX.

To enable the collection of this data, run:
```
chdev -l sys0 -a iostat=true
```

To display the current setting, run:
```
lsattr -E -l sys0 -a iostat
```

Another way to enable the collection of the disk input and output statistics is to use the `sys_parm` API and the `SYSP_V_IOSTRUN` flag:

To get the current status of the flag, run the following:
```
struct vario var;
sys_parm(SYSP_GET,SYSP_V_IOSTRUN, &var);
```

To set the flag, run the following:
```
struct vario var;
var.v.v_iostrun.value=1; /* 1 to set & 0 to unset */
sys_parm(SYSP_SET,SYSP_V_IOSTRUN, &var);
```

Parameters
- **name**: Must set to NULL.
- **userbuff**: Points to the memory area that is to be filled with one or more `perfstat_disk_total_t` structures.
- **sizeof_struct**: Specifies the size of the `perfstat_disk_total_t` structure: `sizeof(perfstat_disk_total_t)`
- **desired_number**: Must set to 1.

Return Values
Upon successful completion, the number of structures that could be filled is returned. This is always 1. If unsuccessful, a value of -1 is returned and the `errno` global variable is set.

Error Codes
The `perfstat_disk_total` subroutine is unsuccessful if one of the following is true:

- **EINVAL**: One of the parameters is not valid.
- **EFAULT**: Insufficient memory.
Files
The libperfstat.h file defines standard macros, data types, and subroutines.

Related Information
"perfstat_netbuffer Subroutine" on page 1003, "perfstat_cpu Subroutine" on page 992, "perfstat_cpu_total Subroutine" on page 994, "perfstat_disk Subroutine" on page 995, "perfstat_diskadapter Subroutine" on page 998, "perfstat_memory_total Subroutine", "perfstat_netinterface Subroutine" on page 1004, "perfstat_netinterface_total Subroutine" on page 1006, "perfstat_pagingspace Subroutine" on page 1007, "perfstat_partial_reset Subroutine" on page 1008, "perfstat_protocol Subroutine" on page 1011, and "perfstat_reset Subroutine" on page 1013.

Perfstat API in Performance Tools and APIs Technical Reference.

**perfstat_memory_total Subroutine**

**Purpose**
Retrieves global memory usage statistics.

**Library**
Perfstat Library (libperfstat.a)

**Syntax**
```
#include <libperfstat.h>

int perfstat_memory_total (name, userbuff, sizeof_struct, desired_number)
```

**Description**
The **perfstat_memory_total** subroutine returns global memory usage statistics in a **perfstat_memory_total_t** structure.

To get statistics that are global to the whole system, the **name** parameter must be set to NULL, the **userbuff** parameter must be allocated, and the **desired_number** parameter must be set to 1.

**Parameters**
- **name**
  - Must be set to NULL.
- **userbuff**
  - Points to the memory area that is to be filled with the **perfstat_memory_total_t** structure.
- **sizeof_struct**
  - Specifies the size of the **perfstat_memory_total_t** structure: `sizeof(perfstat_memory_total_t)`.
- **desired_number**
  - Must be set to 1.

**Return Values**
Upon successful completion, the number of structures filled is returned. This will always be 1. If unsuccessful, a value of -1 is returned and the **errno** global variable is set.
Error Codes
The **perfstat_memory_total** subroutine is unsuccessful if the following is true:

**EINVAL**  
One of the parameters is not valid.

Files
The **libperfstat.h** file defines standard macros, data types, and subroutines.

Related Information
“**perfstat_netbuffer Subroutine**” on page 992, “**perfstat_cpu Subroutine**” on page 994, “**perfstat_disk Subroutine**” on page 995, “**perfstat_diskadapter Subroutine**” on page 997, “**perfstat_diskpath Subroutine**” on page 998, “**perfstat_disk_total Subroutine”** on page 1000, “**perfstat_netinterface Subroutine**” on page 1004, “**perfstat_netinterface_total Subroutine**” on page 1006, “**perfstat_pagingspace Subroutine**” on page 1007, “**perfstat_partial_reset Subroutine**” on page 1008, and “**perfstat_protocol Subroutine**” on page 1011.

Perfstat API in Performance Tools and APIs Technical Reference.

---

**perfstat_netbuffer Subroutine**

**Purpose**
Retrieves network buffer allocation usage statistics.

**Library**
Perfstat Library (**libperfstat.a**)

**Syntax**

```c
#include <libperfstat.h>

int perfstat_netbuffer (name, userbuff, sizeof_struct, desired_number, name_t *name, perfstat_netbuffer_t *userbuff, size_t *sizeof_struct, int *desired_number);
```

**Description**
The **perfstat_netbuffer** subroutine retrieves statistics about network buffer allocations for each possible buffer size. Returned counts are the sum of allocation statistics for all processors (kernel statistics are kept per size per processor) corresponding to a buffer size.

To get one or more sets of network buffer allocation usage metrics, set the **name** parameter to the network buffer size for which statistics are desired, and set the **desired_number** parameter. To start from the first network buffer size, specify “” or FIRST_NETBUFFER in the **name** parameter. The **userbuff** parameter must point to a memory area big enough to contain the desired number of **perfstat_netbuffer_t** structures which will be copied by this function.

Upon return, the **name** parameter will be set to either the ASCII size of the next buffer type, or to “” if all structures have been copied. Only the statistics for network buffer sizes that have been used are returned. Consequently, there can be holes in the returned array of statistics, and the structure corresponding to allocations of size 4096 may directly follow the structure for size 256 (in case 512, 1024 and 2048 have not been used yet). The structure corresponding to a buffer size not used yet is returned (with all fields set to 0) when it is directly asked for by name.
To retrieve the number of available sets of network buffer usage metrics, set the name and userbuff parameters to NULL, and the desired_number parameter to 0. The returned value will be the number of available sets.

**Parameters**

- **name**: Contains either "", FIRST_NETBUFFER, or the size of the network buffer in ASCII. It is a power of 2. For example: 32, 64, 128, ..., 16384
- **userbuff**: Points to the memory area to be filled with one or more perfstat_netbuffer_t structures.
- **sizeof_struct**: Specifies the size of the perfstat_netbuffer_t structure: sizeof(perfstat_netbuffer_t)
- **desired_number**: Specifies the number of perfstat_netbuffer_t structures to copy to userbuff.

**Return Values**

Upon successful completion, the number of structures which could be filled is returned. If unsuccessful, a value of -1 is returned and the errno global variable is set.

**Error Codes**

The perfstat_netbuffer subroutine is unsuccessful if the following is true:

- **EINVAL**: One of the parameters is not valid.

**Files**

The `libperfstat.h` file defines standard macros, data types, and subroutines.

**Related Information**

- "perfstat_cpu Subroutine" on page 992, "perfstat_cpu_total Subroutine" on page 994, "perfstat_memory_total Subroutine" on page 1002, "perfstat_disk Subroutine" on page 995, "perfstat_diskpath Subroutine" on page 998, "perfstat_disk_total Subroutine" on page 1000, "perfstat_netinterface_total Subroutine" on page 1006, "perfstat_diskadapter Subroutine" on page 997, "perfstat_partial_reset Subroutine" on page 1006, "perfstat_protocol Subroutine" on page 1011, and "perfstat_pagingspace Subroutine" on page 1007.

Perfstat API in Performance Tools and APIs Technical Reference.

**perfstat_netinterface Subroutine**

**Purpose**

Retrieves individual network interface usage statistics.

**Library**

Perfstat Library (`libperfstat.a`)

**Syntax**

```c
#include <libperfstat.h>

int perfstat_netinterface (name, userbuff, sizeof_struct, desired_number)

perfstat_id_t *name;
```
perfstat_netinterface_t *userbuff;
size_t sizeof_struct;
int desired_number;

Description
The perfstat_netinterface subroutine retrieves one or more individual network interface usage statistics. The same function can also be used to retrieve the number of available sets of network interface statistics.

To get one or more sets of network interface usage metrics, set the name parameter to the name of the first network interface for which statistics are desired, and set the desired_number parameter. To start from the first network interface, set the name parameter to "" or FIRST_NETINTERFACE. The userbuff parameter must always point to a memory area big enough to contain the desired number of perfstat_netinterface_t structures that will be copied by this function. Upon return, the name parameter will be set to either the name of the next network interface, or to "" after all structures have been copied.

To retrieve the number of available sets of network interface usage metrics, set the name and userbuff parameters to NULL, and the desired_number parameter to 0. The returned value will be the number of available sets.

The perfstat_netinterface subroutine retrieves information from the ODM database. This information is automatically cached into a dictionary which is assumed to be frozen once loaded. The perfstat_reset subroutine must be called to flush the dictionary whenever the machine configuration has changed.

Parameters
name Contains either "", FIRST_NETINTERFACE, or a name identifying the first network interface for which statistics are desired. For example;
en0, tr10, ...
userbuff Points to the memory area that is to be filled with one or more perfstat_netinterface_t structures.
sizeof_struct Specifies the size of the perfstat_netinterface_t structure: sizeof(perfstat_netinterface_t)
desired_number Specifies the number of perfstat_netinterface_t structures to copy to userbuff.

Return Values
Upon successful completion unless the function is used to retrieve the number of available structures, the number of structures filled is returned. If unsuccessful, a value of -1 is returned and the errno global variable is set.

Error Codes
The perfstat_netinterface subroutine is unsuccessful if one of the following is true:
EINVAL One of the parameters is not valid.
EFAULT Insufficient memory.
ENOMEM The string default length is too short.
ENOMSG Cannot access the dictionary.

Files
The libperfstat.h file defines standard macros, data types, and subroutines.

Related Information
perfstat_netbuffer Subroutine on page 1003, perfstat_cpu Subroutine on page 992, perfstat_cpu_total Subroutine on page 994, perfstat_disk Subroutine on page 995, perfstat_diskadapter Subroutine on page 997
**perfstat_netinterface_total Subroutine**

**Purpose**
Retrieves global network interface usage statistics.

**Library**
Perfstat Library (libperfstat.a)

**Syntax**
```
#include <libperfstat.h>

int perfstat_netinterface_total (name, userbuff, sizeof_struct, desired_number);
```

**Description**
The `perfstat_netinterface_total` subroutine returns global network interface usage statistics in a `perfstat_netinterface_total_t` structure.

To get statistics that are global to the whole system, the `name` parameter must be set to NULL, the `userbuff` parameter must be allocated, and the `desired_number` parameter must be set to 1.

The `perfstat_netinterface_total` subroutine retrieves information from the ODM database. This information is automatically cached into a dictionary which is assumed to be frozen once loaded. The `perfstat_reset` subroutine must be called to flush the dictionary whenever the machine configuration has changed.

**Parameters**
- **name**
  Must be set to NULL.
- **userbuff**
  Points to the memory area that is to be filled with the `perfstat_netinterface_total_t` structure.
- **sizeof_struct**
  Specifies the size of the `perfstat_netinterface_total_t` structure:
  `sizeof(perfstat_netinterface_total_t)`.
- **desired_number**
  Must be set to 1.

**Return Values**
Upon successful completion, the number of structures filled is returned. This will always be 1. If unsuccessful, a value of -1 is returned and the `errno` variable is set.

**Error Codes**
The `perfstat_netinterface_total` subroutine is unsuccessful if one of the following is true:

- **EINVAL**
  One of the parameters is not valid.
Files
The `libperfstat.h` file defines standard macros, data types, and subroutines.

Related Information
"perfstat_netbuffer Subroutine” on page 1003, "perfstat_cpu Subroutine” on page 992, "perfstat_cpu_total Subroutine” on page 994, "perfstat_disk Subroutine” on page 995, "perfstat_diskadapter Subroutine” on page 997, "perfstat_diskpath Subroutine” on page 998, "perfstat_disk_total Subroutine” on page 1000, "perfstat_memory_total Subroutine” on page 1002, "perfstat_netinterface Subroutine” on page 1004, "perfstat_pagingspace Subroutine,” "perfstat_partial_reset Subroutine” on page 1008, "perfstat_protocol Subroutine” on page 1011, and "perfstat_reset Subroutine” on page 1013.

Perfstat API in Performance Tools and APIs Technical Reference.

**perfstat_pagingspace Subroutine**

**Purpose**
Retrieves individual paging space usage statistics.

**Library**
Perfstat Library (`libperfstat.a`)

**Syntax**
```c
#include "libperfstat.h"

int perfstat_pagingspace (name, userbuff, sizeof_struct, desired_number)
    perfstat_id_t *name;
    perfstat_pagingspace_t *userbuff;
    size_t sizeof_struct;
    int desired_number;
```

**Description**
This function retrieves one or more individual pagingspace usage statistics. The same functions can also be used to retrieve the number of available sets of paging space statistics.

To get one or more sets of paging space usage metrics, set the `name` parameter to the name of the first paging space for which statistics are desired, and set the `desired_number` parameter. To start from the first paging space, set the `name` parameter to "" or FIRST_PAGINGSPACE. In either case, `userbuff` must point to a memory area big enough to contain the desired number of `perfstat_pagingspace_t` structures which will be copied by this function. Upon return, the `name` parameter will be set to either the name of the next paging space, or to "" if all structures have been copied.

To retrieve the number of available sets of paging space usage metrics, set the `name` and `userbuff` parameters to NULL, and the `desired_number` parameter to 0. The number of available sets will be returned.

The `perfstat_pagingspace` subroutine retrieves information from the ODM database. This information is automatically cached into a dictionary which is assumed to be frozen once loaded. The `perfstat_reset` subroutine must be called to flush the dictionary whenever the machine configuration has changed.
Parameters

name
Contains either "", FIRST_PAGINGSPACE, or a name identifying the first paging space for which statistics are desired. For example:
	paging00, hd6, ...

userbuff
Points to the memory area to be filled with one or more perfstat_pagingspace_t structures.

sizeof_struct
Specifies the size of the perfstat_pagingspace_t structure:

sizeof(perfstat_pagingspace_t)

desired_number
Specifies the number of perfstat_pagingspace_t structures to copy to userbuff.

Return Values

Unless the perfstat_pagingspace subroutine is used to retrieve the number of available structures, the number of structures which could be filled is returned upon successful completion. If unsuccessful, a value of -1 is returned and the errno global variable is set.

Error Codes

The perfstat_pagingspace subroutine is unsuccessful if one of the following are true:

EINVAL
One of the parameters is not valid.

Files

The libperfstat.h file defines standard macros, data types, and subroutines.

Related Information

"perfstat_netbuffer Subroutine" on page 1003, "perfstat_cpu Subroutine" on page 992, "perfstat_cpu_total Subroutine" on page 994, "perfstat_disk Subroutine" on page 995, "perfstat_diskadapter Subroutine" on page 997, "perfstat_diskpath Subroutine" on page 998, "perfstat_disk_total Subroutine" on page 1000, "perfstat_memory_total Subroutine" on page 1002, "perfstat_netinterface Subroutine" on page 1004, "perfstat_netinterface_total Subroutine" on page 1006, "perfstat_partial_reset Subroutine," "perfstat_protocol Subroutine" on page 1011, and "perfstat_reset Subroutine" on page 1013.

Perfstat API in Performance Tools and APIs Technical Reference.

perfstat_partial_reset Subroutine

Purpose

Empties part of the libperfstat configuration information cache or resets system minimum and maximum counters for disks.

Library

perfstat library (libperfstat.a)

Syntax

#include <libperfstat.h>

int perfstat_partial_reset (name, resetmask)
char * name;
ulonglong_t resetmask;

1008 Technical Reference, Volume 1: Base Operating System and Extensions
Description

The `perfstat_cpu_total`, `perfstat_disk`, `perfstat_diskadapter`, `perfstat_netinterface`, and `perfstat_pagingspace` subroutines return configuration information that is retrieved from the ODM database and automatically cached by the library. Other metrics provided by the LVM library and the `swapqry` subroutine are also cached for performance purpose.

The `perfstat_partial_reset` subroutine flushes some of this information cache and should be called whenever an identified part of the machine configuration has changed.

The `perfstat_partial_reset` subroutine can be used to reset a particular component (such as `hdisk0` or `en1`) when the `name` parameter is not NULL and the `resetmask` parameter contains only one bit. It can also be used to remove a whole category (such as disks or disk paths) from the cached information.

When the `name` parameter is NULL, the `resetmask` parameter can contain a combination of bits, such as `FLUSH_DISK|RESET_DISK_MINMAX|FLUSH_CPUTOTAL`. For more information on the `perfstat_partial_reset` subroutine, see Perfstat API Programming.

Several bit masks are available for the `resetmask` parameter. The behavior of the function is as follows:

<table>
<thead>
<tr>
<th>resetmask</th>
<th>Action when <code>name</code> is NULL</th>
<th>Action when <code>name</code> is not NULL and a single <code>resetmask</code> is set</th>
</tr>
</thead>
<tbody>
<tr>
<td>FLUSH_CPUTOTAL</td>
<td>Flush speed and description in the <code>perfstat_cputotal_t</code> structure</td>
<td>An error is returned, and <code>errno</code> is set to EINVAL.</td>
</tr>
<tr>
<td>FLUSH_DISK</td>
<td>Flush description, adapter, size, free, and <code>vgname</code> in every <code>perfstat_disk_t</code> structure. Flush the list of disk adapters. Flush size, free, and description in every <code>perfstat_diskadapter_t</code> structure.</td>
<td>Flush description, adapter, size, free, and <code>vgname</code> in the specified <code>perfstat_disk_t</code> structure. Flush adapter in every <code>perfstat_diskpath_t</code> that matches the disk name followed by <code>_Path</code>. Flush size, free, and description of each <code>perfstat_diskadapter_t</code> that is linked to a path leading to this disk or to the disk itself.</td>
</tr>
<tr>
<td>RESET_DISK_ALL</td>
<td>Reset system resident all fields in every <code>perfstat_disk_t</code> structure.</td>
<td>An error is returned, and <code>errno</code> is set to EINVAL.</td>
</tr>
<tr>
<td>RESET_DISK_MINMAX</td>
<td>Reset system resident <code>min_rserv</code>, <code>max_rserv</code>, <code>min_wserv</code>, <code>max_wserv</code>, <code>wq_min_time</code> and <code>wq_max_time</code> in every <code>perfstat_disk_t</code> structure.</td>
<td>An error is returned, and <code>errno</code> is set to ENOTSUP.</td>
</tr>
<tr>
<td>FLUSH_DISKADAPTER</td>
<td>Flush the list of disk adapters. Flush size, free, and description in every <code>perfstat_diskadapter_t</code> structure. Flush adapter in every <code>perfstat_diskpath_t</code> structure. Flush description and adapter in every <code>perfstat_disk_t</code> structure.</td>
<td>Flush the list of disk adapters. Flush size, free, and description in the specified <code>perfstat_diskadapter_t</code> structure.</td>
</tr>
<tr>
<td>FLUSH_DISKPATH</td>
<td>Flush adapter in every <code>perfstat_diskpath_t</code> structure.</td>
<td>Flush adapter in the specified <code>perfstat_diskpath_t</code> structure.</td>
</tr>
<tr>
<td>FLUSH_PAGINGSPACE</td>
<td>Flush the list of paging spaces. Flush automatic, type, <code>lpsize</code>, <code>mbsize</code>, <code>hostname</code>, <code>filename</code>, and <code>vgname</code> in every <code>perfstat_pagingspace_t</code> structure.</td>
<td>Flush the list of paging spaces. Flush automatic, type, <code>lpsize</code>, <code>mbsize</code>, <code>hostname</code>, <code>filename</code>, and <code>vgname</code> in the specified <code>perfstat_pagingspace_t</code> structure.</td>
</tr>
<tr>
<td>FLUSH_NETINTERFACE</td>
<td>Flush description in every <code>perfstat_netinterface_t</code> structure.</td>
<td>Flush description in the specified <code>perfstat_netinterface_t</code> structure.</td>
</tr>
</tbody>
</table>
Parameters

name
Contains a name identifying the component that metrics should be reset from the libperfstat cache. If this parameter is NULL, matches every component.

resetmask
The category of the component if the name parameter is not NULL. The available values are listed in the preceding table. In case the name parameter is NULL, the resetmask parameter can be a combination of bits.

Return Values
The `perfstat_partial_reset` subroutine returns a value of 0 upon successful completion. If unsuccessful, a value of -1 is returned, and the `errno` global variable is set to the appropriate code.

Error Codes
Einval
One of the parameters is not valid.

Files
The `libperfstat.h` file defines standard macros, data types, and subroutines.

Related Information
The `perfstat_cpu Subroutine` on page 992, `perfstat_cpu_total Subroutine` on page 994, `perfstat_disk Subroutine` on page 995, `perfstat_diskadapter Subroutine` on page 997, `perfstat_diskpath Subroutine` on page 998, `perfstat_disk_total Subroutine` on page 1000, `perfstat_memory_total Subroutine` on page 1002, `perfstat_netbuffer Subroutine` on page 1003, `perfstat_netinterface Subroutine` on page 1004, `perfstat_netinterface_total Subroutine` on page 1006, `perfstat_pagingspace Subroutine` on page 1007, `perfstat_partition_total Subroutine`, `perfstat_protocol Subroutine` on page 1011, and `perfstat_reset Subroutine` on page 1013.

Perfstat API Programming

**perfstat_partition_total Subroutine**

**Purpose**
Retrieves global Micro-Partitioning usage statistics.

**Library**
`perfsstat library (libperfsstat.a)

**Syntax**
```c
#include <libperfsstat.h>
int perfstat_partition_total(name, userbuff, sizeof_struct, desired_number
perftstat_id_t *name;
perfstat_partition_total_t *userbuff;
sizet sizeof_struct;
int desired_number;
u_longlong_t reserved_pages;
u_longlong_t reserved_pagesize.
```
Description

The `perfstat_partition_total` subroutine returns global Micro-Partitioning usage statistics in a `perfstat_partition_total_t` structure. To retrieve statistics that are global to the whole system, the `name` parameter must be set to NULL, the `userbuff` parameter must be allocated, and the `desired_number` parameter must be set to 1.

Parameters

- **name**: Must be set to NULL.
- **userbuff**: Points to the memory area to be filled with the `perfstat_partition_total_t` structures.
- **sizeof_struct**: Specifies the size of the `perfstat_partition_total_t` structure:
  
  ```c
  sizeof(perfstat_partition_total_t).
  ```
- **desired_number**: Must be set to 1.
- **reserved_pagesize**: Specifies the size of the pages for reserved memory. Not for use with DR operations.
- **reserved_pages**: Specifies the number of pages of type `reserved_pagesize`. This information can be retrieved by calling `vmgetinfo`. Not for use with DR operations.

Return Values

Upon successful completion, the number of structures filled is returned. If unsuccessful, a value of -1 is returned and the `errno` global variable is set.

Error Codes

- **EINVAL**: One of the parameters is not valid.
- **EFAULT**: Insufficient memory.

Files

The `libperfstat.h` file defines standard macros, data types, and subroutines.

Related Information

- “perfstat_cpu Subroutine” on page 992
- “perfstat_cpu_total Subroutine” on page 994
- “perfstat_disk Subroutine” on page 995
- “perfstat_diskadapter Subroutine” on page 997
- “perfstat_disk_total Subroutine” on page 1000
- “perfstat_memory_total Subroutine” on page 1002
- “perfstat_netbuffer Subroutine” on page 1003
- “perfstat_netinterface Subroutine” on page 1004
- “perfstat_netinterface_total Subroutine” on page 1006
- “perfstat_pagingspace Subroutine” on page 1007
- “perfstat_protocol Subroutine”
- “perfstat_reset Subroutine” on page 1013

Perfstat API in Performance Tools and APIs Technical Reference.

**perfstat_protocol Subroutine**

**Purpose**

Retrieves protocol usage statistics.

**Library**

Perfstat Library (`libperfstat.a`)
Syntax

#include <libperfstat.h>

int perfstat_protocol
   (name, userbuff, sizeof_struct, desired_number);

Perfstat API in Performance Tools and APIs Technical Reference.

1012 Technical Reference, Volume 1: Base Operating System and Extensions
**perfstat_reset Subroutine**

**Purpose**
Empties libperfstat configuration information cache.

**Library**
Perfstat Library (*libperfstat.a*)

**Syntax**
```c
#include <libperfstat.h>

void perfstat_reset ( void )
```

**Description**
The `perfstat_cpu_total`, `perfstat_disk`, `perfstat_diskadapter`, `perfstat_netinterface`, and `perfstat_pagingspace` subroutines return configuration information retrieved from the ODM database and automatically cached by the library.

The `perfstat_reset` subroutine flushes this information cache and should be called whenever the machine configuration has changed.

**Files**
The `libperfstat.h` defines standard macros, data types and subroutines.

**Related Information**
- "perfstat_cpu_total Subroutine" on page 994
- "perfstat_disk Subroutine" on page 995
- "perfstat_diskadapter Subroutine" on page 997
- "perfstat_diskpath Subroutine" on page 998
- "perfstat_netinterface Subroutine" on page 1004
- "perfstat_pagingspace Subroutine" on page 1007
- "perfstat_partial_reset Subroutine" on page 1008

Perfstat API in Performance Tools and APIs Technical Reference.

---

**perror Subroutine**

**Purpose**
Writes a message explaining a subroutine error.

**Library**
Standard C Library (*libc.a*)

**Syntax**
```c
#include <errno.h>
#include <stdio.h>

void perror ( String )

const char *String;
extern int errno;
extern char *sys_errlist[ ];
extern int sys_nerr;
```
Description

The `perror` subroutine writes a message on the standard error output that describes the last error encountered by a system call or library subroutine. The error message includes the `String` parameter string followed by a `:` (colon), a space character, the message, and a new-line character. The `String` parameter string should include the name of the program that caused the error. The error number is taken from the `errno` global variable, which is set when an error occurs but is not cleared when a successful call to the `perror` subroutine is made.

To simplify various message formats, an array of message strings is provided in the `sys_errlist` structure or use the `errno` global variable as an index into the `sys_errlist` structure to get the message string without the new-line character. The largest message number provided in the table is `sys_nerr`. Be sure to check the `sys_nerr` structure because new error codes can be added to the system before they are added to the table.

The `perror` subroutine retrieves an error message based on the language of the current locale.

After successfully completing, and before a call to the `exit` or `abort` subroutine or the completion of the `fflush` or `fclose` subroutine on the standard error stream, the `perror` subroutine marks for update the `st_ctime` and `st_mtime` fields of the file associated with the standard error stream.

Parameter

`String` Specifies a parameter string that contains the name of the program that caused the error. The ensuing printed message contains this string, a `:` (colon), and an explanation of the error.

Related Information

The `abort` subroutine, `exit` subroutine, `fflush` or `fclose` subroutine, `printf`, `fprintf`, `sprintf`, `vprintf`, `vfprintf`, `vsprintf`, or `vswprintf` subroutine, `strerror` subroutine.

Subroutines Overview in AIX 5L Version 5.3 General Programming Concepts: Writing and Debugging Programs.

pipe Subroutine

Purpose

Creates an interprocess channel.

Library

Standard C Library (`libc.a`)

Syntax

```c
#include <unistd.h>

int pipe (int FileDescriptor[2]);
```

Description

The `pipe` subroutine creates an interprocess channel called a pipe and returns two file descriptors, `FileDescriptor[0]` and `FileDescriptor[1]`. `FileDescriptor[0]` is opened for reading and `FileDescriptor[1]` is opened for writing.
A read operation on the FileDescriptor[0] parameter accesses the data written to the FileDescriptor[1] parameter on a first-in, first-out (FIFO) basis.

Write requests of PIPE_BUF bytes or fewer will not be interleaved (mixed) with data from other processes doing writes on the same pipe. PIPE_BUF is a system variable described in the pathconf subroutine. Writes of greater than PIPE_BUF bytes may have data interleaved, on arbitrary boundaries, with other writes.

If O_NONBLOCK or O_NDELAY are set, writes requests of PIPE_BUF bytes or fewer will either succeed completely or fail and return -1 with the errno global variable set to EAGAIN. A write request for more than PIPE_BUF bytes will either transfer what it can and return the number of bytes actually written, or transfer no data and return -1 with the errno global variable set to EAGAIN.

Parameters

FileDescriptor Specifies the address of an array of two integers into which the new file descriptors are placed.

Return Values

Upon successful completion, a value of 0 is returned. Otherwise, a value of -1 is returned, and the errno global variable is set to identify the error.

Error Codes

The pipe subroutine is unsuccessful if one or more the following are true:

EFAULT The FileDescriptor parameter points to a location outside of the allocated address space of the process.
EMFILE The number of open file descriptors exceeds the OPEN_MAX value.
ENFILE The system file table is full, or the device containing pipes has no free i-nodes.

Related Information

The read subroutine, select subroutine, write subroutine.

The ksh command, sh command.

Files, Directories, and File Systems for Programmers in AIX 5L Version 5.3 General Programming Concepts: Writing and Debugging Programs.

plock Subroutine

Purpose

Locks the process, text, or data in memory.

Library

Standard C Library (libc.a)

Syntax

#include <sys/lock.h>

int plock (Operation);
int Operation;
Description
The plock subroutine allows the calling process to lock or unlock its text region (text lock), its data region (data lock), or both its text and data regions (process lock) into memory. The plock subroutine does not lock the shared text segment or any shared data segments. Locked segments are pinned in memory and are immune to all routine paging. Memory locked by a parent process is not inherited by the children after a fork subroutine call. Likewise, locked memory is unlocked if a process executes one of the exec subroutines. The calling process must have the root user authority to use this subroutine.

A real-time process can use this subroutine to ensure that its code, data, and stack are always resident in memory.

Note: Before calling the plock subroutine, the user application must lower the maximum stack limit value using the ulimit subroutine.

Parameters
Operation Specifies one of the following:
- PROCLOCK
  Locks text and data into memory (process lock).
- TXTLOCK
  Locks text into memory (text lock).
- DATLOCK
  Locks data into memory (data lock).
- UNLOCK
  Removes locks.

Return Values
Upon successful completion, a value of 0 is returned to the calling process. Otherwise, a value of -1 is returned and the errno global variable is set to indicate the error.

Error Codes
The plock subroutine is unsuccessful if one or more of the following is true:
- EPERM The effective user ID of the calling process does not have the root user authority.
- EINVAL The Operation parameter has a value other than PROCLOCK, TXTLOCK, DATLOCK, or UNLOCK.
- EINVAL The Operation parameter is equal to PROCLOCK, and a process lock, text lock, or data lock already exists on the calling process.
- EINVAL The Operation parameter is equal to TXTLOCK, and a text lock or process lock already exists on the calling process.
- EINVAL The Operation parameter is equal to DATLOCK, and a data lock or process lock already exists on the calling process.
- EINVAL The Operation parameter is equal to UNLOCK, and no type of lock exists on the calling process.

Related Information
The exec ("exec: execl, execle, execvp, execve, execvp, or exec Subroutine" on page 235) subroutines, _exit, exit, or atexit ("exit, atexit, unatexit, _exit, or Exit Subroutine" on page 242) subroutine, fork ("fork, _fork, or vfork Subroutine" on page 287) subroutine, ulimit subroutine.
pm_cycles Subroutine

Purpose
Measures processor speed in cycles per second.

Library
Performance Monitor APIs Library (libpmapi.a)

Syntax
#include <pmapi.h>
double pm_cycles (void)

Description
The pm_cycles subroutine uses the Performance Monitor cycle counter and the processor real-time clock to measure the actual processor clock speed. The speed is returned in cycles per second.

Return Values
0 An error occurred.
Processor speed in cycles per second No errors occurred.

Files
/usr/include/pmapi.h Defines standard macros, data types, and subroutines.

Related Information

pm_delete_program Subroutine

Purpose
Deletes previously established systemwide Performance Monitor settings.

Library
Performance Monitor APIs Library (libpmapi.a)

Syntax
#include <pmapi.h>
int pm_delete_program ()

Description
The pm_delete_program subroutine deletes previously established systemwide Performance Monitor settings.
Return Values

0  No errors occurred.

Positive error code  Refer to the pm_error subroutine to decode the error code.

Error Codes

Refer to the pm_error subroutine.

Files

/usr/include/pmapi.h  Defines standard macros, data types, and subroutines.

Related Information

pm_init, pm_set_program, pm_get_program, pm_get_data, pm_get_tdata, pm_get_Tdata, pm_get_data_cpu, pm_get_tdata_cpu, pm_get_Tdata_cpu, pm_get_data_lcpu, pm_get_tdata_lcpu, pm_get_Tdata_lcpu, pm_start, pm_stop, pm_reset_data subroutines.

pm_delete_program_group Subroutine

Purpose

Deletes previously established Performance Monitor settings for the counting group to which a target thread belongs.

Library

Performance Monitor APIs Library (libpmapi.a)

Syntax

#include <pmapi.h>

int pm_delete_program_group (pid_t pid, tid_t tid);

Description

This subroutine supports only the 1:1 threading model. It has been superseded by the pm_delete_program_pgroup subroutine, which supports both the 1:1 and the M:N threading models. A call to this subroutine is equivalent to a call to the pm_delete_program_pgroup subroutine with a ptid parameter equal to 0.

The pm_delete_program_group subroutine deletes previously established Performance Monitor settings for a target kernel thread. The thread must be stopped and must be part of a debuggee process under the control of the calling process. The settings for the group to which the target thread belongs and from all the other threads in the same group are also deleted.
Parameters

_pid_  
Process identifier of target thread. The target process must be a debuggee under the control of the calling process.

tid  
Thread identifier of a target thread.

Return Values

<table>
<thead>
<tr>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>No errors occurred.</td>
</tr>
<tr>
<td>Positive error code</td>
<td>Refer to the pm_error subroutine to decode the error code.</td>
</tr>
</tbody>
</table>

Error Codes

Refer to the pm_error subroutine.

Files

/usr/include/pmapi.h  
Defines standard macros, data types, and subroutines.

Related Information

The pm_init subroutine, pm_error subroutine, pm_set_program_group subroutine, pm_get_program_group subroutine, pm_get_data_group subroutine, pm_start_group subroutine, pm_stop_group subroutine, pm_reset_data_group subroutine.  

pm_delete_program_mygroup Subroutine

Purpose

Deletes previously established Performance Monitor settings for the counting group to which the calling thread belongs.

Library

Performance Monitor APIs Library (libpmapi.a)

Syntax

```
#include <pmapi.h>

int pm_delete_program_mygroup ();
```

Description

The pm_delete_program_mygroup subroutine deletes previously established Performance Monitor settings for the calling kernel thread, the counting group to which it belongs, and for all the threads that are members of the same group.
Return Values

0
Positive error code Refer to the pm_error subroutine to decode the error code.

Error Codes

Refer to the pm_error subroutine.

Files

/usr/include/pma/api.h Defines standard macros, data types, and subroutines.

Related Information

pm_init (pm_init Subroutine on page 1069), pm_error (pm_error Subroutine on page 1024), pm_set_program_mygroup (pm_set_program_mygroup Subroutine on page 1086), pm_get_program_mygroup (pm_get_program_mygroup Subroutine on page 1055), pm_get_data_mygroup (pm_get_data_mygroup, pm_get_tdata_mygroup or pm_get_Tdata_mygroup Subroutine on page 1033), pm_start_mygroup (pm_start_mygroup and pm_tstart_mygroup Subroutine on page 1104), pm_stop_mygroup (pm_stop_mygroup and pm_tstop_mygroup Subroutine on page 1112), pm_reset_data_mygroup (pm_reset_data_mygroup Subroutine on page 1075) subroutines.


pm_delete_program_mythread Subroutine

Purpose

Deletes the previously established Performance Monitor settings for the calling thread.

Library

Performance Monitor APIs Library (libpmaapi.a)

Syntax

#include <pmaapi.h>

int pm_delete_program_mythread ()

Description

The pm_delete_program_mythread subroutine deletes the previously established Performance Monitor settings for the calling kernel thread.

Return Values

0 No errors occurred.
Positive error code Refer to the pm_error subroutine to decode the error code.

Error Codes

Refer to the pm_error subroutine.

1020 Technical Reference, Volume 1: Base Operating System and Extensions
pm_delete_program_pgroup Subroutine

Purpose
Deletes previously established Performance Monitor settings for the counting group to which a target pthread belongs.

Library
Performance Monitor APIs Library (libpmapi.a)

Syntax
#include <pmapi.h>

int pm_delete_program_pgroup (pid, tid, ptid)

pid_t pid;

Description
The pm_delete_program_pgroup subroutine deletes previously established Performance Monitor settings for a target pthread. The pthread must be stopped and must be part of a debuggee process under the control of the calling process. The settings for the group to which the target pthread belongs and from all the other pthreads in the same group are also deleted.

If the pthread is running in 1:1 mode, only the tid parameter must be specified. If the pthread is running in m:n mode, only the ptid parameter must be specified. If both the ptid and tid parameters are specified, they must be referring to a single pthread with the ptid parameter specified and currently running on a kernel thread with specified tid parameter.

Parameters
pid
Process ID of target thread. The target process must be a debuggee under the control of the calling process.

tid
Thread ID of target pthread. To ignore this parameter, set it to 0.

ptid
Pthread ID of the target pthread. To ignore this parameter, set it to 0.
Return Values

0
No errors occurred.

Positive error code
Refer to the
"pm_error Subroutine" on page 1024 to decode the error code.

Error Codes
Refer to the
"pm_error Subroutine" on page 1024.

Files
/usr/include/pmapi.h
Defines standard macros, data types, and subroutines.

Related Information
The
"pm_delete_program_pgroup Subroutine" on page 1021,
"pm_error Subroutine" on page 1024,
"pm_get_data_pgroup, pm_get_tdata_pgroup and pm_get_Tdata_pgroup Subroutine" on page 1039,
"pm_set_program_pgroup Subroutine" on page 1092,
"pm_initialize Subroutine" on page 1071,
"pm_reset_data_pgroup Subroutine" on page 1076,
"pm_start_pgroup and pm_tstart_pgroup Subroutine" on page 1106,
and the
"pm_stop_pgroup and pm_tstop_pgroup Subroutine " on page 1114.


pm_delete_program_pthread Subroutine

Purpose
Deletes the previously established Performance Monitor settings for a target pthread.

Library
Performance Monitor APIs Library (libpmapi.a)

Syntax

#include <pmapi.h>

int pm_delete_program_pthread ( pid, tid, ptid)

pid_t pid;

tid_t tid;

ptid_t ptid;

Description
The pm_delete_program_pthread subroutine deletes the previously established Performance Monitor settings for a target pthread. The pthread must be stopped and must be part of a debuggee process under the control of the calling process.

If the pthread is running in 1:1 mode, only the tid parameter must be specified. If the pthread is running in m:n mode, only the ptid parameter must be specified. If both the ptid and tid parameters are specified, they must be referring to a single pthread with the ptid parameter specified and currently running on a kernel thread with specified tid parameter.
Parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>pid</td>
<td>Process ID of target pthread. Target process must be a debuggee under the control of the caller process.</td>
</tr>
<tr>
<td>tid</td>
<td>Thread ID of target pthread. To ignore this parameter, set it to 0.</td>
</tr>
<tr>
<td>ptid</td>
<td>Pthread ID of the target pthread. To ignore this parameter, set it to 0.</td>
</tr>
</tbody>
</table>

Return Values

- **0**: No errors occurred.
- **Positive error code**: Refer to the "pm_error Subroutine" on page 1024 to decode the error code.

Error Codes

Refer to the "pm_error Subroutine" on page 1024.

Files

`/usr/include/pmapi.h` Defines standard macros, data types, and subroutines.

Related Information

The "pm_delete_program_pthread Subroutine" on page 1022, "pm_error Subroutine" on page 1024, "pm_get_data_pthread, pm_get_tdata_pthread or pm_get_tdata_pthread Subroutine" on page 1043, "pm_get_program_pthread Subroutine" on page 1063, "pm_initialize Subroutine" on page 1071, "pm_read_data_pthread Subroutine" on page 1078, "pm_set_program_pthread Subroutine" on page 1095, "pm_start_pthread and pm_tstart_pthread Subroutine" on page 1107, "pm_stop_pthread and pm_tstop_pthread Subroutine" on page 1116.


pm_delete_program_thread Subroutine

Purpose

Deletes the previously established Performance Monitor settings for a target thread.

Library

Performance Monitor APIs Library (libpmapi.a)

Syntax

```c
#include <pmapi.h>

int pm_delete_program_thread (pid_t pid, tid_t tid);
```

Description

This subroutine supports only the 1:1 threading model. It has been superseded by the pm_delete_program_pthread subroutine, which supports both the 1:1 and the M:N threading models. A call to this subroutine is equivalent to a call to the pm_delete_program_pthread subroutine with a ptid parameter equal to 0.
The `pm_delete_program_thread` subroutine deletes the previously established Performance Monitor settings for a target kernel thread. The thread must be stopped and must be part of a debuggee process under the control of the calling process.

**Parameters**

- **pid**: Process identifier of target thread. Target process must be a debuggee under the control of the calling process.
- **tid**: Thread identifier of the target thread.

**Return Values**

- **0**: No errors occurred.
- **Positive error code**: Refer to the `pm_error` subroutine to decode the error code.

**Error Codes**

Refer to the `pm_error` subroutine.

**Files**

`/usr/include/pmapi.h`: Defines standard macros, data types, and subroutines.

**Related Information**

- `pm_init`: `pm_init Subroutine` on page 1069
- `pm_error`: `pm_error Subroutine`
- `pm_set_program_thread`: `pm_set_program_thread Subroutine` on page 1098
- `pm_get_program_thread`: `pm_get_program_thread Subroutine` on page 1066
- `pm_get_data_thread`: `pm_get_data_thread, pm_get_tdata_thread or pm_get_Tdata_thread Subroutine` on page 1046
- `pm_start_thread`: `pm_start_thread and pm_tstart_thread Subroutine` on page 1117
- `pm_stop_thread`: `pm_stop_thread and pm_tstop_thread Subroutine` on page 1117
- `pm_reset_data_thread`: `pm_reset_data_thread Subroutine` on page 1079


**pm_error Subroutine**

**Purpose**

Decodes Performance Monitor APIs error codes.

**Library**

Performance Monitor APIs Library (`libpmapi.a`)

**Syntax**

```c
#include <pmapi.h>

void pm_error (char *where, int errorcode);
```

```c
char *where;
int errorcode;
```
Description
The pm_error subroutine writes a message on the standard error output that describes the parameter errorcode encountered by a Performance Monitor API library subroutine. The error message includes the Where parameter string followed by a : (colon), a space character, the message, and a new-line character. The Where parameter string includes the name of the program that caused the error.

Parameters

*Where Specifies where the error was encountered.
errorcode Specifies the error code as returned by one of the Performance Monitor APIs library subroutines.

Files

/usr/include/pmapi.h Defines standard macros, data types, and subroutines.

Related Information
The pm_init subroutine, pm_set_program subroutine, pm_get_program subroutine, pm_delete_program subroutine, pm_get_data subroutine, pm_start subroutine, pm_stop subroutine, pm_reset_data subroutine.

The pm_set_program_mythread subroutine, pm_get_program_mythread subroutine, pm_delete_program_mythread subroutine, pm_get_data_mythread subroutine, pm_start_mythread subroutine, pm_stop_mythread subroutine, pm_reset_data_mythread subroutine.

The pm_set_program_mygroup subroutine, pm_get_program_mygroup subroutine, pm_delete_program_mygroup subroutine, pm_get_data_mygroup subroutine, pm_start_mygroup subroutine, pm_stop_mygroup subroutine, pm_reset_data_mygroup subroutine.

The pm_set_program_thread subroutine, pm_get_program_thread subroutine, pm_delete_program_thread subroutine, pm_get_data_thread subroutine, pm_start_thread subroutine, pm_stop_thread subroutine, pm_reset_data_thread subroutine.

The pm_set_program_group subroutine, pm_get_program_group subroutine, pm_delete_program_group subroutine, pm_get_data_group subroutine, pm_start_group subroutine, pm_stop_group subroutine, pm_reset_data_group subroutine.


Purpose
Returns systemwide Performance Monitor data.

Library
Performance Monitor APIs Library (libpmapi.a)
Syntax

#include <pmapi.h>

int pm_get_data (pmdata_t *pmdata);

int pm_get_tdata (pmdata, time_t *time);

int pm_get_Tdata (pmdata, time_t *times);

int pm_get_data_cpu (cpuid, pmdata_t *pmdata);

int pm_get_tdata_cpu (cpuid, pmdata, time_t *time);

int pm_get_Tdata_cpu (cpuid, pmdata, time_t *times);

int pm_get_data_lcpu (lcpuid, pmdata_t *pmdata);

int pm_get_tdata_lcpu (lcpuid, pmdata, time_t *time);

int pm_get_Tdata_lcpu (lcpuid, pmdata, time_t *times);

Description

The **pm_get_data** subroutine retrieves the current systemwide Performance Monitor data.

The **pm_get_tdata** subroutine retrieves the current systemwide Performance Monitor data, and a timestamp indicating the last time the hardware counters were read.

The **pm_get_Tdata** subroutine retrieves the current systemwide Performance Monitor data, and the accumulated time (timebase, PURR time and SPURR time) the events were counted.

The **pm_get_data_cpu**, **pm_get_tdata_cpu**, and **pm_get_Tdata_cpu** subroutines retrieve the current Performance Monitor data for a specified processor. The given processor ID represents a contiguous number ranging from 0 to \_system\_configuration\_ncpus. These subroutines can only be used when no Dynamic Reconfiguration operations are made on the machine, because when processors are added or removed, the processor numbering is modified and the specified processor number can designate different processors from one call to another. These subroutines are maintained for compatibility with previous versions.

The **pm_get_data_cpu** subroutine retrieves the current Performance Monitor data for the specified processor.
The `pm_get_tdata_cpu` subroutine retrieves the current Performance Monitor data for the specified processor, and a timestamp indicating the last time the hardware counters were read.

The `pm_get_Tdata_cpu` subroutine retrieves the current Performance Monitor data for the specified processor, and the accumulated time (timebase, PURR time and SPURR time) the events were counted.

The `pm_get_data_lcpu`, `pm_get_tdata_lcpu`, and `pm_get_Tdata_lcpu` subroutines retrieve the current Performance Monitor data for a specified logical processor. The given processor ID represents a value ranging from 0 to `_system_configuration.max_ncpus`. This value always represents the same processor, even after Dynamic Reconfiguration operations have occurred. These subroutines might return an error if the specified logical processor number has never run during the counting interval.

The `pm_get_data_lcpu` subroutine retrieves the current Performance Monitor data for the specified logical processor.

The `pm_get_tdata_lcpu` subroutine retrieves the current Performance Monitor data for the specified logical processor, and a timestamp indicating the last time the hardware counters were read.

The `pm_get_Tdata_lcpu` subroutine retrieves the current Performance Monitor data for the specified logical processor, and the accumulated time (timebase, PURR time and SPURR time) the events were counted.

The Performance Monitor data is always a set (one per hardware counter on the machines used) of 64-bit values.

**Parameters**

- `*pmdata`  
  Pointer to a structure that contains the returned systemwide Performance Monitor data.

- `*time`  
  Pointer to a structure containing the timebase value the last time the hardware Performance Monitoring counters were read. This can be converted to time using the `time_base_to_time` subroutine.

- `*times`  
  Pointer to a structure containing the accumulated time (timebase, PURR time and SPURR time) the events were counted. Each time counter can be converted to time using the `time_base_to_time` subroutine.

- `cpuid`  
  Contiguous processor numbers ranging from 0 to `_system_configuration.ncpus`. This value does not always designate the same processor, even after Dynamic Reconfiguration operations have occurred.

- `lcpuid`  
  Logical processor identifier. Each identifier stays linked to a particular processor between reboots, even after Dynamic Reconfiguration operations. This value must be in the range from 0 to `_system_configuration.max_ncpus`.

**Return Values**

- 0  
  Operation completed successfully.

- Positive error code  
  Refer to the `pm_error Subroutine` to decode the error code.

**Error Codes**

Refer to the `pm_error Subroutine`.

**Files**

`/usr/include/papi.h`  
Defines standard macros, data types, and subroutines.
pm_get_data_group, pm_get_tdata_group and pm_get_Tdata_group Subroutine

Purpose

Returns Performance Monitor data for the counting group to which a target thread belongs.

Library

Performance Monitor APIs Library (libpmapi.a)

Syntax

```c
#include <pmapi.h>

int pm_get_data_group (pid_t pid, tid_t tid, pm_data_t *pmdata);

int pm_get_tdata_group (pid_t pid, tid_t tid, pm_data_t *pmdata, timebasestruct_t *time);

int pm_get_Tdata_group (pid_t pid, tid_t tid, pm_data_t *pmdata, pm_accu_time_t *times);
```

Description

These subroutines support only the 1:1 threading model. They have been superseded by the `pm_get_data_pgroup` and `pm_get_tdata_pgroup` subroutines, which support both the 1:1 and the M:N threading models. Calls to these subroutines are equivalent to calls to the `pm_get_data_pgroup` and `pm_get_tdata_pgroup` subroutines with a `ptid` parameter equal to 0.

The `pm_get_data_group` subroutine retrieves the current Performance Monitor data for the counting group to which a target kernel thread belongs. The thread must be stopped and must be part of a debuggee process under the control of the calling process.

The `pm_get_tdata_group` subroutine retrieves the current Performance Monitor data for the counting group to which a target thread belongs, and a timestamp indicating the last time the hardware counters were read.
The **pm_get_Tdata_group** subroutine retrieves the current Performance Monitor data for the counting group to which a target thread belongs, and the accumulated time (timebase, PURR time and SPURR time) the events were counted.

The Performance Monitor data is always a set (one per hardware counter on the machine used) of 64-bit values. The information returned also includes the characteristics of the group, such as the number of its members, if it is a process level group, and if its counters are consistent with the sum of the counters for all of the threads in the group.

**Parameters**

* pid
  Process identifier of a target thread. The target process must be an argument of a debug process.

* tid
  Thread identifier of a target thread.

* pmdata
  Pointer to a structure to return the Performance Monitor data for the group to which the target thread belongs.

* time
  Pointer to a structure containing the timebase value the last time the hardware Performance Monitoring counters were read. This can be converted to time using the **time_base_to_time** subroutine.

* times
  Pointer to a structure containing the accumulated time (timebase, PURR time and SPURR time) the events were counted. Each time counter can be converted to time using the **time_base_to_time** subroutine.

**Return Values**

0
No errors occurred.

Positive error code
Refer to the “pm_error Subroutine” on page 1024 to decode the error code.

**Error Codes**

Refer to the “pm_error Subroutine” on page 1024.

**Files**

/usr/include/pmapi.h
Defines standard macros, data types, and subroutines.

**Related Information**

The **pm_init Subroutine** on page 1069, **pm_error Subroutine** on page 1024, **pm_set_program_group Subroutine** on page 1081, **pm_get_program_group Subroutine** on page 1051, **pm_get_data_group, pm_get_tdata_group and pm_get_Tdata_group Subroutine** on page 1028, **pm_start_group and pm_tstart_group Subroutine** on page 1102, **pm_stop_group and pm_tstop_group Subroutine** on page 1111, **pm_reset_data_group Subroutine** on page 1073.

The **read_real_time or time_base_to_time** subroutine in in AIX 5L Version 5.3 Technical Reference: Base Operating System and Extensions Volume 2.

**pm_get_data_group_mx and pm_get_tdata_group_mx Subroutine**

**Purpose**

Returns Performance Monitor data in counter multiplexing mode for the counting group to which a target thread belongs.

**Library**

Performance Monitor APIs Library (*libpmapi.a*)

**Syntax**

```
#include <pmapi.h>

int pm_get_data_group_mx (pid_t pid, tid_t tid, pm_data_mx_t *pmdata);

int pm_get_tdata_group_mx (pid_t pid, tid_t tid, pm_data_mx_t *pmdata, timebasestruct_t *time);
```

**Description**

These subroutines support only the 1:1 threading model. They have been superseded by the **pm_get_data_pgroup_mx** and **pm_get_tdata_pgroup_mx** subroutines, which support both the 1:1 and the M:N threading models. Calls to these subroutines are equivalent to calls to the **pm_get_data_pgroup_mx** and **pm_get_tdata_pgroup_mx** subroutines with a `ptid` parameter equal to 0.

The **pm_get_data_group_mx** subroutine retrieves the current Performance Monitor data in counter multiplexing mode for the counting group to which a target kernel thread belongs. The thread must be stopped and must be part of a debuggee process under the control of the calling process.

The **pm_get_tdata_group_mx** subroutine retrieves the current Performance Monitor data in counter multiplexing mode for the counting group to which a target thread belongs, and a timestamp indicating the last time the hardware counters were read.

The Performance Monitor data is always an array of a set (one per hardware counter on the machine used) of 64-bit values. The information returned also includes the characteristics of the group, such as the number of its members, whether it is a process level group, and whether its counters are consistent with the sum of the counters for all of the threads in the group.

The user application must free the array allocated to store accumulated counts and times (the `accu_set` field of the `pmdata` parameter).

**Parameters**

- **pid**
  Process identifier of a target thread. The target process must be an argument of a debug process.

- **tid**
  Thread identifier of a target thread.

- **pmdata**
  Pointer to a structure to return the Performance Monitor data (array of accumulated counters, accumulated time and accumulated PURR and SPURR time for each event set counted) for the group to which the target thread belongs.
*time* 
Pointer to a structure containing the timebase value the last time the hardware Performance Monitoring counters were read. This can be converted to time using the `time_base_to_time` subroutine.

**Return Values**

- **0**  
  No errors occurred.
- **Positive error code**  
  Refer to the "pm_error Subroutine" on page 1024 to decode the error code.

**Error Codes**

Refer to the "pm_error Subroutine" on page 1024.

**Files**

`/usr/include/pmapi.h`  
Defines standard macros, data types, and subroutines.

**Related Information**

- "pm_init Subroutine" on page 1069, "pm_error Subroutine" on page 1024,  
- "pm_set_program_group_mx Subroutine" on page 1083, "pm_get_program_group_mx Subroutine" on page 1052, "pm_start_group and pm_tstart_group Subroutine" on page 1102, "pm_stop_group and pm_tstop_group Subroutine" on page 1111, and "pm_reset_data_group Subroutine" on page 1073.

The `read_real_time or time_base_to_time` subroutine is in in *AIX 5L Version 5.3 Technical Reference: Base Operating System and Extensions Volume 2*.

**Performance Monitor API Programming Concepts** in *AIX 5L Version 5.3 Performance Tools Guide and Reference*.

- `pm_get_data_mx`, `pm_get_tdata_mx`, `pm_get_data_cpu_mx`, 
- `pm_get_tdata_cpu_mx`, `pm_get_data_lcpu_mx` and 
- `pm_get_tdata_lcpu_mx` Subroutine

**Purpose**

Returns systemwide Performance Monitor data in counter multiplexing mode.

**Library**

Performance Monitor APIs Library (`libpmapi.a`)

**Syntax**

```c
#include <pmapi.h>

int pm_get_data_mx (pm_data_mx_t *pmdata);
int pm_get_tdata_mx (pm_data_mx_t *pmdata);
int pm_get_data_cpu_mx (cpuid, pm_data_cpu_mx_t *pmdata);
int pm_get_tdata_cpu_mx (pm_data_cpu_mx_t *pmdata);
int pm_get_data_lcpu_mx (pm_data_lcpu_mx_t *pmdata);
int pm_get_tdata_lcpu_mx (pm_data_lcpu_mx_t *pmdata);
```
int pm_get_tdata_cpu_mx (cpuid, *pmdata, *time)
int cpuid;
pm_data_mx_t *pmdata;
timebasestruct_t *time;

int pm_get_data_lcpu_mx (lcpuid, *pmdata)
int lcpuid;
pm_data_mx_t *pmdata;

int pm_get_tdata_lcpu_mx (lcpuid, *pmdata, *time)
int lcpuid;
pm_data_mx_t *pmdata;
timebasestruct_t *time;

Description
The pm_get_data_mx subroutine retrieves the current systemwide Performance Monitor data in counter
multiplexing mode.

The pm_get_tdata_mx subroutine retrieves the current systemwide Performance Monitor data in counter
multiplexing mode, and a timestamp indicating the last time the hardware counters were read.

The pm_get_data_cpu_mx and the pm_get_tdata_cpu_mx subroutines retrieve the current Performance
Monitor data for a specified processor. The given processor ID represents a contiguous number ranging
from 0 to _system_configuration.ncpus. These subroutines can only be used when no Dynamic
Reconfiguration operations are made on the machine, because when processors are added or removed,
the processor numbering is modified and the specified processor number can designate different
processors from one call to another. These subroutines are maintained for compatibility with previous
versions.

The pm_get_data_cpu_mx subroutine retrieves the current Performance Monitor data in counter
multiplexing mode for the specified processor.

The pm_get_tdata_cpu_mx subroutine retrieves the current Performance Monitor data in counter
multiplexing mode for the specified processor, and a timestamp indicating the last time the hardware
counters were read.

The pm_get_data_lcpu_mx and the pm_get_tdata_lcpu_mx subroutines retrieve the current
Performance Monitor data for a specified logical processor. The given processor ID represents a value
ranging from 0 to _system_configuration.max_ncpus. This value always represents the same processor,
even after Dynamic Reconfiguration operations have occurred. These subroutines might return an error if
the specified logical processor number has never run during the counting interval.

The pm_get_data_lcpu_mx subroutine retrieves the current Performance Monitor data for the specified
logical processor in counter multiplexing mode.

The pm_get_tdata_lcpu_mx subroutine retrieves the current Performance Monitor data for the specified
logical processor in counter multiplexing mode, and a timestamp indicating the last time the hardware
counters were read.

The Performance Monitor data is always an array of a set (one per hardware counter on the machines
used) of 64-bit values.

The user application must free the array allocated to store accumulated counts and times (the accu_set
field of the pmdata parameter).
Parameters

*pmdata
  Pointer to a structure that contains the returned systemwide Performance Monitor data. (array of accumulated counters, accumulated time and accumulated PURR and SPURR time for each event set counted)

*time
  Pointer to a structure containing the timebase value the last time the hardware Performance Monitoring counters were read. This can be converted to time using the time_base_to_time subroutine.

cpuid
  Contiguous processor numbers going from 0 to _system_configuration.ncpus. This value does not always designate the same processor, even after Dynamic Reconfiguration operations have occurred.

lcpuid
  Logical processor identifier. Each identifier stays linked to a particular processor between reboots, even after Dynamic Reconfiguration operations. This value must be in the range from 0 to _system_configuration.max_ncpus.

Return Values

0  Operation completed successfully.
Positive error code  Refer to the pm_error Subroutine to decode the error code.

Error Codes

Refer to the pm_error Subroutine.

Files

/usr/include/pmapi.h  Defines standard macros, data types, and subroutines.

Related Information

The pm_init Subroutine on page 1069, pm_error Subroutine on page 1024, pm_set_program_mx Subroutine on page 1085, pm_get_program_mx Subroutine on page 1053, pm_delete_program Subroutine on page 1017, pm_start and pm_tstart Subroutine on page 1101, pm_stop and pm_tstop Subroutine on page 1110, and the pm_reset_data Subroutine on page 1073.

The read_real_time or time_base_to_time subroutine in in AIX 5L Version 5.3 Technical Reference: Base Operating System and Extensions Volume 2.


pm_get_data_mygroup, pm_get_tdata_mygroup or pm_get_Tdata_mygroup Subroutine

Purpose

Returns Performance Monitor data for the counting group to which the calling thread belongs.

Library

Performance Monitor APIs Library (libpmapi.a)
Syntax

```c
#include <pmapi.h>

int pm_get_data_mygroup (*pmdata)
pm_data_t *pmdata;

int pm_get_tdata_mygroup (*pmdata, *time)
pm_data_t *pmdata;
timebasestruct_t *time;

int pm_get_Tdata_mygroup (*pmdata, *times)
pm_data_t *pmdata;
pm_accu_time_t *times;
```

Description

The `pm_get_data_mygroup` subroutine retrieves the current Performance Monitor data for the group to which the calling kernel thread belongs.

The `pm_get_tdata_mygroup` subroutine retrieves the current Performance Monitor data for the group to which the calling thread belongs, and a timestamp indicating the last time the hardware counters were read.

The `pm_get_Tdata_mygroup` subroutine retrieves the current Performance Monitor data for the group to which the calling thread belongs, and the accumulated time (timebase, PURR time and SPURR time) the events were counted.

The Performance Monitor data is always a set (one per hardware counter on the machine used) of 64-bit values. The information returned also includes the characteristics of the group, such as the number of its members, if it is a process level group, and if its counters are consistent with the sum of the counters for all of the threads in the group.

Parameters

* `*pmdata` Pointer to a structure to return the Performance Monitor data for the group to which the calling thread belongs.

* `*time` Pointer to a structure containing the timebase value the last time the hardware Performance Monitoring counters were read. This can be converted to time using the `time_base_to_time` subroutine.

* `*times` Pointer to a structure containing the accumulated time (timebase, PURR time and SPURR time) the events were counted. Each time counter can be converted to time using the `time_base_to_time` subroutine.

Return Values

0 No errors occurred.

Positive error code Refer to the "pm_error Subroutine" on page 1024 to decode the error code.

Error Codes

Refer to the "pm_error Subroutine" on page 1024.

Files

```
/usr/include/pmapi.h
```

Defines standard macros, data types, and subroutines.
Related Information

The "pm_init Subroutine" on page 1069, "pm_error Subroutine" on page 1024, "pm_set_program_mygroup Subroutine" on page 1086, "pm_get_program_mygroup Subroutine" on page 1055, "pm_get_data_mygroup, pm_get_tdata_mygroup or pm_get_Tdata_mygroup Subroutine" on page 1033, "pm_start_mygroup and pm_tstart_mygroup Subroutine" on page 1104, "pm_stop_mygroup and pm_tstop_mygroup Subroutine" on page 1112, "pm_reset_data_mygroup Subroutine" on page 1075.

The read_real_time or time_base_to_time subroutine in in AIX 5L Version 5.3 Technical Reference: Base Operating System and Extensions Volume 2.


pm_get_data_mygroup_mx or pm_get_tdata_mygroup_mx Subroutine

Purpose

Returns Performance Monitor data in counter multiplexing mode for the counting group to which the calling thread belongs.

Library

Performance Monitor APIs Library (libpmapi.a)

Syntax

#include "pmapi.h"

int pm_get_data_mygroup_mx (pmdata)
    pm_data_mx_t *pmdata;

int pm_get_tdata_mygroup_mx (pmdata, time)
    pm_data_mx_t *pmdata;
    timebasestruct_t *time;

Description

The pm_get_data_mygroup_mx subroutine retrieves the current Performance Monitor data in counter multiplexing mode for the group to which the calling kernel thread belongs.

The pm_get_tdata_mygroup_mx subroutine retrieves the current Performance Monitor data in counter multiplexing mode for the group to which the calling thread belongs, and a timestamp indicating the last time the hardware counters were read.

The Performance Monitor data is always an array of set (one per hardware counter on the machine used) of 64-bit values. The information returned also includes the characteristics of the group, such as the number of its members, if it is a process level group, and if its counters are consistent with the sum of the counters for all of the threads in the group.

The user application must free the array allocated to store accumulated counts and times (accu_set field of pmdata).
Parameters

*pmdata

Pointer to a structure to return the Performance Monitor data (array of accumulated counters, accumulated time and accumulated PURR and SPURR time for each event set counted) for the group to which the calling thread belongs.

*time

Pointer to a structure containing the timebase value the last time the hardware Performance Monitoring counters were read. This can be converted to time using the time_base_to_time subroutine.

Return Values

0

No errors occurred.

Positive error code

Refer to the pm_error Subroutine on page 1024 to decode the error code.

Error Codes

Refer to the pm_error Subroutine on page 1024.

Files

/usr/include/pmapi.h

Defines standard macros, data types, and subroutines.

Related Information

The pm_init Subroutine on page 1069, pm_error Subroutine on page 1024, pm_set_program_mygroup_mx Subroutine on page 1087, pm_get_program_mygroup_mx Subroutine on page 1056, pm_start_mygroup and pm_tstart_mygroup Subroutine on page 1104, pm_stop_mygroup and pm_tstop_mygroup Subroutine on page 1112, pm_reset_data_mygroup Subroutine on page 1075.

The read_real_time or time_base_to_time subroutine in in AIX 5L Version 5.3 Technical Reference: Base Operating System and Extensions Volume 2.


pm_get_data_mythread, pm_get_tdata_mythread or pm_get_Tdata_mythread Subroutine

Purpose

Returns Performance Monitor data for the calling thread.

Library

Performance Monitor APIs Library (libpmapi.a)

Syntax

#include <pmapi.h>

int pm_get_data_mythread (*pmdata)

pm_data_t *pmdata;
Description
The \texttt{pm\_get\_data\_mythread} subroutine retrieves the current Performance Monitor data for the calling kernel thread.

The \texttt{pm\_get\_tdata\_mythread} subroutine retrieves the current Performance Monitor data for the calling kernel thread, and a timestamp indicating the last time the hardware counters were read.

The \texttt{pm\_get\_Tdata\_mythread} subroutine retrieves the current Performance Monitor data for the calling kernel thread, and the accumulated time (timebase, PURR time and SPURR time) the events were counted.

The Performance Monitor data is always a set (one per hardware counter on the machine used) of 64-bit values.

Parameters
\begin{itemize}
\item \texttt{*pmdata} \quad Pointer to a structure to contain the returned Performance Monitor data for the calling kernel thread.
\item \texttt{*time} \quad Pointer to a structure containing the timebase value the last time the hardware Performance Monitoring counters were read. This can be converted to time using the \texttt{time\_base\_to\_time} subroutine.
\item \texttt{*times} \quad Pointer to a structure containing the accumulated time (timebase, PURR time and SPURR time) the events were counted. Each time counter can be converted to time using the \texttt{time\_base\_to\_time} subroutine.
\end{itemize}

Return Values
\begin{itemize}
\item 0 \quad No errors occurred.
\item Positive error code \quad Refer to the \texttt{pm\_error Subroutine} on page 1024 to decode the error code.
\end{itemize}

Error Codes
Refer to the \texttt{pm\_error Subroutine} on page 1024.

Files
\begin{itemize}
\item \texttt{/usr/include/pma\_h} \quad Defines standard macros, data types, and subroutines.
\end{itemize}

Related Information
The \texttt{pm\_init Subroutine} on page 1069, \texttt{pm\_error Subroutine} on page 1024, \texttt{pm\_set\_program\_mythread Subroutine} on page 1089, \texttt{pm\_get\_program\_mythread Subroutine} on page 1057, \texttt{pm\_get\_data\_mythread, pm\_get\_tdata\_mythread or pm\_get\_Tdata\_mythread Subroutine} on page 1036, \texttt{pm\_start\_mythread and pm\_tstart\_mythread Subroutine} on page 1105, \texttt{pm\_stop\_mythread and pm\_tstop\_mythread Subroutine} on page 1113, \texttt{pm\_reset\_data\_mythread Subroutine} on page 1076.
pm_get_data_mythread_mx or pm_get_tdata_mythread_mx Subroutine

Purpose
Returns Performance Monitor data in counter multiplexing mode for the calling thread.

Library
Performance Monitor APIs Library (libpmapi.a)

Syntax

#include <pmapi.h>

int pm_get_data_mythread_mx (pmdata)
    pm_data_mx_t *pmdata;

int pm_get_tdata_mythread_mx (*pmdata, *time)
    pm_data_mx_t *pmdata;
    timebasestruct_t *time;

Description
The pm_get_data_mythread_mx subroutine retrieves the current Performance Monitor data in counter multiplexing mode for the calling kernel thread.

The pm_get_tdata_mythread_mx subroutine retrieves the current Performance Monitor data in counter multiplexing mode for the calling kernel thread, and a timestamp indicating the last time the hardware counters were read.

The Performance Monitor data is always an array of a set (one per hardware counter on the machine used) of 64-bit values.

The user application must free the array allocated to store accumulated counts and times (the accu_set field of the pmdata parameter).

Parameters

*pmdata

Pointer to a structure to contain the returned Performance Monitor data (array of accumulated counters, accumulated time and accumulated PURR and SPURR time for each event set counted) for the calling kernel thread.

*time

Pointer to a structure containing the timebase value the last time the hardware Performance Monitoring counters were read. This can be converted to time using the time_base_to_time subroutine.

Return Values

0
No errors occurred.

Positive error code
Refer to the "pm_error Subroutine" on page 1024 to decode the error code.
Error Codes
Refer to the "pm_error Subroutine" on page 1024.

Files
/usr/include/pmapi.h
Defines standard macros, data types, and subroutines.

Related Information
The "pm_init Subroutine" on page 1069, "pm_error Subroutine" on page 1024, "pm_set_program_mythread_mx Subroutine" on page 1090, "pm_get_program_mythread_mx Subroutine" on page 1058, "pm_start_mythread and pm_tstart_mythread Subroutine" on page 1105, "pm_stop_mythread and pm_tstop_mythread Subroutine" on page 1113, "pm_reset_data_mythread Subroutine" on page 1076.

The read_real_time or time_base_to_time subroutine in in AIX 5L Version 5.3 Technical Reference: Base Operating System and Extensions Volume 2.


pm_get_data_pgroup, pm_get_tdata_pgroup and pm_get_Tdata_pgroup Subroutine

Purpose
Returns Performance Monitor data for the counting group to which a target pthread belongs.

Library
Performance Monitor APIs Library (libpmapi.a)

Syntax
#include <pmapi.h>

int pm_get_data_pgroup (pid, tid, ptid, pmdata)
    pid_t pid;
    tid_t tid;
    ptid_t ptid;
    pm_data_t *pmdata;

int pm_get_tdata_pgroup (pid, tid, pmdata, time)
    pm_data_t *pmdata;
    pid_t pid;
    tid_t tid;
    ptid_t ptid;
    timebasestruct_t *time;

int pm_get_Tdata_pgroup (pid, tid, pmdata, times)
    pm_data_t *pmdata;
    pid_t pid;
    tid_t tid;
    ptid_t ptid;
    pm_accu_time_t *times;
Description
The `pm_get_data_pgroup` subroutine retrieves the current Performance Monitor data for the counting group to which a target pthread belongs. The pthread must be stopped and must be part of a debuggee process under the control of the calling process.

The `pm_get_tdata_pgroup` subroutine retrieves the current Performance Monitor data for the counting group to which a target pthread belongs, and a timestamp indicating the last time the hardware counters were read.

The `pm_get_Tdata_pgroup` subroutine retrieves the current Performance Monitor data for the counting group to which a target pthread belongs, and the accumulated time (timebase, PURR time and SPURR time) the events were counted.

If the pthread is running in 1:1 mode, only the `tid` parameter must be specified. If the pthread is running in m:n mode, only the `ptid` parameter must be specified. If both the `ptid` and `tid` parameters are specified, they must be referring to a single pthread with the `ptid` parameter specified and currently running on a kernel thread with specified `tid` parameter.

The Performance Monitor data is always a set (one per hardware counter on the machine used) of 64-bit values. The information returned also includes the characteristics of the group, such as the number of its members, if it is a process level group, and if its counters are consistent with the sum of the counters for all of the pthreads in the group.

Parameters

- **pid**: Process identifier of a target thread. The target process must be an argument of a debug process.
- **tid**: Thread ID of target pthread. To ignore this parameter, set it to 0.
- **ptid**: Pthread ID of the target pthread. To ignore this parameter, set it to 0.
- **pmdata**: Pointer to a structure to return the Performance Monitor data for the group to which the target pthread belongs.
- **time**: Pointer to a structure containing the timebase value the last time the hardware Performance Monitoring counters were read. This can be converted to time using the `time_base_to_time` subroutine.
- **times**: Pointer to a structure containing the accumulated time (timebase, PURR time and SPURR time) the events were counted. Each time counter can be converted to time using the `time_base_to_time` subroutine.

Return Values

- **0**: No errors occurred.
- **Positive error code**: Refer to the "pm_error Subroutine" on page 1024 to decode the error code.

Error Codes
Refer to the "pm_error Subroutine" on page 1024.

Files

- `/usr/include/pmapi.h`: Defines standard macros, data types, and subroutines.
Related Information

The `pm_delete_program_pthread Subroutine` on page 1022, `pm_error Subroutine` on page 1024, `pm_get_data_pgroup, pm_get_tdata_pgroup and pm_get_Tdata_pgroup Subroutine` on page 1039, `pm_get_program_pgroup Subroutine` on page 1060, `pm_initialize Subroutine` on page 1071, `pm_reset_data_pgroup Subroutine` on page 1076, `pm_set_program_pgroup Subroutine` on page 1092, `pm_start_pgroup and pm_tstart_pgroup Subroutine` on page 1106, `pm_stop_pgroup and pm_tstop_pgroup Subroutine` on page 1114.

The `read_real_time or time_base_to_time` subroutine in in AIX 5L Version 5.3 Technical Reference: Base Operating System and Extensions Volume 2.


---

**pm_get_data_pgroup_mx and pm_get_tdata_pgroup_mx Subroutine**

**Purpose**

Returns Performance Monitor data in counter multiplexing mode for the counting group to which a target pthread belongs.

**Library**

Performance Monitor APIs Library (libpmapi.a)

**Syntax**

```c
#include <pmapi.h>

int pm_get_data_pgroup_mx (pid, tid, ptid, *pmdata)
    pid_t pid;
    tid_t tid;
    ptid_t ptid;
    pm_data_mx_t *pmdata;

int pm_get_tdata_pgroup_mx (pid, tid, *pmdata, *time)
    pm_data_mx_t *pmdata;
    timebasestruct_t *time;
```

**Description**

The `pm_get_data_pgroup_mx` subroutine retrieves the current Performance Monitor data in counter multiplexing mode for the counting group to which a target pthread belongs. The pthread must be stopped and must be part of a debuggee process under the control of the calling process.

The `pm_get_tdata_pgroup_mx` subroutine retrieves the current Performance Monitor data in counter multiplexing mode for the counting group to which a target pthread belongs, and a timestamp indicating the last time the hardware counters were read.

If the pthread is running in 1:1 mode, only the `tid` parameter must be specified. If the pthread is running in m:n mode, only the `ptid` parameter must be specified. If both the `ptid` and `tid` parameters are specified, they must be referring to a single pthread with the `ptid` parameter specified and currently running on a kernel thread with specified `tid` parameter.

The Performance Monitor data is always an array of a set (one per hardware counter on the machine used) of 64-bit values. The information returned also includes the characteristics of the group, such as the
number of its members, whether it is a process level group, and whether its counters are consistent with the sum of the counters for all of the pthreads in the group.

The user application must free the array allocated to store accumulated counts and times (the accu_set field of the pmdata parameter).

**Parameters**

- **pid**
  Process identifier of a target thread. The target process must be an argument of a debug process.

- **tid**
  Thread ID of target pthread. To ignore this parameter, set it to 0.

- **ptid**
  Pthread ID of the target pthread. To ignore this parameter, set it to 0.

- **pmdata**
  Pointer to a structure to return the Performance Monitor data (array of accumulated counters, accumulated time and accumulated PURR and SPURR time for each event set counted) for the group to which the target pthread belongs.

- **time**
  Pointer to a structure containing the timebase value the last time the hardware Performance Monitoring counters were read. This can be converted to time using the time_base_to_time subroutine.

**Return Values**

- **0**
  No errors occurred.

- **Positive error code**
  Refer to the "pm_error Subroutine" on page 1024 to decode the error code.

**Error Codes**

Refer to the "pm_error Subroutine" on page 1024.

**Files**

/usr/include/pmafi.h

Defines standard macros, data types, and subroutines.

**Related Information**

The "pm_delete_program_pthread Subroutine" on page 1022, "pm_error Subroutine" on page 1024, "pm_get_program_pgroup_mx Subroutine" on page 1061, "pm_initialize Subroutine" on page 1071, "pm_reset_data_pgroup Subroutine" on page 1076, "pm_set_program_pgroup_mx Subroutine" on page 1093, "pm_start_pgroup and pm_tstart_pgroup Subroutine" on page 1106, "pm_stop_pgroup and pm_tstop_pgroup Subroutine " on page 1114.

The **read_real_time or time_base_to_time** subroutine in in AIX 5L Version 5.3 Technical Reference: Base Operating System and Extensions Volume 2.

**pm_get_data_pthread, pm_get_tdata_pthread or pm_get_Tdata_pthread Subroutine**

**Purpose**
Returns Performance Monitor data for a target pthread.

**Library**
Performance Monitor APIs Library (libpmapi.a)

**Syntax**
```
#include <pmapi.h>

int pm_get_data_pthread (pid, tid, ptid, *pmdata)
pid_t pid;
tid_t tid;
ptid_t ptid;
pm_data_t *pmdata;

int pm_get_tdata_pthread (pid, tid, ptid, *pmdata, *time)
pid_t pid;
tid_t tid;
ptid_t ptid;
pm_data_t *pmdata;
timebasestruct_t *time;

int pm_get_Tdata_pthread (pid, tid, ptid,*pmdata, *times)
pid_t pid;
tid_t tid;
ptid_t ptid;
pm_data_t *pmdata;
pm_accu_time_t *times;
```

**Description**
The `pm_get_data_pthread` subroutine retrieves the current Performance Monitor data for a target pthread. The pthread must be stopped and must be part of a debuggee process under the control of a calling process.

The `pm_get_tdata_pthread` subroutine retrieves the current Performance Monitor data for a target pthread, and a timestamp indicating the last time the hardware counters were read.

The `pm_get_Tdata_pthread` subroutine retrieves the current Performance Monitor data for a target pthread, and the accumulated time (timebase, PURR time and SPURR time) the events were counted.

If the pthread is running in 1:1 mode, only the `tid` parameter must be specified. If the pthread is running in m:n mode, only the `ptid` parameter must be specified. If both the `ptid` and `tid` parameters are specified, they must be referring to a single pthread with the `ptid` parameter specified and currently running on a kernel thread with specified `tid` parameter.

The Performance Monitor data is always a set (one per hardware counter on the machine used) of 64-bit values.

**Parameters**

*pid*  
Process ID of target pthread. Target process must be a debuggee of the caller process.
**tid**
Thread ID of target pthread. To ignore this parameter, set it to 0.

**ptid**
Pthread ID of the target pthread. To ignore this parameter, set it to 0.

**pmdata**
Pointer to a structure to return the Performance Monitor data for the target pthread.

**time**
Pointer to a structure containing the timebase value the last time the hardware Performance Monitoring counters were read. This can be converted to time using the `time_base_to_time` subroutine.

**times**
Pointer to a structure containing the accumulated time (timebase, PURR time and SPURR time) the events were counted. Each time counter can be converted to time using the `time_base_to_time` subroutine.

### Return Values

<table>
<thead>
<tr>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>No errors occurred.</td>
</tr>
<tr>
<td>Positive</td>
<td>Positive error code</td>
</tr>
</tbody>
</table>

Refer to the "pm_error Subroutine" on page 1024 to decode the error code.

### Error Codes

Refer to the "pm_error Subroutine" on page 1024.

### Files

`/usr/include/pmapi.h` Defines standard macros, data types, and subroutines.

### Related Information

The "pm_get_program_pthread Subroutine" on page 1063, "pm_get_data_pthread, pm_get_tdata_pthread or pm_get_Tdata_pthread Subroutine" on page 1043, "pm_get_program_pthread Subroutine" on page 1063, "pm_initialize Subroutine" on page 1071, "pm_set_program_pthread Subroutine" on page 1095, "pm_get_data_pthread_mx or pm_get_tdata_pthread_mx Subroutine" on page 1043, "pm_set_program_pthread Subroutine" on page 1095, "pm_delete_program_pthread Subroutine" on page 1022, "pm_error Subroutine" on page 1024, "pm_get_data_pthread Subroutine" on page 1022, "pm_get_tdata_pthread Subroutine" on page 1024.

The `read_real_time or time_base_to_time` subroutine in in AIX 5L Version 5.3 Technical Reference: Base Operating System and Extensions Volume 2.


### pm_get_data_pthread_mx or pm_get_tdata_pthread_mx Subroutine

**Purpose**

Returns Performance Monitor data in counter multiplexing mode for a target pthread.

**Library**

Performance Monitor APIs Library (libpmapi.a)
Syntax

```c
#include <pmapi.h>

int pm_get_data_pthread_mx (pid_t pid, tid_t tid, ptid_t ptid, pmdata_mx_t *pmdata)

pid_t pid;
tid_t tid;
ptid_t ptid;
pm_data_mx_t *pmdata;

int pm_get_tdata_pthread_mx (pid_t pid, tid_t tid, ptid_t ptid, pmdata_mx_t *pmdata, timebasestruct_t *time)

pid_t pid;
tid_t tid;
ptid_t ptid;
pm_data_mx_t *pmdata;
timebasestruct_t *time;
```

Description

The `pm_get_data_pthread_mx` subroutine retrieves the current Performance Monitor data in counter multiplexing mode for a target pthread. The pthread must be stopped and must be part of a debuggee process under the control of a calling process.

The `pm_get_tdata_pthread_mx` subroutine retrieves the current Performance Monitor data in counter multiplexing mode for a target pthread, and a timestamp indicating the last time the hardware counters were read.

If the pthread is running in 1:1 mode, only the `tid` parameter must be specified. If the pthread is running in m:n mode, only the `ptid` parameter must be specified. If both the `ptid` and `tid` parameters are specified, they must be referring to a single pthread with the `ptid` parameter specified and currently running on a kernel thread with specified `tid` parameter.

The Performance Monitor data is always an array of a set (one per hardware counter on the machine used) of 64-bit values.

The user application must free the array allocated to store accumulated counts and times (the accu_set field of the `pmdata` parameter).

Parameters

- `pid` Process ID of target pthread. Target process must be a debuggee of the caller process.
- `tid` Thread ID of target pthread. To ignore this parameter, set it to 0.
- `ptid` Pthread ID of the target pthread. To ignore this parameter, set it to 0.
- `pmdata` Pointer to a structure to return the Performance Monitor data (array of accumulated counters, accumulated time and accumulated PURR and SPURR time for each event set counted) for the target pthread.
- `time` Pointer to a structure containing the timebase value the last time the hardware Performance Monitoring counters were read. This can be converted to time using the `time_base_to_time` subroutine.

Return Values

<table>
<thead>
<tr>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>No errors occurred.</td>
</tr>
<tr>
<td>Positive error code</td>
<td>Refer to the <a href="#">“pm_error Subroutine” on page 1024</a> to decode the error code.</td>
</tr>
</tbody>
</table>
Error Codes
Refer to the "pm_error Subroutine" on page 1024.

Files
/usr/include/pmapi.h Defines standard macros, data types, and subroutines.

Related Information
The "pm_delete_program_pthread Subroutine" on page 1022, "pm_error Subroutine" on page 1024, "pm_get_program_pthread mx Subroutine" on page 1064, "pm_initialize Subroutine" on page 1071, "pm_reset_data_pthread Subroutine" on page 1078, "pm_set_program_pthread_mx Subroutine" on page 1097, "pm_start_pthread and pm_tstart_pthread Subroutine" on page 1107, "pm_stop_pthread and pm_tstop_pthread Subroutine" on page 1116.

The read_real_time or time_base_to_time subroutine in in AIX 5L Version 5.3 Technical Reference: Base Operating System and Extensions Volume 2.


pm_get_data_thread, pm_get_tdata_thread or pm_get_Tdata_thread Subroutine

Purpose
Returns Performance Monitor data for a target thread.

Library
Performance Monitor APIs Library (libpmapi.a)

Syntax

```c
#include <pmapi.h>

int pm_get_data_thread (pid_t pid, tid_t tid, pm_data_t *pmdata);

int pm_get_tdata_thread (pid_t pid, tid_t tid, pm_data_t *pmdata, timebasestruct_t *time);

int pm_get_Tdata_thread (pid_t pid, tid_t tid, pm_data_t *pmdata, pm_accu_time_t *times);
```

**Description**
These subroutines support only the 1:1 threading model. They have been superseded by the `pm_get_data_pthread` and `pm_get_tdata_pthread` subroutines, which support both the 1:1 and the M:N threading models. Calls to these subroutines are equivalent to calls to the `pm_get_data_pthread` and `pm_get_tdata_pthread` subroutines with a `ptid` parameter equal to 0.

The `pm_get_data_thread` subroutine retrieves the current Performance Monitor data for a target kernel thread. The thread must be stopped and must be part of a debuggee process under the control of a calling process.

The `pm_get_tdata_thread` subroutine retrieves the current Performance Monitor data for a target thread, and a timestamp indicating the last time the hardware counters were read.

The `pm_get_Tdata_thread` subroutine retrieves the current Performance Monitor data for a target thread, and the accumulated time (timebase, PURR time and SPURR time) the events were counted.

The Performance Monitor data is always a set (one per hardware counter on the machine used) of 64-bit values.

**Parameters**

- **pid**
  Process identifier of a target thread. The target process must be a debuggee of the caller process.

- **tid**
  Thread identifier of a target thread.

- **pmdata**
  Pointer to a structure to return the Performance Monitor data for the target kernel thread.

- **time**
  Pointer to a structure containing the timebase value the last time the hardware Performance Monitoring counters were read. This can be converted to time using the `time_base_to_time` subroutine.

- **times**
  Pointer to a structure containing the accumulated time (timebase, PURR time and SPURR time) the events were counted. Each time counter can be converted to time using the `time_base_to_time` subroutine.

**Return Values**

- **0**
  No errors occurred.

- **Positive error code**
  Refer to the ["pm_error Subroutine" on page 1024](#) to decode the error code.

**Error Codes**
Refer to the ["pm_error Subroutine" on page 1024](#).

**Files**

- `/usr/include/pmapi.h`
  Defines standard macros, data types, and subroutines.

**Related Information**
The ["pm_init Subroutine" on page 1069](#), ["pm_error Subroutine" on page 1024](#), ["pm_set_program_thread Subroutine" on page 1098](#), ["pm_get_program_thread Subroutine" on page 1066](#), ["pm_get_data_thread Subroutine" on page 1046](#), ["pm_start_thread and pm_get_data_pthread or pm_get_tdata_thread Subroutine" on page 1046](#).
The `pm_get_data_thread_mx` subroutine retrieves the current Performance Monitor data in counter multiplexing mode for a target kernel thread. The thread must be stopped and must be part of a debuggee process under the control of a calling process.

The `pm_get_tdata_thread_mx` subroutine retrieves the current Performance Monitor data in counter multiplexing mode for a target thread, and a timestamp indicating the last time the hardware counters were read.

The Performance Monitor data is always an array of a set (one per hardware counter on the machine used) of 64-bit values.

The user application must free the array allocated to store accumulated counts and times (the accu_set field of the `pmdata` parameter).

**Parameters**

- **pid**: Process identifier of a target thread. The target process must be a debuggee of the calling process.
tid
*pmdata
*time

Thread identifier of a target thread.
Pointer to a structure to return the Performance Monitor
data (array of accumulated counters, accumulated time
and accumulated PURR and SPURR time for each event
set counted) for the target kernel thread.
Pointer to a structure containing the timebase value the
last time the hardware Performance Monitoring counters
were read. This can be converted to time using the
time_base_to_time subroutine.

Return Values

0 No errors occurred.
Positive error code Refer to the pm_error Subroutine on page 1024 to decode the error code.

Error Codes
Refer to the pm_error Subroutine on page 1024.

Files
/usr/include/pmapi.h Defines standard macros, data types, and subroutines.

Related Information
The pm_init Subroutine on page 1069, pm_error Subroutine on page 1024,
pm_set_program_thread_mx Subroutine on page 1100, pm_get_program_thread_mx Subroutine on
page 1067, pm_start_thread and pm_tstart_thread Subroutine on page 1109, pm_stop_thread and
pm_tstop_thread Subroutine on page 1117, pm_reset_data_thread Subroutine on page 1079.

The read_real_time or time_base_to_time subroutine in in AIX 5L Version 5.3 Technical Reference:
Base Operating System and Extensions Volume 2.

Performance Monitor API Programming Concepts in AIX 5L Version 5.3 Performance Tools Guide and
Reference.

pm_get_program Subroutine

Purpose
Retrieves systemwide Performance Monitor settings.

Library
Performance Monitor APIs Library (libpmapi.a)

Syntax
#include <pmapi.h>

int pm_get_program ( *prog,
    pm_prog_t *prog;
Description

The `pm_get_program` subroutine retrieves the current systemwide Performance Monitor settings. This includes mode information and the events being counted, which are in a list of event identifiers. The identifiers come from the lists returned by the `pm_init` subroutine.

The counting mode includes user mode, the kernel mode, the current counting state, and the process tree mode. If the process tree mode is on, the counting applies only to the calling process and its descendants.

If the list includes an event which can be used with a threshold (as indicated by the `pm_init` subroutine), a threshold value is also returned.

If the events are represented by a group ID, then the `is_group` bit is set in the mode, and the first element of the events array contains the group ID. The other elements of the events array are not meaningful.

Parameters

`prog`  
Returns which Performance Monitor events and modes are set. Supported modes are:

- **PM_USER**  
  Counting processes running in user mode
- **PM_KERNEL**  
  Counting processes running in kernel mode
- **PM_COUNT**  
  Counting is on
- **PM_PROCTREE**  
  Counting applies only to the calling process and its descendants

Return Values

<table>
<thead>
<tr>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>No errors occurred.</td>
</tr>
<tr>
<td>Positive error code</td>
<td>Refer to the <a href="#">pm_error Subroutine</a> on page 1024 to decode the error code.</td>
</tr>
</tbody>
</table>

Error Codes

Refer to the [pm_error Subroutine](#) on page 1024.

Files

`/usr/include/pmapi.h`  
Defines standard macros, data types, and subroutines.

Related Information

The `pm_init Subroutine` on page 1069, `pm_error Subroutine` on page 1024, `pm_set_program Subroutine` on page 1080, `pm_delete_program Subroutine` on page 1017, `pm_get_data, pm_get_tdata, pm get Tdata, pm get data cpu, pm get tdata cpu, pm get Tdata cpu, pm get data lcpu, pm get tdata lcpu and pm get _Tdata lcpu Subroutine` on page 1025, `pm_start and pm_tstart Subroutine` on page 1101, `pm_stop and pm_tstop Subroutine` on page 1110, `pm_reset_data Subroutine` on page 1073.

pm_get_program_group Subroutine

Purpose
Retrieves the Performance Monitor settings for the counting group to which a target thread belongs.

Library
Performance Monitor APIs Library (libpmapi.a)

Syntax
#include <pmapi.h>

int pm_get_program_group ( pid, tid, *prog

pid_t pid;
int tid;
pm_prog_t *prog;

Description
This subroutine supports only the 1:1 threading model. It has been superseded by the pm_get_program_pgroup subroutine, which supports both the 1:1 and the M:N threading models. A call to this subroutine is equivalent to a call to the pm_get_program_pgroup subroutine with a ptid parameter equal to 0.

The pm_get_program_group subroutine retrieves the Performance Monitor settings for the counting group to which a target kernel thread belongs. The thread must be stopped and must be part of a debuggee process under the control of the calling process. This includes mode information and the events being counted, which are in a list of event identifiers. The identifiers come from the lists returned by the pm_init subroutine.

The counting mode includes the user mode and kernel mode, and the current counting state.

If the list includes an event which can be used with a threshold (as indicated by the pm_init subroutine), a threshold value is also returned.

Parameters

pid
Process identifier of target thread. The target process must be an argument of a debug process.

tid
Thread identifier of the target thread.

prog
Returns which Performance Monitor events and modes are set. Supported modes are:

PM_USER
Counting process running in user mode

PM_KERNEL
Counting process running kernel mode

PM_COUNT
Counting is on

PM_PROCESS
Process level counting group
Return Values

0
No errors occurred.

Positive error code
Refer to the pm_error Subroutine on page 1024 to decode the error code.

Error Codes
Refer to the pm_error Subroutine on page 1024.

Files

/usr/include/pmapi.h
Defines standard macros, data types, and subroutines.

Related Information
The pm_init Subroutine on page 1069, pm_error Subroutine on page 1024, pm_set_program_group Subroutine on page 1081, pm_delete_program_group Subroutine on page 1018, pm_get_data_group, pm_get_tdata_group and pm_get_Tdata_group Subroutine on page 1028, pm_start_group and pm_tstart_group Subroutine on page 1102, pm_stop_group and pm_tstop_group Subroutine on page 1111, pm_reset_data_group Subroutine on page 1073.


pm_get_program_group_mx Subroutine

Purpose
Retrieves the Performance Monitor settings in counter multiplexing mode for the counting group to which a target thread belongs.

Library
Performance Monitor APIs Library (libpmapi.a)

Syntax

#include <pmapi.h>

int pm_get_program_group_mx ( pid_t pid, tid_t tid, pm_prog_mx_t *prog);

Description
This subroutine supports only the 1:1 threading model. It has been superseded by the pm_get_program_pgroup_mx subroutine, which supports both the 1:1 and the M:N threading models. A call to this subroutine is equivalent to a call to the pm_get_program_pgroup_mx subroutine with a ptid parameter equal to 0.

The pm_get_program_group_mx subroutine retrieves the Performance Monitor settings for the counting group to which a target kernel thread belongs. The thread must be stopped and must be part of a debuggee process under the control of the calling process. This includes mode information and the events being counted, which are in an array of list of event identifiers. The identifiers come from the lists returned by the pm_initialize subroutine.
The counting mode includes the user mode and kernel mode, and the current counting state.

If the list includes an event which can be used with a threshold (as indicated by the \texttt{pm_init} subroutine), a threshold value is also returned.

The user application must free the array allocated to store the event lists (the events_set field in the \textit{prog} parameter).

**Parameters**

\begin{itemize}
  \item \textit{pid} \hspace{2cm} \text{Process identifier of target thread. The target process must be an argument of a debug process.}
  \item \textit{tid} \hspace{2cm} \text{Thread identifier of the target thread.}
  \item \textit{*prog} \hspace{2cm} \text{Returns which Performance Monitor events and modes are set. Supported modes are:}
  \begin{itemize}
    \item \texttt{PM\_USER} \hspace{2cm} \text{Counting process running in user mode}
    \item \texttt{PM\_KERNEL} \hspace{2cm} \text{Counting process running kernel mode}
    \item \texttt{PM\_COUNT} \hspace{2cm} \text{Counting is on}
    \item \texttt{PM\_PROCESS} \hspace{2cm} \text{Process level counting group}
  \end{itemize}
\end{itemize}

**Return Values**

\begin{itemize}
  \item 0 \hspace{2cm} \text{No errors occurred.}
  \item Positive error code \hspace{2cm} \text{Refer to the \textit{pm_error Subroutine} on page 1024 to decode the error code.}
\end{itemize}

**Error Codes**

Refer to the \textit{pm_error Subroutine} on page 1024.

**Files**

\texttt{/usr/include/pmapi.h} \hspace{2cm} Defines standard macros, data types, and subroutines.

**Related Information**

The \textit{pm_init Subroutine} on page 1069, \textit{pm_error Subroutine} on page 1024, \textit{pm_set_program_group_mx Subroutine} on page 1083, \textit{pm_delete_program_group Subroutine} on page 1018, \textit{pm_get_data_group_mx and pm_get_tdata_group_mx Subroutine} on page 1030, \textit{pm_start_group and pm_tstart_group Subroutine} on page 1102, \textit{pm_stop_group and pm_tstop_group Subroutine} on page 1111, \textit{pm_reset_data_group Subroutine} on page 1073.


\textit{pm_get_program_mx Subroutine}

**Purpose**

Retrieves systemwide Performance Monitor settings in counter multiplexing mode.
Library
Performance Monitor APIs Library (libpmapi.a)

Syntax
```c
#include <pmapi.h>

int pm_get_program_mx ( pm_prog_mx_t *prog );
```

Description
The `pm_get_program_mx` subroutine retrieves the current systemwide Performance Monitor settings. This includes mode information and the events being counted, which are in an array of list of event identifiers. The identifiers come from the lists returned by the `pm_initialize` subroutine.

The counting mode includes user mode, the kernel mode, the current counting state, and the process tree mode. If the process tree mode is on, the counting applies only to the calling process and its descendents.

If the list includes an event which can be used with a threshold (as indicated by the `pm_init` subroutine), a threshold value is also returned.

If the events are represented by a group ID, then the `is_group` bit is set in the mode, and the first element of each events array contains the group ID. The other elements of the events array are not meaningful.

The user application must free the array allocated to store the event lists (events_set field in prog).

Parameters
`prog` Returns which Performance Monitor events and modes are set. Supported modes are:

- **PM_USER** Counting processes running in user mode
- **PM_KERNEL** Counting processes running in kernel mode
- **PM_COUNT** Counting is on
- **PM_PROCTREE** Counting applies only to the calling process and its descendents

Return Values

<table>
<thead>
<tr>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>No errors occurred.</td>
</tr>
<tr>
<td>Positive error code</td>
<td>Refer to the <code>pm_error</code> subroutine to decode the error code.</td>
</tr>
</tbody>
</table>

Error Codes
Refer to the `pm_error` subroutine.

Files
```
/usr/include/pmapi.h
```
Defines standard macros, data types, and subroutines.
Related Information
The "pm_init Subroutine" on page 1069, "pm_error Subroutine" on page 1024, "pm_set_program_mx Subroutine" on page 1085, "pm_delete_program Subroutine" on page 1017, "pm_get_data_mx, pm_get_tdata_mx, pm_get_data_cpu_mx, pm_get_tdata_cpu_mx, pm_get_data_lcpu_mx and pm_get_tdata_lcpu_mx Subroutine" on page 1031, "pm_start and pm_tstart Subroutine" on page 1101, "pm_stop and pm_tstop Subroutine" on page 1110, "pm_reset_data Subroutine" on page 1073.


pm_get_program_mygroup Subroutine

Purpose
Retrieves the Performance Monitor settings for the counting group to which the calling thread belongs.

Library
Performance Monitor APIs Library (libpmapi.a)

Syntax
#include <pmapi.h>

int pm_get_program_mygroup ( pm_prog_t *prog);

Description
The pm_get_program_mygroup subroutine retrieves the Performance Monitor settings for the counting group to which the calling kernel thread belongs. This includes mode information and the events being counted, which are in a list of event identifiers. The identifiers come from the lists returned by the pm_init subroutine.

The counting mode includes user mode and kernel mode, and the current counting state.

If the list includes an event which can be used with a threshold (as indicated by the pm_init subroutine), a threshold value is also returned.

Parameters

*prog

Returns which Performance Monitor events and modes are set. Supported modes are:

PM_USER
Counting processes running in user mode

PM_KERNEL
Counting processes running in kernel mode

PM_COUNT
Counting is on

PM_PROCESS
Process level counting group
Return Values

0                      No errors occurred.

Positive error code    Refer to the pm_error Subroutine on page 1024 to decode the error code.

Error Codes

Refer to the pm_error Subroutine on page 1024.

Files

/usr/include/pmapi.h    Defines standard macros, data types, and subroutines.

Related Information

The pm_init Subroutine on page 1069, pm_error Subroutine on page 1024, pm_set_program_mygroup
Subroutine on page 1086, pm_delete_program_mygroup Subroutine on page 1019,
pm_get_data_mygroup, pm_get_tdata_mygroup or pm_get_Tdata_mygroup Subroutine on page 1033,
pm_start_mygroup and pm_tstart_mygroup Subroutine on page 1104, pm_stop_mygroup and
pm_tstop_mygroup Subroutine on page 1112, pm_reset_data_mygroup Subroutine on page 1075.

Performance Monitor API Programming Concepts in AIX 5L Version 5.3 Performance Tools Guide and
Reference.

pm_get_program_mygroup_mx Subroutine

Purpose

Retrieves the Performance Monitor settings in counter multiplexing mode for the counting group to which
the calling thread belongs.

Library

Performance Monitor APIs Library (libpmapi.a)

Syntax

#include <pmapi.h>

int pm_get_program_mygroup_mx ( pmprog
pm_prog_mx_t *prog;

Description

The pm_get_program_mygroup_mx subroutine retrieves the Performance Monitor settings for the
counting group to which the calling kernel thread belongs. This includes mode information and the events
being counted, which are in an array of list of event identifiers. The identifiers come from the lists returned
by the pm_initialize subroutine.

The counting mode includes user mode and kernel mode, and the current counting state.

If the list includes an event which can be used with a threshold (as indicated by the pm_init subroutine), a
threshold value is also returned.

The user application must free the array allocated to store the event lists ( the events_set field in the prog
parameter).
Parameters

*prog

Returns which Performance Monitor events and modes are set. Supported modes are:

- **PM_USER**
  Counting processes running in user mode

- **PM_KERNEL**
  Counting processes running in kernel mode

- **PM_COUNT**
  Counting is on

- **PM_PROCESS**
  Process level counting group

Return Values

<table>
<thead>
<tr>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>No errors occurred.</td>
</tr>
<tr>
<td>Positive Error Code</td>
<td>Refer to the “pm_error Subroutine” on page 1024 to decode the error code.</td>
</tr>
</tbody>
</table>

Error Codes

Refer to the “pm_error Subroutine” on page 1024.

Files

/usr/include/pmpi.h

Defines standard macros, data types, and subroutines.

Related Information

The “pm_init Subroutine” on page 1069, “pm_error Subroutine” on page 1024, “pm_set_program_mygroup_mx Subroutine” on page 1087, “pm_delete_program_mygroup Subroutine” on page 1019, “pm_get_data_mygroup_mx or pm_get_tdata_mygroup_mx Subroutine” on page 1035, “pm_start_mygroup and pm_tstart_mygroup Subroutine” on page 1104, “pm_stop_mygroup and pm_tstop_mygroup Subroutine” on page 1112, “pm_reset_data_mygroup Subroutine” on page 1075.


pm_get_program_mythread Subroutine

Purpose

Retrieves the Performance Monitor settings for the calling thread.

Library

Performance Monitor APIs Library (libpmapi.a)

Syntax

```c
#include <pmpi.h>

int pm_get_program_mythread ( *prog
pm_prog_t *prog;```

Base Operating System (BOS) Runtime Services (A-P) 1057
Description

The `pm_get_program_mythread` subroutine retrieves the Performance Monitor settings for the calling kernel thread. This includes mode information and the events being counted, which are in a list of event identifiers. The identifiers come from the lists returned by the `pm_init` subroutine.

The counting mode includes user mode and kernel mode, and the current counting state.

If the list includes an event which can be used with a threshold (as indicated by the `pm_init` subroutine), a threshold value is also returned.

Parameters

*`prog`*  
Returns which Performance Monitor events and modes are set. Supported modes are:

- **PM_USER**  
  Counting processes running in user mode

- **PM_KERNEL**  
  Counting processes running in kernel mode

- **PM_COUNT**  
  Counting is on

Return Values

<table>
<thead>
<tr>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>No errors occurred.</td>
</tr>
<tr>
<td>Positive error code</td>
<td>Refer to the [pm_error Subroutine] on page 1024 to decode the error code.</td>
</tr>
</tbody>
</table>

Error Codes

Refer to the [pm_error Subroutine] on page 1024.

Files

[/usr/include/pmapi.h](/usr/include/pmapi.h)  
Defines standard macros, data types, and subroutines.

Related Information

The [pm_init Subroutine] on page 1069, [pm_error Subroutine] on page 1024, [pm_set_program_mythread Subroutine] on page 1089, [pm_delete_program_mythread Subroutine] on page 1020, [pm_get_data_mythread, pm_get_tdata_mythread or pm_get_tdata_mythread Subroutine] on page 1036, [pm_start_mythread and pm_tstart_mythread Subroutine] on page 1105, [pm_stop_mythread and pm_tstop_mythread Subroutine] on page 1113, [pm_reset_data_mythread Subroutine] on page 1076.


pm_get_program_mythread_mx Subroutine

**Purpose**

Retrieves the Performance Monitor settings in counter multiplexing mode for the calling thread.
Library
Performance Monitor APIs Library (libpmapi.a)

Syntax
#include <pmapi.h>

int pm_get_program_mythread_mx ( pm_prog_mx_t *prog )

Description
The pm_get_program_mythread_mx subroutine retrieves the Performance Monitor settings for the
calling kernel thread. This includes mode information and the events being counted, which are in an array
of list of event identifiers. The identifiers come from the lists returned by the pm_initialize subroutine.

The counting mode includes user mode and kernel mode, and the current counting state.

If the list includes an event which can be used with a threshold (as indicated by the pm_init subroutine), a
threshold value is also returned.

The user application must free the array allocated to store the event lists ( the events_set field in the prog
parameter).

Parameters
*prog

Returns which Performance Monitor events and modes
are set. Supported modes are:

PM_USER
Counting processes running in user mode

PM_KERNEL
Counting processes running in kernel mode

PM_COUNT
Counting is on

Return Values
0
No errors occurred.
Positive error code
Refer to the "pm_error Subroutine" on page 1024 to decode the error code.

Error Codes
Refer to the "pm_error Subroutine" on page 1024.

Files
/usr/include/pmapi.h
Defines standard macros, data types, and subroutines.

Related Information
The "pm_init Subroutine" on page 1069, "pm_error Subroutine" on page 1024,
"pm_set_program_mythread_mx Subroutine" on page 1090, "pm_delete_program_mythread Subroutine"
on page 1020, "pm_get_data_mythread_mx or pm_get_tdata_mythread_mx Subroutine" on page 1038.
pm_get_program_pgroup Subroutine

Purpose
Retrieves Performance Monitor settings for the counting group to which a target pthread belongs.

Library
Performance Monitor APIs Library (libpmapi.a)

Syntax
#include <pmapi.h>

int pm_get_program_pgroup ( pid, tid, ptid, *prog)

pid_t pid;
tid_t tid;
ptid_t ptid;
pm_prog_t *prog;

Description
The pm_get_program_pgroup subroutine retrieves the Performance Monitor settings for the counting group to which a target pthread belongs. The pthread must be stopped and must be part of a debuggee process, under the control of the calling process. This includes mode information and the events being counted, which are in a list of event identifiers. The identifiers come from the lists returned by the pm_initialize subroutine.

If the pthread is running in 1:1 mode, only the tid parameter must be specified. If the pthread is running in m:n mode, only the ptid parameter must be specified. If both the ptid and tid parameters are specified, they must be referring to a single pthread with the ptid parameter specified and currently running on a kernel thread with specified tid parameter.

The counting mode includes the user mode and kernel mode, and the current counting state.

If the list includes an event that can be used with a threshold (as indicated by the pm_initialize subroutine), a threshold value is also returned.

Parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>pid</td>
<td>Process ID of target pthread. The target process must be an argument of a debug process.</td>
</tr>
<tr>
<td>tid</td>
<td>Thread ID of target pthread. To ignore this parameter, set it to 0.</td>
</tr>
<tr>
<td>ptid</td>
<td>Pthread ID of the target pthread. To ignore this parameter, set it to 0.</td>
</tr>
</tbody>
</table>
*prog* Returns which Performance Monitor events and modes are set. The following modes are supported:

**PM_USER**
Counts process running in user mode

**PM_KERNEL**
Counts process running kernel mode

**PM_COUNT**
Counting is on

**PM_PROCESS**
Process-level counting group

Return Values

<table>
<thead>
<tr>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>No errors occurred.</td>
</tr>
</tbody>
</table>

Positive error code Refer to the ["pm_error Subroutine" on page 1024](#) to decode the error code.

Error Codes
Refer to the ["pm_error Subroutine" on page 1024](#).

Files

`/usr/include/pmapi.h`
Defines standard macros, data types, and subroutines.

Related Information

The ["pm_delete_program_pgroup Subroutine" on page 1021](#), ["pm_error Subroutine" on page 1024](#), ["pm_get_data_pgroup, pm_get_tdata_pgroup and pm_get_Tdata_pgroup Subroutine" on page 1039](#), ["pm_set_program_pgroup Subroutine" on page 1092](#), ["pm_initialize Subroutine" on page 1071](#), ["pm_reset_data_pgroup Subroutine" on page 1076](#), ["pm_start_pgroup and pm_tstart_pgroup Subroutine" on page 1106](#), ["pm_stop_pgroup and pm_tstop_pgroup Subroutine" on page 1114](#).


**pm_get_program_pgroup_mx Subroutine**

**Purpose**
Retrieves Performance Monitor settings in counter multiplexing mode for the counting group to which a target pthread belongs.

**Library**
Performance Monitor APIs Library (`libpmapi.a`)

**Syntax**

```c
#include <pmapi.h>

int pm_get_program_pgroup_mx (pid_t pid, tid_t tid, ptid_t ptid, *prog)
```

*prog*
Description
The `pm_get_program_pgroup_mx` subroutine retrieves the Performance Monitor settings for the counting group to which a target pthread belongs. The pthread must be stopped and must be part of a debuggee process, under the control of the calling process. This includes mode information and the events being counted, which are in an array of list of event identifiers. The identifiers come from the lists returned by the `pm_initialize` subroutine.

If the pthread is running in 1:1 mode, only the `tid` parameter must be specified. If the pthread is running in m:n mode, only the `ptid` parameter must be specified. If both the `ptid` and `tid` parameters are specified, they must be referring to a single pthread with the `ptid` parameter specified and currently running on a kernel thread with specified `tid` parameter.

The counting mode includes the user mode and kernel mode, and the current counting state.

If the list includes an event that can be used with a threshold (as indicated by the `pm_initialize` subroutine), a threshold value is also returned.

The user application must free the array allocated to store the event lists (the events_set field in the `prog` parameter).

Parameters

- `pid`  Process ID of target pthread. The target process must be an argument of a debug process.
- `tid`  Thread ID of target pthread. To ignore this parameter, set it to 0.
- `ptid`  Pthread ID of the target pthread. To ignore this parameter, set it to 0.
- `*prog`  Returns which Performance Monitor events and modes are set. The following modes are supported:

  - `PM_USER`  Counts process running in user mode
  - `PM_KERNEL`  Counts process running kernel mode
  - `PM_COUNT`  Counting is on
  - `PM_PROCESS`  Process-level counting group

Return Values

- 0  No errors occurred.
- Positive error code  Refer to the "pm_error Subroutine" on page 1024 to decode the error code.

Error Codes

Refer to the "pm_error Subroutine" on page 1024.
Related Information
The "pm_delete_program_pgroup Subroutine" on page 1021, "pm_error Subroutine" on page 1024, "pm_get_data_pgroup_mx and pm_get_tdata_pgroup_mx Subroutine" on page 1041, "pm_set_program_pgroup_mx Subroutine" on page 1093, "pm_initialize Subroutine" on page 1071, "pm_reset_data_pgroup Subroutine" on page 1076, "pm_start_pgroup and pm_tstart_pgroup Subroutine" on page 1106, "pm_stop_pgroup and pm_tstop_pgroup Subroutine" on page 1114.


pm_get_program_pthread Subroutine

Purpose
Retrieves the Performance Monitor settings for a target pthread.

Library
Performance Monitor APIs Library (libpmpi.a)

Syntax
#include <pmapi.h>

int pm_set_program_pthread (pid, tid, ptid, *prog)
pid_t pid;
tid_t tid;
ptid_t ptid;
prog_t *prog;

Description
The pm_get_program_pthread subroutine retrieves the Performance Monitor settings for a target pthread. The pthread must be stopped and must be part of a debuggee process, under the control of the calling process. This includes mode information and the events being counted, which are in a list of event identifiers. The identifiers must be selected from the lists returned by the pm_initialize subroutine.

If the pthread is running in 1:1 mode, only the tid parameter must be specified. If the pthread is running in m:n mode, only the ptid parameter must be specified. If both the ptid and tid parameters are specified, they must be referring to a single pthread with the ptid parameter specified and currently running on a kernel thread with specified tid parameter.

The counting mode includes user mode and kernel mode, and the current counting state.

If the list includes an event that can be used with a threshold (as indicated by the pm_initialize subroutine), a threshold value is also returned.

Parameters

pid Process ID of target pthread. Target process must be an argument of a debug process.
tid

Thread ID of target pthread. To ignore this parameter, set it to 0.

ptid

Pthread ID of the target pthread. To ignore this parameter, set it to 0.

*prog

Returns which Performance Monitor events and modes are set. The following modes are supported:

PM_USER

Counts processes running in User Mode

PM_KERNEL

Counts processes running in Kernel Mode

PM_COUNT

Counting is On

Return Values

0

No errors occurred.

Positive error code

Refer to the "pm_error Subroutine" on page 1024 to decode the error code.

Error Codes

Refer to the "pm_error Subroutine" on page 1024.

Files

/usr/include/pmapi.h

Defines standard macros, data types, and subroutines.

Related Information

The "pm_delete_program_pthread Subroutine" on page 1022, "pm_error Subroutine" on page 1024, "pm_get_data_pthread, pm_get_tdata_pthread or pm_get_Tdata_pthread Subroutine" on page 1043, "pm_set_program_pthread Subroutine" on page 1095, "pm_initialize Subroutine" on page 1071, "pm_reset_data_pthread Subroutine" on page 1078, "pm_start_pthread and pm_tstart_pthread Subroutine" on page 1107, "pm_stop_pthread and pm_tstop_pthread Subroutine" on page 1116.


pm_get_program_pthread_mx Subroutine

Purpose

Retrieves the Performance Monitor settings in counter multiplexing mode for a target pthread.

Library

Performance Monitor APIs Library (libpmapi.a)

Syntax

#include <pmapi.h>

int pm_set_program_pthread_mx ( pid, tid, ptid, *prog )

pid_t pid;
Description
The `pm_get_program_pthread_mx` subroutine retrieves the Performance Monitor settings for a target pthread. The pthread must be stopped and must be part of a debuggee process, under the control of the calling process. This includes mode information and the events being counted, which are in an array of list of event identifiers. The identifiers must be selected from the lists returned by the `pm_initialze` subroutine.

If the pthread is running in 1:1 mode, only the `tid` parameter must be specified. If the pthread is running in m:n mode, only the `ptid` parameter must be specified. If both the `ptid` and `tid` parameters are specified, they must be referring to a single pthread with the `ptid` parameter specified and currently running on a kernel thread with specified `tid` parameter.

The counting mode includes user mode and kernel mode, and the current counting state.

If the list includes an event that can be used with a threshold (as indicated by the `pm_initialize` subroutine), a threshold value is also returned.

The user application must free the array allocated to store the event lists (the `events_set` field in the `prog` parameter).

Parameters

- `pid`: Process ID of target pthread. Target process must be an argument of a debug process.
- `tid`: Thread ID of target pthread. To ignore this parameter, set it to 0.
- `ptid`: Pthread ID of the target pthread. To ignore this parameter, set it to 0.
- `*prog`: Returns which Performance Monitor events and modes are set. The following modes are supported:
  - `PM_USER`: Counts processes running in User Mode
  - `PM_KERNEL`: Counts processes running in Kernel Mode
  - `PM_COUNT`: Counting is On

Return Values

- `0`: No errors occurred.
- Positive error code: Refer to the [pm_error Subroutine](#) on page 1024 to decode the error code.

Error Codes

Refer to the `pm_error` [pm_error Subroutine](#) subroutine.

Files

- `/usr/include/pmapi.h`: Defines standard macros, data types, and subroutines.
Related Information

The "pm_delete_program_pthread Subroutine" on page 1022, "pm_error Subroutine" on page 1024,
"pm_get_data_pthread_mx or pm_get_tdata_pthread_mx Subroutine" on page 1044,
"pm_set_program_pthread_mx Subroutine" on page 1097,"pm_initialize Subroutine" on page 1071,
"pm_reset_data_pthread Subroutine" on page 1078, "pm_start_pthread and pm_tstart_pthread Subroutine" on page 1107, "pm_stop_pthread and pm_tstop_pthread Subroutine " on page 1116.


pm_get_program_thread Subroutine

Purpose
Retrieves the Performance Monitor settings for a target thread.

Library
Performance Monitor APIs Library (libpmapi.a)

Syntax
#include <pmapi.h>

int pm_get_program_thread ( pid, tid, *prog)

pid_t pid;

 tid_t tid;

 pm_prog_t *prog;

Description
This subroutine supports only the 1:1 threading model. It has been superseded by the
pm_get_program_pthread subroutine, which supports both the 1:1 and the M:N threading models. A call
to this subroutine is equivalent to a call to the pm_get_program_pthread subroutine with a ptid parameter
equal to 0.

The pm_get_program_thread subroutine retrieves the Performance Monitor settings for a target kernel
thread. The thread must be stopped and must be part of a debuggee process under the control of the
calling process. This includes mode information and the events being counted, which are in a list of event
identifiers. The identifiers come from the lists returned by the pm_init subroutine.

The counting mode includes user mode and kernel mode, and the current counting state.

If the list includes an event which can be used with a threshold (as indicated by the pm_init subroutine), a
threshold value is also returned.

Parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>pid</td>
<td>Process identifier of the target thread. The target process must be an argument of a debug process.</td>
</tr>
<tr>
<td>tid</td>
<td>Thread identifier of the target thread.</td>
</tr>
</tbody>
</table>
Returns which Performance Monitor events and modes are set. Supported modes are:

- **PM_USER**
  - Counting processes running in **User** mode
- **PM_KERNEL**
  - Counting processes running in **Kernel** mode
- **PM_COUNT**
  - Counting is On

### Return Values

<table>
<thead>
<tr>
<th>Error Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>No errors occurred.</td>
</tr>
<tr>
<td>Positive error code</td>
<td>Refer to the &quot;pm_error Subroutine&quot; on page 1024 to decode the error code.</td>
</tr>
</tbody>
</table>

### Error Codes

Refer to the "pm_error Subroutine" on page 1024.

### Files

/usr/include/pmapi.h

Defines standard macros, data types, and subroutines.

### Related Information

The "pm_init Subroutine" on page 1069, "pm_error Subroutine" on page 1024, "pm_set_program_thread Subroutine" on page 1098, "pm_delete_program_thread Subroutine" on page 1023, "pm_get_data_thread, pm_get_tdata_thread or pm_get_Tdata_thread Subroutine" on page 1046, "pm_start_thread and pm_tstart_thread Subroutine" on page 1109, "pm_stop_thread and pm_tstop_thread Subroutine" on page 1117, "pm_reset_data_thread Subroutine" on page 1079.

The "pm_get_program_thread_mx Subroutine" on page 1069 manages the Performance Monitor settings for a target thread.

### Syntax

```c
#include <pmapi.h>

int pm_get_program_thread_mx ( pid_t pid, tid_t tid, pm_prog_mx_t *prog )
```

### Library

Performance Monitor APIs Library (libpmapi.a)
Description
This subroutine supports only the 1:1 threading model. It has been superseded by the
`pm_get_program_pthread_mx` subroutine, which supports both the 1:1 and the M:N threading models. A
call to this subroutine is equivalent to a call to the `pm_get_program_pthread_mx` subroutine with a `ptid`
parameter equal to 0.

The `pm_get_program_thread_mx` subroutine retrieves the Performance Monitor settings for a target
kernel thread. The thread must be stopped and must be part of a debuggee process under the control of
the calling process. This includes mode information and the events being counted, which are in an array of
list of event identifiers. The identifiers come from the lists returned by the `pm_initialize` subroutine.

The counting mode includes user mode and kernel mode, and the current counting state.

If the list includes an event which can be used with a threshold (as indicated by the `pm_init` subroutine), a
threshold value is also returned.

The user application must free the array allocated to store the event lists (the `events_set` field in the `prog`
parameter).

Parameters

`pid`  Process identifier of the target thread. The target process
    must be an argument of a debug process.

`tid`  Thread identifier of the target thread.

`*prog`  Returns which Performance Monitor events and modes
    are set. Supported modes are:

    `PM_USER`  Counting processes running in **User** mode

    `PM_KERNEL`  Counting processes running in **Kernel** mode

    `PM_COUNT`  Counting is On

Return Values

0  No errors occurred.

Positive error code  Refer to the "pm_error Subroutine" on page 1024 to decode the error code.

Error Codes

Refer to the "pm_error Subroutine" on page 1024.

Files

`/usr/include/pmapi.h`  Defines standard macros, data types, and subroutines.

Related Information

The "pm_init Subroutine" on page 1069, "pm_error Subroutine" on page 1024,
"pm_set_program_thread_mx Subroutine" on page 1100, "pm_delete_program_thread Subroutine" on page
1023, "pm_get_data_thread_mx or pm_get_tdata_thread_mx Subroutine" on page 1048, "pm_start_thread
and pm_tstart_thread Subroutine" on page 1109, "pm_stop_thread and pm_tstop_thread Subroutine" on
page 1117, "pm_reset_data_thread Subroutine" on page 1079.
pm_init Subroutine

Purpose
Initializes the Performance Monitor APIs.

Library
Performance Monitor APIs Library (libpmapi.a)

Syntax
#include <pmapi.h>

int pm_init (int filter, pm_info_t *pminfo, pm_groups_info_t *pm_groups_info);

int filter;
pm_info_t *pminfo;
pm_groups_info_t *pm_groups_info;

Description

Note: The pm_init subroutine cannot be used on processors newer than POWER4™. With such processors, the pm_initialize subroutine must be used.

The pm_init subroutine initializes the Performance Monitor API library. It returns, after taking into account a filter on the status of the events, the number of counters available on this processor, and one table per counter with the list of events available. For each event, an event identifier, a status, a flag indicating if the event can be used with a threshold, two names, and a description are provided.

The event identifier is used with all the pm_set_program interfaces and is also returned by all of the pm_get_program interfaces. Only event identifiers present in the table returned can be used. In other words, the filter is effective for all API calls.

The status describes whether the event has been verified, is still unverified, or works with some caveat, as explained in the description. This field is necessary because the filter can be any combination of the three available status bits. The flag points to events that can be used with a threshold.

Only events categorized as verified have gone through full verification. Events categorized as caveat have been verified only within the limitations documented in the event description. Events categorized as unverified have undefined accuracy. Use caution with unverified events; the Performance Monitor software is essentially providing a service to read hardware registers which may or may not have any meaningful content. Users may experiment with unverified event counters and determine for themselves what, if any, use they may have for specific tuning situations.

The short mnemonic name is provided for easy keyword-based search in the event table (see the sample program /usr/samples/pmapi/sysapit2.c for code using mnemonic names). The complete name of the event is also available and a full description for each event is returned.

The structure returned also has the threshold multiplier for this processor and the processor name.

On some platforms, it is possible to specify event groups instead of individual events. Event groups are predefined sets of events. Rather than specify each event individually, a single group ID is specified. On some platforms, such as POWER4, use of the event groups is required, and attempts to specify individual events return an error.
The interface to `pm_init` has been enhanced to return the list of supported event groups in an optional third parameter. For binary compatibility, the third parameter must be explicitly requested by OR-ing the bitflag, PM_GET_GROUPS, into the `filter` parameter.

If the `pm_groups_info` parameter returned by `pm_init` is NULL, there are no supported event groups for the platform. Otherwise an array of `pm_groups_t` structures are returned in the `event_groups` field. The length of the array is given by the `max_groups` field.

The `pm_groups_t` structure contains a group identifier, two names and a description that are similar to those of the individual events. In addition, there is an array of integers that specify the events contained in the group.

**Parameters**

- **filter**
  Specifies which event types to return.
  - `PM_VERIFIED` Events which have been verified
  - `PM_UNVERIFIED` Events which have not been verified
  - `PM_CAVEAT` Events which are usable but with caveats as described in the long description

- **pminfo**
  Returned structure with processor name, threshold multiplier, and a filtered list of events with their current status.

- **pm_groups_info**
  Returned structure with list of supported groups. This parameter is only meaningful if PM_GET_GROUPS is OR-ed into the `filter` parameter.

**Return Values**

- **0** No errors occurred.
- **Positive error code** Refer to the `pm_error` subroutine to decode the error code.

**Error Codes**

Refer to the `pm_error` subroutine.

**Files**

- `/usr/include/pmaapi.h` Defines standard macros, data types, and subroutines.

**Related Information**

- "pm_initialize Subroutine" on page 1071.
pm_initialize Subroutine

Purpose
Initializes the Performance Monitor APIs and returns information about a processor.

Library
Performance Monitor APIs Library (libpmapi.a)

Syntax
#include <pmapi.h>

int pm_initialize (filter, pminfo, pmgroups, proctype)

int filter;
pm_info2_t *pminfo;
pm_groups_info_t *pmgroups;
int proctype;

Description
The pm_initialize subroutine initializes the Performance Monitor API library and retrieves information about a type of processor (if the specified proctype is not PM_CURRENT). It takes into account a filter on the events status, then it returns the number of counters available on this processor and one table per counter containing the list of available events. For each event, it provides an event identifier, a status, two names, and a description. The status contains a set of flags indicating: the event status, if the event can be used with a threshold, if the event is a shared event, and if the event is a grouped-only event.

The event identifier is used with all pm_set_program interfaces and is also returned by all of the pm_get_program interfaces. Only event identifiers present in the returned table can be used. In other words, the filter is effective for all API calls.

The status describes whether the event has been verified, is still unverified, or works with some caveat, as explained in the description. This field is necessary because the filter can be any combination of the three available status bits. The flag points to events that can be used with a threshold.

Only events categorized as verified have been fully verified. Events categorized as caveat have been verified only with the limitations documented in the event description. Events categorized as unverified have an undefined accuracy. Use unverified events cautiously; the Performance Monitor software provides essentially a service to read hardware registers, which might or might not have meaningful content. Users might experiment for themselves with unverified event counters to determine if they can be used for specific tuning situations.

The short mnemonic name is provided for an easy keyword-based search in the event table (see the sample program /usr/samples/pmapi/cpi.c for code using mnemonic names). The complete name of the event is also available, and a full description for each event is returned.

The returned structure also contains the threshold multipliers for this processor, the processor name, and its characteristics. On some platforms, up to three threshold multipliers are available.

On some platforms, it is possible to specify event groups instead of individual events. Event groups are predefined sets of events. Rather than specify each event individually, a single group ID is specified. On some platforms, such as POWER4, using event groups is mandatory, and specifying individual events returns an error.
The interface to \texttt{pm\_initialize} returns the list of supported event groups in its third parameter. If the \texttt{pmgroups} parameter returned by \texttt{pm\_initialize} is NULL, there are no supported event groups for the platform. Otherwise an array of \texttt{pm\_groups\_t} structures is returned in the \texttt{event\_groups} field. The length of the array is given by the \texttt{max\_groups} field.

The \texttt{pm\_groups\_t} structure contains a group identifier, two names, and a description that are all similar to those of the individual events. In addition, an array of integers specifies the events contained in the group.

If the \texttt{proctype} parameter is not set to \texttt{PM\_CURRENT}, the Performance Monitor APIs library is not initialized, and the subroutine only returns information about the specified processor and those events and groups available in its parameters (\texttt{pminfo} and \texttt{pmgroups}) taking into account the filter. If the \texttt{proctype} parameter is set to \texttt{PM\_CURRENT}, in addition to returning the information described, the Performance Monitor APIs library is initialized and ready to accept other calls.

### Parameters

- **filter**
  - Specifies which event types to return.
  - \texttt{PM\_VERIFIED}
    - Events that have been verified.
  - \texttt{PM\_UNVERIFIED}
    - Events that have not been verified.
  - \texttt{PM\_CAVEAT}
    - Events that are usable but with caveats, as explained in the long description.

- **pmgroups**
  - Returned structure containing the list of supported groups.

- **pminfo**
  - Returned structure containing the processor name, the threshold multiplier and a filtered list of events with their current status.

- **proctype**
  - Initializes the Performance Monitor API and retrieves information about a specific processor type:
  - \texttt{PM\_CURRENT}
    - Retrieves information about the current processor and initializes the Performance Monitor API library.
  - \texttt{other}
    - Retrieves information about a specific processor.

### Return Values

- 0
  - No errors occurred.

- Positive error code
  - Refer to the \texttt{pm\_error Subroutine} on page 1024 to decode the error code.

### Error Codes

Refer to the \texttt{pm\_error Subroutine} on page 1024.

### Files

\texttt{/usr/include/pmapi.h}
- Defines standard macros, data types, and subroutines.

## Related Information

The \texttt{pm\_initialize} subroutine replaces \texttt{pm\_init} subroutine. It is mandatory to initialize the Performance Monitor API library for processors newer than Power4.

\texttt{pm\_error Subroutine} on page 1024.
pm_reset_data Subroutine

Purpose
Resets system wide Performance Monitor data.

Library
Performance Monitor APIs Library (libpmapi.a)

Syntax
#include <pmapi.h>
int pm_reset_data ()

Description
The pm_reset_data subroutine resets the current system wide Performance Monitor data. The data is a set (one per hardware counter on the machine used) of 64-bit values. All values are reset to 0.

Return Values
0 Operation completed successfully.
Positive Error Code Refer to the pm_error subroutine to decode the error code.

Error Codes
See the pm_error subroutine.

Files
/usr/include/pmapi.h Defines standard macros, data types, and subroutines.

Related Information
The pm_init subroutine, pm_error subroutine, pm_set_program subroutine, pm_get_program subroutine, pm_delete_program subroutine, pm_get_data subroutine, pm_get_Tdata subroutine, pm_get_data_cpu subroutine, pm_get_Tdata_cpu subroutine, pm_get_data_lcpu subroutine, pm_get_Tdata_lcpu subroutine, pm_start subroutine, pm_stop subroutine.

pm_reset_data_group Subroutine

Purpose
Resets Performance Monitor data for a target thread and the counting group to which it belongs.
Library
Performance Monitor APIs Library (libpmapi.a)

Syntax
#include <pmapi.h>

int pm_reset_data_group ( pid_t pid; tid_t tid;

Description
This subroutine supports only the 1:1 threading model. It has been superseded by the
pm_reset_data_pgroup subroutine, which supports both the 1:1 and the M:N threading models. A call to
this subroutine is equivalent to a call to the pm_reset_data_pgroup subroutine with a ptid parameter
equal to 0.

The pm_reset_data_group subroutine resets the current Performance Monitor data for a target kernel
thread and the counting group to which it belongs. The thread must be stopped and must be part of a
debugee process, under control of the calling process. The data is a set (one per hardware counter on the
machine used) of 64-bit values. All values are reset to 0. Because the data for all the other threads in the
group is not affected, the group is marked as inconsistent unless it has only one member.

Parameters

pid Process ID of target thread. Target process must be a
debugee of the caller process.

tid Thread ID of target thread.

Return Values

0 Operation completed successfully.

Positive Error Code Refer to the pm_error subroutine to decode the
error code.

Error Codes
Refer to the pm_error subroutine.

Files
/usr/include/pmapi.h Defines standard macros, data types, and subroutines.

Related Information
The pm_init subroutine, pm_error subroutine, pm_set_program_group subroutine, pm_get_program_group subroutine, pm_delete_program_group subroutine, pm_start_group subroutine, pm_stop_group subroutine, pm_get_data_group subroutine, pm_get_tdata_group subroutine, pm_get_Tdata_group subroutine.
pm_reset_data_mygroup Subroutine

Purpose
Resets Performance Monitor data for the calling thread and the counting group to which it belongs.

Library
Performance Monitor APIs Library (libpmapi.a)

Syntax
#include <pmapi.h>

int pm_reset_data_mygroup()

Description
The pm_reset_data_mygroup subroutine resets the current Performance Monitor data for the calling kernel thread and the counting group to which it belongs. The data is a set (one per hardware counter on the machine used) of 64-bit values. All values are reset to 0. Because the data for all the other threads in the group is not affected, the group is marked as inconsistent unless it has only one member.

Return Values
0 Operation completed successfully.
Positive Error Code Refer to the pm_error subroutine to decode the error code.

Error Codes
Refer to the pm_error subroutine.

Files
/usr/include/pmapi.h Defines standard macros, data types, and subroutines.

Related Information
The pm_init subroutine, pm_error subroutine, pm_set_program_mygroup subroutine, pm_get_program_mygroup subroutine, pm_delete_program_mygroup subroutine, pm_start_mygroup subroutine, pm_stop_mygroup subroutine, pm_get_data_mygroup subroutine.
pm_reset_data_mythread Subroutine

Purpose
Resets Performance Monitor data for the calling thread.

Library
Performance Monitor APIs Library (libpmapi.a)

Syntax
#include <pmapi.h>

int pm_reset_data_mythread()

Description
The pm_reset_data_mythread subroutine resets the current Performance Monitor data for the calling kernel thread. The data is a set (one per hardware counter on the machine) of 64-bit values. All values are reset to 0.

Return Values
0 Operation completed successfully.
Positive Error Code Refer to the pm_error subroutine to decode the error code.

Error Codes
Refer to the pm_error subroutine.

Files
/usr/include/pmapi.h Defines standard macros, data types, and subroutines.

Related Information
The pm_init subroutine, pm_error subroutine, pm_set_program_mythread subroutine, pm_get_program_mythread subroutine, pm_delete_program_mythread subroutine, pm_start_mythread subroutine, pm_stop_mythread subroutine, pm_get_data_mythread subroutine.


pm_reset_data_pgroup Subroutine

Purpose
Resets Performance Monitor data for a target pthread and the counting group to which it belongs.
Library
Performance Monitor APIs Library (libpmaopi.a)

Syntax
#include <pmapi.h>

int pm_reset_data_pgroup (pid, tid, ptid)

pid_t pid;
tid_t tid;
ptid_t ptid;

Description
The pm_reset_data_pgroup subroutine resets the current Performance Monitor data for a target pthread and the counting group to which it belongs. The pthread must be stopped and must be part of a deuggee process, under control of the calling process. The data is a set (one per hardware counter on the machine used) of 64-bit values. All values are reset to 0. Because the data for all the other pthreads in the group is not affected, the group is marked as inconsistent unless it has only one member.

If the pthread is running in 1:1 mode, only the tid parameter must be specified. If the pthread is running in m:n mode, only the ptid parameter must be specified. If both the ptid and tid parameters are specified, they must be referring to a single pthread with the ptid parameter specified and currently running on a

kernel thread with specified tid parameter.

Parameters
 pid Process ID of target pthread. Target process must be a debuggee of the caller process.
 tid Thread ID of target pthread. To ignore this parameter, set it to 0.
 ptid Pthread ID of the target pthread. To ignore this parameter, set it to 0.

Return Values
0 Operation completed successfully.
Positive error code Refer to the pm_error Subroutine on page 1024 to decode the error code.

Error Codes
Refer to the pm_error Subroutine on page 1024.

Files
/usr/include/pmaopi.h Defines standard macros, data types, and subroutines.

Related Information
The pm_delete_program_pgroup Subroutine on page 1021, pm_error Subroutine on page 1024,
 pm_get_data_pgroup, pm_get_tdata_pgroup and pm_get_Tdata_pgroup Subroutine on page 1039,
 pm_get_program_pgroup Subroutine on page 1060, pm_initialize Subroutine on page 1071,
 pm_reset_data_pgroup Subroutine on page 1076, pm_set_program_pgroup Subroutine on page 1092,
 pm_start_pgroup and pm_tstart_pgroup Subroutine on page 1106, pm_stop_pgroup and
 pm_tstop_pgroup Subroutine on page 1114.
pm_reset_data_pthread Subroutine

Purpose
Resets Performance Monitor data for a target pthread.

Library
Performance Monitor APIs Library (libpmapi.a)

Syntax
#include <pmapi.h>

int pm_reset_data_pthread ( pid, tid, ptid);

pid_t pid;
tid_t tid;
ptid_t ptid;

Description
The pm_reset_data_pthread subroutine resets the current Performance Monitor data for a target pthread. The pthread must be stopped and must be part of a debuggee process. The data is a set (one per hardware counter on the machine used) of 64-bit values. All values are reset to 0.

If the pthread is running in 1:1 mode, only the tid parameter must be specified. If the pthread is running in m:n mode, only the ptid parameter must be specified. If both the ptid and tid parameters are specified, they must be referring to a single pthread with the ptid parameter specified and currently running on a kernel thread with specified tid parameter.

Parameters

pid
Process ID of target pthread. Target process must be a debuggee of the caller process.

tid
Thread ID of target pthread. To ignore this parameter, set it to 0.

ptid
Pthread ID of the target pthread. To ignore this parameter, set it to 0.

Return Values

0 Operation completed successfully.

Positive error code Refer to the "pm_error Subroutine" on page 1024 to decode the error code.

Error Codes
Refer to the "pm_error Subroutine" on page 1024.

Files
/usr/include/pmapi.h Defines standard macros, data types, and subroutines.
pm_reset_data_thread Subroutine

Purpose
Resets Performance Monitor data for a target thread.

Library
Performance Monitor APIs Library (libpmapi.a)

Syntax
#include <pmapi.h>

int pm_reset_data_thread (pid_t pid, tid_t tid);

Description
This subroutine supports only the 1:1 threading model. It has been superseded by the
pm_reset_data_pthread subroutine, which supports both the 1:1 and the M:N threading models. A call to
this subroutine is equivalent to a call to the pm_reset_data_pthread subroutine with a ptid parameter
equal to 0.

The pm_reset_data_thread subroutine resets the current Performance Monitor data for a target kernel
thread. The thread must be stopped and must be part of a debuggee process. The data is a set (one per
hardware counter on the machine used) of 64-bit values. All values are reset to 0.

Parameters

pid Process id of target thread. Target process must be a
debugee of the caller process.

tid Thread id of target thread.

Return Values

0 Operation completed successfully.

Positive Error Code Refer to the pm_error subroutine to decode the
error code.

Error Codes
Refer to the pm_error subroutine.
Related Information

The pm_init subroutine, pm_error subroutine, pm_set_program_thread subroutine, pm_get_program_thread subroutine, pm_delete_program_thread subroutine, pm_start_thread subroutine, pm_stop_thread subroutine, pm_get_data_thread subroutine.


pm_set_program Subroutine

Purpose
Sets system wide Performance Monitor programmation.

Library
Performance Monitor APIs Library (libpmapi.a)

Syntax

    #include <pmai.h>

    int pm_set_program ( *prog
    pm_prog_t *prog;

Description

The pm_set_program subroutine sets system wide Performance Monitor programmation. The setting includes the events to be counted, and a mode in which to count. The events to count are in a list of event identifiers. The identifiers must be selected from the lists returned by the pm_init subroutine.

The counting mode includes User Mode and/or Kernel Mode, the Initial Counting State, and the Process Tree Mode. The Process Tree Mode sets counting to On only for the calling process and its descendants. The defaults are set to Off for User Mode and Kernel Mode. The initial default state is set to delay counting until the pm_start subroutine is called, and to count the activity of all the processes running in the system.

If the list includes an event which can be used with a threshold (as indicated by the pm_init subroutine), a threshold value can also be specified.

On some platforms, event groups can be specified instead of individual events. This is done by setting the bitfield is_group in the mode, and placing the group ID into the first element of the events array. (The group ID was obtained by pm_init).
Parameters

`*prog` Specifies the events and modes to use in Performance Monitor setup. The following modes are supported:

- **PM_USER**
  Counts processes running in User Mode (default is set to Off)

- **PM_KERNEL**
  Counts processes running in Kernel Mode (default is set to Off)

- **PM_COUNT**
  Starts counting immediately (default is set to Not to Start Counting)

- **PM_PROCTREE**
  Sets counting to On only for the calling process and its descendants (default is set to Off)

Return Values

<table>
<thead>
<tr>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Operation completed successfully.</td>
</tr>
<tr>
<td>Positive error code</td>
<td>Refer to the <a href="#"><code>pm_error Subroutine</code> on page 1024</a> to decode the error code.</td>
</tr>
</tbody>
</table>

Error Codes

Refer to the [`pm_error Subroutine` on page 1024](#).

Files

`/usr/include/pmapi.h` Defines standard macros, data types, and subroutines.

Related Information

The `pm_init Subroutine` on page 1069, `pm_error Subroutine` on page 1024, `pm_get_program Subroutine` on page 1049, `pm_delete_program Subroutine` on page 1017, `pm_get_data, pm_get_tdata, pm_get_Tdata, pm_get_data_cpu, pm_get_tdata_cpu, pm_get_Tdata_cpu, pm_get_data_lcpu, pm_get_tdata_lcpu and pm_get_Tdata_lcpu Subroutine` on page 1025, `pm_start and pm_tstart Subroutine` on page 1101, `pm_stop and pm_tstop Subroutine` on page 1110, `pm_reset_data Subroutine` on page 1073.

Performance Monitor API Programming Concepts in [AIX 5L Version 5.3 Performance Tools Guide and Reference](#)

`pm_set_program_group Subroutine`

**Purpose**
Sets Performance Monitor programmation for a target thread and creates a counting group.

**Library**
Performance Monitor APIs Library (`libpmapi.a`)
Syntax

```c
#include <pmapi.h>

int pm_set_program_group (pid, tid, *prog);
```

### Description

This subroutine supports only the 1:1 threading model. It has been superseded by the `pm_set_program_pgroup` subroutine, which supports both the 1:1 and the M:N threading models. A call to this subroutine is equivalent to a call to the `pm_set_program_pgroup` subroutine with a `ptid` parameter equal to 0.

The `pm_set_program_group` subroutine sets the Performance Monitor programmation for a target kernel thread. The thread must be stopped and must be part of a debuggee process, under the control of the calling process. The setting includes the events to be counted and a mode in which to count. The events to count are in a list of event identifiers. The identifiers must be selected from the lists returned by the `pm_init` subroutine.

This call also creates a counting group, which includes the target thread and any thread which it, or any of its descendants, will create in the future. Optionally, the group can be defined as also containing all the existing and future threads belonging to the target process.

The counting mode includes User Mode and/or Kernel Mode, and the Initial Counting State. The defaults are set to Off for User Mode and Kernel Mode, and the initial default state is set to delay counting until the `pm_start_group` subroutine is called.

If the list includes an event which can be used with a threshold (as indicated by the `pm_init` subroutine), a threshold value can also be specified.

### Parameters

- **`pid`**
  - Process ID of target thread. Target process must be a debuggee of a calling process.
  - Thread ID of target thread.

- **`tid`**
  - Thread ID of target thread.

- **`*prog`**
  - Counts processes running in User Mode (default is set to Off)
  - Counts processes running in Kernel Mode (default is set to Off)
  - Starts counting immediately (default is set to Not to Start Counting)
  - Creates a process-level counting group

### Return Values

- **0**  
  - Operation completed successfully.

- **Positive error code**  
  - Refer to the "pm_error Subroutine" on page 1024 to decode the error code.
Error Codes
Refer to the "pm_error Subroutine" on page 1024.

Files
/usr/include/pmapi.h Defines standard macros, data types, and subroutines.

Related Information
The "pm_init Subroutine" on page 1069, "pm_error Subroutine" on page 1024, "pm_get_program_group Subroutine" on page 1051, "pm_delete_program_group Subroutine" on page 1018, "pm_get_data_group, pm_get_tdata_group and pm_get_Tdata_group Subroutine" on page 1028, "pm_start_group and pm_tstart_group Subroutine" on page 1102, "pm_stop_group and pm_tstop_group Subroutine" on page 1111, "pm_reset_data_group Subroutine" on page 1073.


pm_set_program_group_mx Subroutine

Purpose
Sets Performance Monitor programmation in counter multiplexing mode for a target thread and creates a counting group.

Library
Performance Monitor APIs Library (libpmapi.a)

Syntax
#include <pmapi.h>

int pm_set_program_group_mx (pid_t pid, tid_t tid, *prog)

Description
This subroutine supports only the 1:1 threading model. It has been superseded by the pm_set_program_pgroup_mx subroutine, which supports both the 1:1 and the M:N threading models. A call to this subroutine is equivalent to a call to the pm_set_program_pgroup_mx subroutine with a ptid parameter equal to 0.

The pm_set_program_group_mx subroutine sets the Performance Monitor programmation in counter multiplexing mode for a target kernel thread. The thread must be stopped and must be part of a debuggee process, under the control of the calling process. The setting includes the list of the event arrays to be counted and a mode in which to count. The events to count are in an array of list of event identifiers. The identifiers must be selected from the lists returned by the pm_initialize subroutine.

This call also creates a counting group, which includes the target thread and any thread which it, or any of its descendants, will create in the future. Optionally, the group can be defined as also containing all the existing and future threads belonging to the target process.
The counting mode includes User Mode and/or Kernel Mode, and the Initial Counting State. The defaults are set to Off for User Mode and Kernel Mode, and the initial default state is set to delay counting until the `pm_start_group` subroutine is called.

If the list includes an event which can be used with a threshold (as indicated by the `pm_init` subroutine), a threshold value can also be specified.

**Parameters**

- `pid`  
  Process ID of target thread. Target process must be a debuggee of a calling process.

- `tid`  
  Thread ID of target thread.

- `*prog`
  
  - `PM_USER`  
    Counts processes running in User Mode (default is set to Off)

  - `PM_KERNEL`  
    Counts processes running in Kernel Mode (default is set to Off)

  - `PM_COUNT`  
    Starts counting immediately (default is set to Not to Start Counting)

  - `PM_PROCESS`  
    Creates a process-level counting group

**Return Values**

- **0**  
  Operation completed successfully.

- Positive Error Code  
  Refer to the `pm_error` subroutine to decode the error code.

**Error Codes**

Refer to the `pm_error Subroutine` on page 1024.

**Files**

- `/usr/include/pmapi.h`  
  Defines standard macros, data types, and subroutines.

**Related Information**

The `pm_init Subroutine` on page 1069, `pm_error Subroutine` on page 1024, `pm_get_program_group_mx Subroutine` on page 1052, `pm_delete_program_group Subroutine` on page 1018, `pm_get_data_group_mx and pm_get_tdata_group_mx Subroutine` on page 1030, `pm_start_group and pm_tstart_group Subroutine` on page 1102, `pm_stop_group and pm_tstop_group Subroutine` on page 1111, `pm_reset_data_group Subroutine` on page 1073.

pm_set_program_mx Subroutine

Purpose
Sets system wide Performance Monitor programmation in counter multiplexing mode.

Library
Performance Monitor APIs Library (libpmapi.a)

Syntax
```c
#include <pmapi.h>

int pm_set_program_mx ( *prog )
pm_prog_mx_t *prog;
```

Description
The `pm_set_program_mx` subroutine sets system wide Performance Monitor programmation in counter multiplexing mode. The setting includes the list of the event arrays to be counted, and a mode in which to count. The events to count are in an array of list of event identifiers. The identifiers must be selected from the lists returned by the `pm_initialize` subroutine.

The counting mode includes User Mode and/or Kernel Mode, the Initial Counting State, and the Process Tree Mode. The Process Tree Mode sets counting to On only for the calling process and its descendants. The defaults are set to Off for User Mode and Kernel Mode. The initial default state is set to delay counting until the `pm_start` subroutine is called, and to count the activity of all the processes running in the system.

If the list includes an event which can be used with a threshold (as indicated by the `pm_init` subroutine), a threshold value can also be specified.

On some platforms, event groups can be specified instead of individual events. This is done by setting the bitfield `is_group` in the mode, and placing the group ID into the first element of each events array. (The group ID was obtained by `pm_init`).

Parameters

* `prog` Specifies the events and modes to use in Performance Monitor setup. The following modes are supported:

  - **PM_USER**
    Counts processes running in User Mode (default is set to Off)

  - **PM_KERNEL**
    Counts processes running in Kernel Mode (default is set to Off)

  - **PM_COUNT**
    Starts counting immediately (default is set to Not to Start Counting)

  - **PM_PROCTREE**
    Sets counting to On only for the calling process and its descendants (default is set to Off)
Return Values

0 Operation completed successfully.
Positive Error Code Refer to the "pm_error Subroutine" on page 1024 to decode the error code.

Error Codes
Refer to the "pm_error Subroutine" on page 1024.

Files
/usr/include/pmapi.h Defines standard macros, data types, and subroutines.

Related Information
The "pm_init Subroutine" on page 1069, "pm_error Subroutine" on page 1024, "pm_get_program_mx Subroutine" on page 1053, "pm_delete_program Subroutine" on page 1017, "pm_get_data_mx, pm_get_tdata_mx, pm_get_data_cpu_mx, pm_get_tdata_cpu_mx, pm_get_data_lcpu_mx and pm_get_tdata_lcpu_mx Subroutine" on page 1031, "pm_start and pm_tstart Subroutine" on page 1101, "pm_stop and pm_tstop Subroutine" on page 1110, "pm_reset_data Subroutine" on page 1073.


pm_set_program_mygroup Subroutine

Purpose
Sets Performance Monitor programmation for the calling thread and creates a counting group.

Library
Performance Monitor APIs Library (libpmapi.a)

Syntax
#include <pmapi.h>

int pm_set_program_mygroup ( pmprog_t *prog);

Description
The pm_set_program_mygroup subroutine sets the Performance Monitor programmation for the calling kernel thread. The setting includes the events to be counted and a mode in which to count. The events to count are in a list of event identifiers. The identifiers must be selected from the lists returned by the pm_init subroutine.

This call also creates a counting group, which includes the calling thread and any thread which it, or any of its descendants, will create in the future. Optionally, the group can be defined as also containing all the existing and future threads belonging to the calling process.

The counting mode includes User Mode and/or Kernel Mode, and the Initial Counting State. The defaults are set to Off for User Mode and Kernel Mode, and the initial default state is set to delay counting until the pm_start_mygroup subroutine is called.
If the list includes an event which can be used with a threshold (as indicated by the `pm_init` subroutine), a threshold value can also be specified.

**Parameters**

`*prog` Specifies the events and mode to use in Performance Monitor setup. The following modes are supported:

- **PM_USER**
  - Counts processes running in User Mode (default is set to Off)

- **PM_KERNEL**
  - Counts processes running in Kernel Mode (default is set to Off)

- **PM_COUNT**
  - Starts counting immediately (default is set to Not to Start Counting)

- **PM_PROCESS**
  - Creates a process-level counting group

**Return Values**

- **0** Operation completed successfully.
- Positive error code Refer to the `pm_error Subroutine` on page 1024 to decode the error code.

**Error Codes**

Refer to the `pm_error Subroutine" on page 1024.

**Files**

`/usr/include/pmapi.h` Defines standard macros, data types, and subroutines.

**Related Information**

The `pm_init Subroutine” on page 1069, `pm_error Subroutine" on page 1024, `pm_get_program_mygroup Subroutine" on page 1055, `pm_delete_program_mygroup Subroutine" on page 1019, `pm_get_data_mygroup, pm_get_tdata_mygroup or pm_get_Tdata_mygroup Subroutine" on page 1033, `pm_start_mygroup and pm_tstart_mygroup Subroutine" on page 1104, `pm_stop_mygroup and pm_tstop_mygroup Subroutine’ on page 1112, `pm_reset_data_mygroup Subroutine’ on page 1075.


**pm_set_program_mygroup_mx Subroutine**

**Purpose**

Sets Performance Monitor programmation in counter multiplexing mode for the calling thread and creates a counting group.

**Library**

Performance Monitor APIs Library (`libpmapi.a`)
Syntax
#include <pmapi.h>

int pm_set_program_mygroup_mx (*prog)
pm_prog_mx_t *prog;

Description
The pm_set_program_mygroup_mx subroutine sets the Performance Monitor programmation in counter multiplexing mode for the calling kernel thread. The setting includes the list of event arrays to be counted and a mode in which to count. The events to count are in an array of list of event identifiers. The identifiers must be selected from the lists returned by the pm_initialize subroutine.

This call also creates a counting group, which includes the calling thread and any thread which it, or any of its descendants, will create in the future. Optionally, the group can be defined as also containing all the existing and future threads belonging to the calling process.

The counting mode includes User Mode and/or Kernel Mode, and the Initial Counting State. The defaults are set to Off for User Mode and Kernel Mode, and the initial default state is set to delay counting until the pm_start_mygroup subroutine is called.

If the list includes an event which can be used with a threshold (as indicated by the pm_init subroutine), a threshold value can also be specified.

Parameters
*prog
Specifies the events and mode to use in Performance Monitor setup. The following modes are supported:

PM_USER
Counts processes running in User Mode (default is set to Off)

PM_KERNEL
Counts processes running in Kernel Mode (default is set to Off)

PM_COUNT
Starts counting immediately (default is set to Not to Start Counting)

PM_PROCESS
Creates a process-level counting group

Return Values
0 Operation completed successfully.
Positive Error Code Refer to the pm_error Subroutine on page 1024 to decode the error code.

Error Codes
Refer to the pm_error Subroutine on page 1024.

Files
/usr/include/pma.pi.h Defines standard macros, data types, and subroutines.
pm_set_program_mythread Subroutine

Purpose
Sets Performance Monitor programmation for the calling thread.

Library
Performance Monitor APIs Library (libpmapi.a)

Syntax
```
#include <pmapi.h>

int pm_set_program_mythread ( pm_prog_t *prog);
```

Description
The pm_set_program_mythread subroutine sets the Performance Monitor programmation for the calling kernel thread. The setting includes the events to be counted, and a mode in which to count. The events to count are in a list of event identifiers. The identifiers must be selected from the lists returned by the pm_init subroutine.

The counting mode includes User Mode and/or Kernel Mode, and the Initial Counting State. The defaults are set to Off for User Mode and Kernel Mode, and the initial default state is set to delay counting until the pm_start_mythread subroutine is called.

If the list includes an event which can be used with a threshold (as indicated by the pm_init subroutine), a threshold value can also be specified.
Parameters

*prog

Specifies the event modes to use in Performance Monitor setup. The following modes are supported:

**PM_USER**
Counts processes running in User Mode (default is set to Off)

**PM_KERNEL**
Counts processes running in Kernel Mode (default is set to Off)

**PM_COUNT**
Starts counting immediately (default is set to Not to Start Counting)

**PM_PROCESS**
Creates a process-level counting group

Return Values

0  Operation completed successfully.

Positive error code  Refer to the "pm_error Subroutine" on page 1024 to decode the error code.

Error Codes

Refer to the "pm_error Subroutine" on page 1024.

Files

/usr/include/pma.h  Defines standard macros, data types, and subroutines.

Related Information

The "pm_init Subroutine" on page 1069, "pm_error Subroutine" on page 1024, "pm_get_program_mythread Subroutine" on page 1057, "pm_delete_program_mythread Subroutine" on page 1020, "pm_get_data_mythread, pm_get_tdata_mythread or pm_get_Tdata_mythread Subroutine" on page 1036, "pm_start_mythread and pm_tstart_mythread Subroutine" on page 1105, "pm_stop_mythread and pm_tstop_mythread Subroutine" on page 1113, "pm_reset_data_mythread Subroutine" on page 1076.


**pm_set_program_mythread_mx Subroutine**

**Purpose**
Sets Performance Monitor programmation in counter multiplexing mode for the calling thread.

**Library**
Performance Monitor APIs Library (libpmapi.a)
Syntax

```c
#include <pmapi.h>

int pm_set_program_mythread_mx (*prog)
    pm_prog_mx_t *prog;
```

Description

The `pm_set_program_mythread_mx` subroutine sets the Performance Monitor programmation in counter multiplexing mode for the calling kernel thread. The setting includes the list of the event arrays to be counted, and a mode in which to count. The events to count are in an array of list of event identifiers. The identifiers must be selected from the lists returned by the `pm_initialize` subroutine.

The counting mode includes User Mode and/or Kernel Mode, and the Initial Counting State. The defaults are set to Off for User Mode and Kernel Mode, and the initial default state is set to delay counting until the `pm_start_mythread` subroutine is called.

If the list includes an event which can be used with a threshold (as indicated by the `pm_init` subroutine), a threshold value can also be specified.

Parameters

*prog

Specifies the event modes to use in Performance Monitor setup. The following modes are supported:

- **PM_USER**
  - Counts processes running in User Mode (default is set to Off)

- **PM_KERNEL**
  - Counts processes running in Kernel Mode (default is set to Off)

- **PM_COUNT**
  - Starts counting immediately (default is set to Not to Start Counting)

- **PM_PROCESS**
  - Creates a process-level counting group

Return Values

- **0** Operation completed successfully.

- Positive Error Code Refer to the "pm_error Subroutine" on page 1024 to decode the error code.

Error Codes

Refer to the "pm_error Subroutine" on page 1024.

Files

- `/usr/include/pmapi.h` Defines standard macros, data types, and subroutines.

Related Information

The "pm_init Subroutine" on page 1069, "pm_error Subroutine" on page 1024, "pm_get_program_mythread_mx Subroutine" on page 1058, "pm_delete_program_mythread Subroutine" on page 1020
pm_set_program_pgroup Subroutine

Purpose
Sets Performance Monitor programmation for a target pthread and creates a counting group.

Library
Performance Monitor APIs Library (libpmapi.a)

Syntax
#include <pmapi.h>

int pm_set_program_pgroup ( pid, tid, ptid, *prog )

pid_t pid;
 tid_t tid;
 ptid_t ptid;
 pm_prog_t *prog;

Description
The pm_set_program_pgroup subroutine sets the Performance Monitor programmation for a target pthread. The pthread must be stopped and must be part of a debuggee process, under the control of the calling process. The setting includes the events to be counted and a mode in which to count. The events to count are in a list of event identifiers. The identifiers must be selected from the lists returned by the pm_initialize subroutine.

This call also creates a counting group, which includes the target pthread and any pthread that it, or any of its descendants, will create in the future. Optionally, the group can be defined as also containing all the existing and future pthreads belonging to the target process.

If the pthread is running in 1:1 mode, only the tid parameter must be specified. If the pthread is running in m:n mode, only the ptid parameter must be specified. If both the ptid and tid parameters are specified, they must be referring to a single pthread with the ptid parameter specified and currently running on a kernel thread with specified tid parameter.

The counting mode includes User Mode and/or Kernel Mode, and the Initial Counting State. The defaults are set to Off for User Mode and Kernel Mode, and the initial default state is set to delay counting until the pm_start_pgroup subroutine is called.

If the list includes an event that can be used with a threshold (as indicated by the pm_initialize subroutine), a threshold value can also be specified.

Parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>pid</td>
<td>Process ID of target pthread. Target process must be a debuggee of the caller process.</td>
</tr>
<tr>
<td>tid</td>
<td>Thread ID of target pthread. To ignore this parameter, set it to 0.</td>
</tr>
</tbody>
</table>
**ptid**

Pthread ID of the target pthread. To ignore this parameter, set it to 0.

**‘prog**

Specifies the event modes to use in Performance Monitor setup. The following modes are supported:

- **PM_USER**
  Counts processes running in User Mode (default is set to Off)

- **PM_KERNEL**
  Counts processes running in Kernel Mode (default is set to Off)

- **PM_COUNT**
  Starts counting immediately (default is set to Not to Start Counting)

- **PM_PROCESS**
  Creates a process-level counting group

**Return Values**

<table>
<thead>
<tr>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Operation completed successfully.</td>
</tr>
<tr>
<td>Positive error code</td>
<td>Refer to the pm_error Subroutine on page 1024 to decode the error code.</td>
</tr>
</tbody>
</table>

**Error Codes**

Refer to the pm_error Subroutine on page 1024.

**Files**

```
/usr/include/pmapi.h
```

Defines standard macros, data types, and subroutines.

**Related Information**

The pm_delete_program_pgroup Subroutine on page 1021, pm_error Subroutine on page 1024, pm_get_data_pgroup, pm_get_tdata_pgroup, and pm_get_Tdata_pgroup Subroutine on page 1039, pm_get_program_pgroup Subroutine on page 1060, pm_initialize Subroutine on page 1071, pm_reset_data_pgroup Subroutine on page 1076, pm_start_pgroup and pm_tstart_pgroup Subroutine on page 1106, pm_stop_pgroup and pm_tstop_pgroup Subroutine on page 1114.


**pm_set_program_pgroup_mx Subroutine**

**Purpose**

Sets Performance Monitor programmation in counter multiplexing mode for a target pthread and creates a counting group.

**Library**

Performance Monitor APIs Library (libpmapi.a)
Syntax
#include <pmapi.h>

int pm_set_program_pgroup_mx (pid, tid, ptid, *prog)
pid_t pid;
tid_t tid;
ptid_t ptid;
pm_prog_mx_t *prog;

Description
The pm_set_program_pgroup_mx subroutine sets the Performance Monitor programmation in counter multiplexing mode for a target pthread. The pthread must be stopped and must be part of a debuggee process, under the control of the calling process. The setting includes the list of the event arrays to be counted and a mode in which to count. The events to count are in an array of list of event identifiers. The identifiers must be selected from the lists returned by the pm_initialze subroutine.

This call also creates a counting group, which includes the target pthread and any pthread that it, or any of its descendants, will create in the future. Optionally, the group can be defined as also containing all the existing and future pthreads belonging to the target process.

If the pthread is running in 1:1 mode, only the tid parameter must be specified. If the pthread is running in m:n mode, only the ptid parameter must be specified. If both the ptid and tid parameters are specified, they must be referring to a single pthread with the ptid parameter specified and currently running on a kernel thread with specified tid parameter.

The counting mode includes User Mode and/or Kernel Mode, and the Initial Counting State. The defaults are set to Off for User Mode and Kernel Mode, and the initial default state is set to delay counting until the pm_start_pgroup subroutine is called.

If the list includes an event that can be used with a threshold (as indicated by the pm_initialize subroutine), a threshold value can also be specified.

Parameters

pid
Process ID of target pthread. Target process must be a debuggee of the caller process.

tid
Thread ID of target pthread. To ignore this parameter, set it to 0.

ptid
Pthread ID of the target pthread. To ignore this parameter, set it to 0.

*prog
Specifies the event modes to use in Performance Monitor setup. The following modes are supported:

PM_USER
Counts processes running in User Mode (default is set to Off)

PM_KERNEL
Counts processes running in Kernel Mode (default is set to Off)

PM_COUNT
Starts counting immediately (default is set to Not to Start Counting)

PM_PROCESS
Creates a process-level counting group
Return Values

0 Operation completed successfully.

Positive Error Code Refer to the "pm_error Subroutine" on page 1024 to decode the error code.

Error Codes
Refer to the "pm_error Subroutine" on page 1024.

Files

/usr/include/pmapi.h Defines standard macros, data types, and subroutines.

Related Information
The "pm_delete_program_pgroup Subroutine" on page 1021, "pm_error Subroutine" on page 1024, "pm_get_data_pgroup_mx and pm_get_tdata_pgroup_mx Subroutine" on page 1041, "pm_get_program_pgroup_mx Subroutine" on page 1061, "pm_initialize Subroutine" on page 1071, "pm_reset_data_pgroup Subroutine" on page 1076, "pm_start_pgroup and pm_tstart_pgroup Subroutine" on page 1106, "pm_stop_pgroup and pm_tstop_pgroup Subroutine" on page 1114.


pm_set_program_pthread Subroutine

Purpose
Sets Performance Monitor programmation for a target pthread.

Library
Performance Monitor APIs Library (libpmapi.a)

Syntax

#include <pmapi.h>

int pm_set_program_pthread (pid, tid, ptid, *prog)

pid_t pid;
tid_t tid;
ptid_t ptid;
pm_prog_t *prog;

Description
The pm_set_program_pthread subroutine sets the Performance Monitor programmation for a target pthread. The pthread must be stopped and must be part of a debuggee process, under the control of the calling process. The setting includes the events to be counted and a mode in which to count. The events to count are in a list of event identifiers. The identifiers must be selected from the lists returned by the pm_initialize subroutine.

If the pthread is running in 1:1 mode, only the tid parameter must be specified. If the pthread is running in m:n mode, only the ptid parameter must be specified. If both the ptid and tid parameters are specified, they must be referring to a single pthread with the ptid parameter specified and currently running on a kernel thread with specified tid parameter.
The counting mode includes User Mode and/or Kernel Mode, and the Initial Counting State. The defaults are set to Off for User Mode and Kernel Mode, and the Initial Default State is set to delay counting until the `pm_start_pthread` subroutine is called.

If the list includes an event which can be used with a threshold (as indicated by the `pm_initialize` subroutine), a threshold value can also be specified.

**Parameters**

- **pid**
  - Process ID of target pthread. Target process must be a debuggee of the caller process.

- **tid**
  - Thread ID of target pthread. To ignore this parameter, set it to 0.

- **ptid**
  - Pthread ID of the target pthread. To ignore this parameter, set it to 0.

- ***prog**
  - Specifies the event modes to use in Performance Monitor setup. The following modes are supported:
    - **PM_USER**
      - Counts processes running in User Mode (default is set to Off)
    - **PM_KERNEL**
      - Counts processes running in Kernel Mode (default is set to Off)
    - **PM_COUNT**
      - Starts counting immediately (default is set to Not to Start Counting)

**Return Values**

- **0**
  - Operation completed successfully.
- **Positive error code**
  - Refer to the "pm_error Subroutine" on page 1024 to decode the error code.

**Error Codes**

Refer to the "pm_error Subroutine" on page 1024.

**Files**

- `/usr/include/pmapi.h`
  - Defines standard macros, data types, and subroutines.

**Related Information**

The "pm_delete_program pthread Subroutine" on page 1022, "pm_error Subroutine" on page 1024, "pm_get_data pthread, pm_get_tdata pthread or pm_get_Tdata pthread Subroutine" on page 1043, "pm_get_program pthread Subroutine" on page 1063, "pm_initialize Subroutine" on page 1071, "pm_reset_data pthread Subroutine" on page 1078, "pm_start pthread and pm_tstart pthread Subroutine" on page 1107, "pm_stop pthread and pm_tstop pthread Subroutine" on page 1116.

pm_set_program_pthread_mx Subroutine

Purpose
Sets Performance Monitor programmation in counter multiplexing mode for a target pthread.

Library
Performance Monitor APIs Library (libpmapi.a)

Syntax
```
#include <pmapi.h>

int pm_set_program_pthread_mx (pid, tid, ptid, *prog);
```

Description
The `pm_set_program_pthread_mx` subroutine sets the Performance Monitor programmation in counter multiplexing mode for a target pthread. The pthread must be stopped and must be part of a debuggee process, under the control of the calling process. The setting includes the list of the event arrays events to be counted and a mode in which to count. The events to count are in an array of list of event identifiers. The identifiers must be selected from the lists returned by the `pm_initialize` subroutine.

If the pthread is running in 1:1 mode, only the `tid` parameter must be specified. If the pthread is running in m:n mode, only the `ptid` parameter must be specified. If both the `ptid` and `tid` parameters are specified, they must be referring to a single pthread with the `ptid` parameter specified and currently running on a kernel thread with specified `tid` parameter.

The counting mode includes User Mode and/or Kernel Mode, and the Initial Counting State. The defaults are set to Off for User Mode and Kernel Mode, and the Initial Default State is set to delay counting until the `pm_start_pthread` subroutine is called.

If the list includes an event which can be used with a threshold (as indicated by the `pm_initialize` subroutine), a threshold value can also be specified.

Parameters
```
pid
    Process ID of target pthread. Target process must be a debuggee of the caller process.

tid
    Thread ID of target pthread. To ignore this parameter, set it to 0.

ptid
    Pthread ID of the target pthread. To ignore this parameter, set it to 0.
```


Specifies the event modes to use in Performance Monitor setup. The following modes are supported:

**PM_USER**
Counts processes running in User Mode (default is set to Off)

**PM_KERNEL**
Counts processes running in Kernel Mode (default is set to Off)

**PM_COUNT**
Starts counting immediately (default is set to Not to Start Counting)

### Return Values

<table>
<thead>
<tr>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Operation completed successfully.</td>
</tr>
<tr>
<td>Positive</td>
<td>Positive error code Refer to the pm_error Subroutine on page 1024 to decode the error code.</td>
</tr>
</tbody>
</table>

### Error Codes

Refer to the pm_error Subroutine on page 1024.

### Files

`/usr/include/pmafi.h`
Defines standard macros, data types, and subroutines.

### Related Information

The pm_delete_program_pthread Subroutine on page 1022, pm_error Subroutine on page 1024, pm_get_data_pthread mx or pm_get_tdata_pthread mx Subroutine on page 1044, pm_get_program_pthread_mx Subroutine on page 1064, pm_initialize Subroutine on page 1071, pm_reset_data_pthread Subroutine on page 1078, pm_start_pthread and pm_tstart_pthread Subroutine on page 1107, pm_stop_pthread and pm_tstop_pthread Subroutine on page 1116.


### pm_set_program_thread Subroutine

#### Purpose
Sets Performance Monitor programmation for a target thread.

#### Library
Performance Monitor APIs Library (libpmapi.a)

#### Syntax

```c
#include <pmafi.h>

int pm_set_program_thread (pid_t pid, tid_t tid, pm_prog_t *prog);
```

pid_t pid;
tid_t tid;
pm_prog_t *prog;

Technical Reference, Volume 1: Base Operating System and Extensions
Description
This subroutine supports only the 1:1 threading model. It has been superseded by the
\texttt{pm\_set\_program\_pthread} subroutine, which supports both the 1:1 and the M:N threading models. A call
to this subroutine is equivalent to a call to the \texttt{pm\_set\_program\_pthread} subroutine with a \texttt{ptid} parameter
equal to 0.

The \texttt{pm\_set\_program\_thread} subroutine sets the Performance Monitor programmation for a target kernel
thread. The thread must be stopped and must be part of a debuggee process, under the control of the
calling process. The setting includes the events to be counted and a mode in which to count. The events
to count are in a list of event identifiers. The identifiers must be selected from the lists returned by the
\texttt{pm\_init} subroutine.

The counting mode includes User Mode and/or Kernel Mode, and the Initial Counting State. The defaults
are set to Off for User Mode and Kernel Mode, and the Initial Default State is set to delay counting until
the \texttt{pm\_start\_thread} subroutine is called.

If the list includes an event which can be used with a threshold (as indicated by the \texttt{pm\_init} subroutine), a
threshold value can also be specified.

Parameters

\begin{itemize}
\item \texttt{pid} \hspace{4cm} \text{Process ID of target thread. Target process must be a}
\text{debuggee of the caller process.}
\item \texttt{tid} \hspace{4cm} \text{Thread ID of target thread.}
\item \texttt{*prog} \hspace{4cm} \text{Specifies the event modes to use in Performance Monitor}
\text{setup. The following modes are supported:}
\begin{itemize}
\item \texttt{PM\_USER} \hspace{4cm} \text{Counts processes running in User Mode (default}
\text{is set to Off)}
\item \texttt{PM\_KERNEL} \hspace{4cm} \text{Counts processes running in Kernel Mode}
\text{(default is set to Off)}
\item \texttt{PM\_COUNT} \hspace{4cm} \text{Starts counting immediately (default is set to Not}
\text{to Start Counting)}
\end{itemize}
\end{itemize}

Return Values

\begin{itemize}
\item 0 \hspace{4cm} \text{Operation completed successfully.}
\item \textbf{Positive Error Code} \hspace{4cm} \text{Refer to the \texttt{pm\_error} \textit{("pm\_error Subroutine" on page 1024)} subroutine to decode the}
\text{error code.}
\end{itemize}

Error Codes

Refer to the \texttt{pm\_error} \textit{("pm\_error Subroutine" on page 1024)} subroutine.

Files

\texttt{/usr/include/pmapi.h} \hspace{4cm} \text{Defines standard macros, data types, and subroutines.}
pm_set_program_thread_mx Subroutine

Purpose
Sets Performance Monitor programation in counter multiplexing mode for a target thread.

Library
Performance Monitor APIs Library (libpmapi.a)

Syntax
#include <pmapi.h>

int pm_set_program_thread_mx (pid_t pid, tid_t tid, *prog);

Description
This subroutine supports only the 1:1 threading model. It has been superseded by the pm_set_program_pthread_mx subroutine, which supports both the 1:1 and the M:N threading models. A call to this subroutine is equivalent to a call to the pm_set_program_pthread_mx subroutine with a ptid parameter equal to 0.

The pm_set_program_thread_mx subroutine sets the Performance Monitor programation in counter multiplexing mode for a target kernel thread. The thread must be stopped and must be part of a debuggee process, under the control of the calling process. The setting includes the list of the event arrays to be counted and a mode in which to count. The events to count are in an array of list of event identifiers. The identifiers must be selected from the lists returned by the pm_initialize subroutine.

The counting mode includes User Mode and/or Kernel Mode, and the Initial Counting State. The defaults are set to Off for User Mode and Kernel Mode, and the Initial Default State is set to delay counting until the pm_start_thread subroutine is called.

If the list includes an event which can be used with a threshold (as indicated by the pm_init subroutine), a threshold value can also be specified.

Parameters

pid Process ID of target thread. Target process must be a debuggee of the caller process.
tid
'prog

Thread ID of target thread.

Specifies the event modes to use in Performance Monitor setup. The following modes are supported:

PM_USER
Counts processes running in User Mode (default is set to Off)

PM_KERNEL
Counts processes running in Kernel Mode (default is set to Off)

PM_COUNT
Starts counting immediately (default is set to Not to Start Counting)

Return Values

0 Operation completed successfully.
Positive error code Refer to the "pm_error Subroutine" on page 1024 to decode the error code.

Error Codes

Refer to the "pm_error Subroutine" on page 1024.

Files

/usr/include/pmapi.h Defines standard macros, data types, and subroutines.

Related Information

The "pm_init Subroutine" on page 1069, "pm_error Subroutine" on page 1024, "pm_get_program_thread_mx Subroutine" on page 1067, "pm_delete_program_thread Subroutine" on page 1023, "pm_get_data_thread_mx or pm_get_tdata_thread_mx Subroutine" on page 1048, "pm_start_thread and pm_tstart_thread Subroutine" on page 1109, "pm_stop_thread and pm_tstop_thread Subroutine" on page 1117, "pm_reset_data_thread Subroutine" on page 1079.


pm_start and pm_tstart Subroutine

Purpose
Starts system wide Performance Monitor counting.

Library
Performance Monitor APIs Library (libpmapi.a)

Syntax

#include <pmapi.h>

int pm_start()
int pm_tstart(timebasestruct_t *time);
Description

The pm_start subroutine starts system wide Performance Monitor counting.

The pm_tstart subroutine starts system wide Performance Monitor counting, and returns a timestamp indicating when the counting was started.

Parameters

*time* Pointer to a structure containing the timebase value when the counting was started. This can be converted to time using the time_base_to_time subroutine.

Return Values

0 Operation completed successfully.
Positive error code Refer to the "pm_error Subroutine" on page 1024 to decode the error code

Error Codes

Refer to the "pm_error Subroutine" on page 1024.

Files

/usr/include/pmapi.h Defines standard macros, data types, and subroutines.

Related Information

The "pm_init Subroutine" on page 1069, "pm_error Subroutine" on page 1024, "pm_set_program Subroutine" on page 1080, "pm_get_program Subroutine" on page 1049, "pm_delete_program Subroutine" on page 1017, "pm_get_data, pm_get_tdata, pm_get_Tdata, pm_get_data_cpu, pm_get_tdata_cpu, pm_get_Tdata_cpu, pm_get_data_lcpu, pm_get_tdata_lcpu and pm_get_Tdata_lcpu Subroutine" on page 1025, "pm_stop and pm_tstop Subroutine " on page 1110, "pm_reset_data Subroutine" on page 1073.


---

pm_start_group and pm_tstart_group Subroutine

Purpose

Starts Performance Monitor counting for the counting group to which a target thread belongs.

Library

Performance Monitor APIs Library (libpmapi.a)

Syntax

#include <pmapi.h>

int pm_start_group ( pid, tid)
pid_t pid;
tid_t tid;

int pm_tstart_group ( pid, tid, *time)


Description
This subroutine supports only the 1:1 threading model. It has been superseded by the `pm_start_pgroup` subroutine, which supports both the 1:1 and the M:N threading models. A call to this subroutine is equivalent to a call to the `pm_start_pgroup` subroutine with a `ptid` parameter equal to 0.

The `pm_start_group` subroutine starts the Performance Monitor counting for a target kernel thread and the counting group to which it belongs. This counting is effective immediately for the target thread. For all the other thread members of the counting group, the counting will start after their next redispacht, but only if their current counting state is already set to On. The counting state of a thread in a group is obtained by ANDing the thread counting state with the group state. If their counting state is currently set to Off, no counting starts until they call either the `pm_start_mythread` subroutine or the `pm_start_mygroup` subroutine, or until a debugger process calls the `pm_start_thread` subroutine or the `pm_start_group` subroutine on their behalf.

The `pm_tstart_group` subroutine starts the Performance Monitor counting for a target kernel thread and the counting group to which it belongs, and returns a timestamp indicating when the counting was started.

Parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>pid</code></td>
<td>Process ID of target thread. Target process must be a debuggee of the caller process.</td>
</tr>
<tr>
<td><code>tid</code></td>
<td>Thread ID of target thread.</td>
</tr>
<tr>
<td><code>*time</code></td>
<td>Pointer to a structure containing the timebase value when the counting was started. This can be converted to time using the <code>time_base_to_time</code> subroutine.</td>
</tr>
</tbody>
</table>

Return Values

<table>
<thead>
<tr>
<th>Return Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Operation completed successfully.</td>
</tr>
<tr>
<td>Positive error code</td>
<td>Refer to the &quot;pm_error Subroutine&quot; on page 1024 to decode the error code.</td>
</tr>
</tbody>
</table>

Error Codes

Refer to the "pm_error Subroutine" on page 1024.

Files

`/usr/include/pma$i$h` Defines standard macros, data types, and subroutines.

Related Information

The "pm_init Subroutine" on page 1069, "pm_error Subroutine" on page 1024, "pm_set_program_group Subroutine" on page 1081, "pm_get_program_group Subroutine" on page 1051, "pm_delete_program_group Subroutine" on page 1018, "pm_get_data_group, pm_get_tdata_group and pm_get_Tdata_group Subroutine" on page 1028, "pm_stop_group and pm_tstop_group Subroutine" on page 1111, "pm_reset_data_group Subroutine" on page 1073.

pm_start_mygroup and pm_tstart_mygroup Subroutine

Purpose
Starts Performance Monitor counting for the group to which the calling thread belongs.

Library
Performance Monitor APIs Library (libpmapi.a)

Syntax
#include <pmapi.h>

int pm_start_mygroup()

int pm_tstart_mygroup(timebasestruct_t *time)

Description
The pm_start_mygroup subroutine starts the Performance Monitor counting for the calling kernel thread and the counting group to which it belongs. Counting is effective immediately for the calling thread. For all the other threads members of the counting group, the counting starts after their next redispach, but only if their current counting state is already set to On. The counting state of a thread in a group is obtained by ANDing the thread counting state with the group state. If their counting state is currently set to Off, no counting starts until they call either the pm_start_mythread subroutine or the pm_start_mygroup subroutine themselves, or until a debugger process calls the pm_start_thread subroutine or the pm_start_group subroutine on their behalf.

The pm_tstart_mygroup subroutine starts the Performance Monitor counting for the calling kernel thread and the counting group to which it belongs, and returns a timestamp indicating when the counting was started.

Parameters
*time
Pointer to a structure containing the timebase value when the counting was started. This can be converted to time using the time_base_to_time subroutine.

Return Values
0
Operation completed successfully.
Positive error code
Refer to the pm_error Subroutine on page 1024 to decode the error code.

Error Codes
Refer to the pm_error Subroutine on page 1024.

Files
/usr/include/pmapi.h
Defines standard macros, data types, and subroutines.

Related Information
The pm_init Subroutine on page 1069, pm_error Subroutine on page 1024, pm_set_program_mygroup Subroutine on page 1086, pm_get_program_mygroup Subroutine on page 1055.
pm_start_mythread and pm_tstart_mythread Subroutine

Purpose
Starts Performance Monitor counting for the calling thread.

Library
Performance Monitor APIs Library (libpmapi.a)

Syntax
#include <pmapi.h>

int pm_start_mythread()

int pm_tstart_mythread(timebasestruct_t *time);

Description
The pm_start_mythread subroutine starts Performance Monitor counting for the calling kernel thread. Counting is effective immediately unless the thread is in a group, and that group's counting is not currently set to On. The counting state of a thread in a group is obtained by ANDing the thread counting state with the group state.

The pm_tstart_mythread subroutine starts Performance Monitor counting for the calling kernel thread, and returns a timestamp indicating when the counting was started.

Parameters
*time Pointer to a structure containing the timebase value when the counting was started. This can be converted to time using the time_base_to_time subroutine.

Return Values
0 Operation completed successfully.
Positive Error Code Refer to the pm_error subroutine to decode the error code.

Error Codes
Refer to the pm_error subroutine

Files
/usr/include/pmapi.h Defines standard macros, data types, and subroutines.
Related Information

The pm_init subroutine, pm_error subroutine, pm_set_program_mythread subroutine, pm_get_program_mythread subroutine, pm_delete_program_mythread subroutine, pm_get_data_mythread subroutine, pm_stop_mythread subroutine, and pm_reset_data_mythread subroutine.


pm_start_pgroup and pm_tstart_pgroup Subroutine

Purpose

Starts Performance Monitor counting for the counting group to which a target pthread belongs.

Library

Performance Monitor APIs Library (libpmapi.a)

Syntax

```
#include <pmapi.h>

int pm_start_pgroup (pid, tid, ptid)
pid_t pid;
tid_t tid;
ptid_t ptid;

int pm_tstart_pgroup (pid, tid, ptid, *time)
pid_t pid;
tid_t tid;
ptid_t ptid;
timebasestruct_t *time
```

Description

The pm_start_pgroup subroutine starts the Performance Monitor counting for a target pthread and the counting group to which it belongs. This counting is effective immediately for the target pthread. For all the other thread members of the counting group, the counting will start after their next redispach, but only if their current counting state is already set to On. The counting state of a pthread in a group is obtained by ANDing the pthread counting state with the group state. If their counting state is currently set to Off, no counting starts until they call either the pm_start_mythread subroutine or the pm_start_mygroup subroutine on their behalf.

The pm_tstart_pgroup subroutine starts the Performance Monitor counting for a target pthread and the counting group to which it belongs, and returns a timestamp indicating when the counting was started.

If the pthread is running in 1:1 mode, only the tid parameter must be specified. If the pthread is running in m:n mode, only the ptid parameter must be specified. If both the ptid and tid parameters are specified, they must be referring to a single pthread with the ptid parameter specified and currently running on a kernel thread with specified tid parameter.
Parameters

pid
Process ID of target pthread. Target process must be a debuggee of the caller process.
tid
Thread ID of target pthread. To ignore this parameter, set it to 0.
ptid
Pthread ID of the target pthread. To ignore this parameter, set it to 0.
*time
Pointer to a structure containing the timebase value when the counting was started. This can be converted to time using the time_base_to_time subroutine.

Return Values

0 Operation completed successfully.
Positive error code Refer to the pm_error Subroutine on page 1024 to decode the error code.

Error Codes
Refer to the pm_error Subroutine on page 1024.

Files
/usr/include/pmapi.h Defines standard macros, data types, and subroutines.

Related Information
The pm_delete_program_pgroup Subroutine on page 1021, pm_error Subroutine on page 1024, pm_get_data_pgroup, pm_get_tdata_pgroup and pm_get_Tdata_pgroup Subroutine on page 1039, pm_get_program_pgroup Subroutine on page 1060, pm_initialize Subroutine on page 1071, pm_reset_data_pgroup Subroutine on page 1076, pm_set_program_pgroup Subroutine on page 1092, pm_stop_pgroup and pm_tstop_pgroup Subroutine on page 1114.

pm_start_pthread and pm_tstart_pthread Subroutine

Purpose
Starts Performance Monitor counting for a target pthread.

Library
Performance Monitor APIs Library (libpmapi.a)

Syntax
#include <pmapi.h>

int pm_start_pthread (pid, tid, ptid)
pid_t pid;
tid_t tid;
ptid_t ptid;
int pm_start_pthread (pid, tid, ptid, *time)

pid_t pid;
tid_t tid;
ptid_t ptid;
timebasestruct_t *time

Description
The pm_start_pthread subroutine starts Performance Monitor counting for a target pthread. The pthread must be stopped and must be part of a debuggee process, under the control of the calling process. Counting is effective immediately unless the thread is in a group and the group counting is not currently set to On. The counting state of a thread in a group is obtained by ANDing the thread counting state with the group state.

The pm_tstart_pthread subroutine starts Performance Monitor counting for a target pthread, and returns a timestamp indicating when the counting was started.

If the pthread is running in 1:1 mode, only the tid parameter must be specified. If the pthread is running in m:n mode, only the ptid parameter must be specified. If both the ptid and tid parameters are specified, they must be referring to a single pthread with the ptid parameter specified and currently running on a kernel thread with specified tid parameter.

Parameters

pid Process ID of target pthread. Target process must be a debuggee of the caller process.

tid Thread ID of target pthread. To ignore this parameter, set it to 0.

ptid Pthread ID of the target pthread. To ignore this parameter, set it to 0.

*time Pointer to a structure containing the timebase value when the counting was started. This can be converted to time using the time_base_to_time subroutine.

Return Values

0 Operation completed successfully.

Positive error code Refer to the pm_error Subroutine on page 1024 to decode the error code.

Error Codes
Refer to the pm_error Subroutine on page 1024.

Files

/usr/include/pmapi.h Defines standard macros, data types, and subroutines.

Related Information
The pm_delete_program_pthread Subroutine on page 1022, pm_error Subroutine on page 1024, pm_get_data_pthread, pm_get_tdata_pthread or pm_get_Tdata_pthread Subroutine on page 1043, pm_get_program_pthread Subroutine on page 1063, pm_initialize Subroutine on page 1071, pm_reset_data_pthread Subroutine on page 1078, pm_set_program_pthread Subroutine on page 1095, pm_stop_pthread and pm_tstop_pthread Subroutine on page 1116.
pm_start_thread and pm_tstart_thread Subroutine

Purpose
Starts Performance Monitor counting for a target thread.

Library
Performance Monitor APIs Library (libpmapi.a)

Syntax
#include <pmapi.h>

int pm_start_thread (pid, tid);

pid_t pid;
tid_t tid;

int pm_tstart_thread (pid, tid, *time);

pid_t pid;
tid_t tid;
timebasestruct_t *time

Description
This subroutine supports only the 1:1 threading model. It has been superseded by the \texttt{pm_start_pthread} subroutine, which supports both the 1:1 and the M:N threading models. A call to this subroutine is equivalent to a call to the \texttt{pm_start_pthread} subroutine with a \texttt{ptid} parameter equal to 0.

The \texttt{pm_start_thread} subroutine starts Performance Monitor counting for a target kernel thread. The thread must be stopped and must be part of a debuggee process, under the control of the calling process. Counting is effective immediately unless the thread is in a group and the group counting is not currently set to On. The counting state of a thread in a group is obtained by ANDing the thread counting state with the group state.

The \texttt{pm_tstart_thread} subroutine starts Performance Monitor counting for a target kernel thread, and returns a timestamp indicating when the counting was started.

Parameters

\begin{itemize}
\item \texttt{pid}
\begin{itemize}
\item Process ID of target thread. Target process must be a debuggee of the caller process.
\end{itemize}
\item \texttt{tid}
\begin{itemize}
\item Thread ID of target thread.
\end{itemize}
\item \texttt{*time}
\begin{itemize}
\item Pointer to a structure containing the timebase value when the counting was started. This can be converted to time using the \texttt{time_base_to_time} subroutine.
\end{itemize}
\end{itemize}

Return Values

\begin{itemize}
\item 0
\begin{itemize}
\item Operation completed successfully.
\end{itemize}
\item Positive Error Code
\begin{itemize}
\item Refer to the \texttt{pm_error} subroutine to decode the error code.
\end{itemize}
\end{itemize}
**Error Codes**

Refer to the [pm_error subroutine](#) on page 1024 subroutine.

**Files**

Files

```bash
/usr/include/papi.h
```

Defines standard macros, data types, and subroutines.

**Related Information**

The [pm_init subroutine](#) on page 1069 subroutine, [pm_error subroutine](#) on page 1024 subroutine, [pm_set_program_thread subroutine](#) on page 1098 subroutine, [pm_get_program_thread subroutine](#) on page 1066 subroutine, [pm_delete_program_thread subroutine](#) on page 1023 subroutine, [pm_get_data_thread subroutine](#) on page 1046 subroutine, [pm_stop_thread subroutine](#) on page 1117 subroutine, [pm_reset_data_thread subroutine](#) on page 1079 subroutine.

Performance Monitor API Programming Concepts in [AIX 5L Version 5.3 Performance Tools Guide and Reference](#).

---

**pm_stop and pm_tstop Subroutine**

**Purpose**

Stops system wide Performance Monitor counting.

**Library**

Performance Monitor APIs Library (libpapi.a)

**Syntax**

```c
#include <papi.h>

int pm_stop()

int pm_tstop(*time)

const timebasestruct_t *time;
```

**Description**

The **pm_stop** subroutine stops system wide Performance Monitoring counting.

The **pm_tstop** subroutine stops system wide Performance Monitoring counting, and returns a timestamp indicating when the counting was stopped.

**Parameters**

* **time**
  
  Pointer to a structure containing the timebase value when the counting was stopped. This can be converted to time using the **time_base_to_time** subroutine.

**Return Values**

* **0**
  
  Operation completed successfully.

* **Positive error code**
  
  Refer to the [pm_error subroutine](#) on page 1024 to decode the error code.
Error Codes
Refer to the "pm_error Subroutine" on page 1024.

Files
/usr/include/pmapi.h
Defines standard macros, data types, and subroutines.

Related Information
The "pm_init Subroutine" on page 1069, "pm_error Subroutine" on page 1024, "pm_set_program Subroutine" on page 1080, "pm_get_program Subroutine" on page 1049, "pm_delete_program Subroutine" on page 1017, "pm_get_data, pm_get_tdata, pm_get_Tdata, pm_get_data_cpu, pm_get_tdata_cpu, pm_get_Tdata_cpu, pm_get_data_lcpu, pm_get_tdata_lcpu and pm_get_Tdata_lcpu Subroutine" on page 1025, "pm_start and pm_tstart Subroutine" on page 1101, "pm_reset_data Subroutine" on page 1073.


pm_stop_group and pm_tstop_group Subroutine

Purpose
Stops Performance Monitor counting for the group to which a target thread belongs.

Library
Performance Monitor APIs Library (libpmapi.a)

Syntax
#include <pmapi.h>

int pm_stop_group ( pid, tid )
pid_t pid;
tid_t tid;

int pm_tstop_group ( pid, tid, *time )
pid_t pid;
tid_t tid;
timebasestruct_t *time;

Description
This subroutine supports only the 1:1 threading model. It has been superseded by the pm_stop_pgroup subroutine, which supports both the 1:1 and the M:N threading models. A call to this subroutine is equivalent to a call to the pm_stop_pgroup subroutine with a ptid parameter equal to 0.

The pm_stop_group subroutine stops Performance Monitor counting for a target kernel thread, the counting group to which it belongs, and all the other thread members of the same group. Counting stops immediately for all the threads in the counting group. The target thread must be stopped and must be part of a debuggee process, under control of the calling process.

The pm_tstop_group subroutine stops Performance Monitor counting for a target kernel thread, the counting group to which it belongs, and all the other thread members of the same group, and returns a timestamp indicating when the counting was stopped.
Parameters

pid
Process ID of target thread. Target process must be a debuggee of the caller process.

tid
Thread ID of target thread.

*time
Pointer to a structure containing the timebase value when the counting was stopped. This can be converted to time using the time_base_to_time subroutine.

Return Values

0
Operation completed successfully.

Positive error code
Refer to the "pm_error Subroutine" on page 1024 to decode the error code.

Error Codes

Refer to the "pm_error Subroutine" on page 1024.

Files

/usr/include/pmaii.h
Defines standard macros, data types, and subroutines.

Related Information

The "pm_init Subroutine" on page 1069, "pm_error Subroutine" on page 1024, "pm_set_program_group Subroutine" on page 1081, "pm_get_program_group Subroutine" on page 1051, "pm_delete_program_group Subroutine" on page 1018, "pm_get_data_group, pm_get_tdata_group and pm_get_Tdata_group Subroutine" on page 1028, "Syntax" on page 1102, "pm_reset_data_group Subroutine" on page 1073.


pm_stop_mygroup and pm_tstop_mygroup Subroutine

Purpose

Stops Performance Monitor counting for the group to which the calling thread belongs.

Library

Performance Monitor APIs Library (libpmaii.a)

Syntax

#include <pmaii.h>

int pm_stop_mygroup ()

int pm_tstop_mygroup (timebasestruct_t *time);

description

The pm_stop_mygroup subroutine stops Performance Monitor counting for the group to which the calling kernel thread belongs. This is effective immediately for all the threads in the counting group.
The **pm_tstop_mygroup** subroutine stops Performance Monitor counting for the group to which the calling kernel thread belongs, and returns a timestamp indicating when the counting was stopped.

### Parameters

* `time`  
  Pointer to a structure containing the timebase value when the counting was stopped. This can be converted to time using the `time_base_to_time` subroutine.

### Return Values

<table>
<thead>
<tr>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Operation completed successfully.</td>
</tr>
<tr>
<td>Positive</td>
<td>Error code</td>
</tr>
<tr>
<td>error code</td>
<td>Refer to the <a href="#">&quot;pm_error Subroutine&quot; on page 1024</a> to decode the error code.</td>
</tr>
</tbody>
</table>

### Error Codes

Refer to the ["pm_error Subroutine" on page 1024](#).

### Files

[`/usr/include/pmapi.h`](#)  
Defines standard macros, data types, and subroutines.

### Related Information

The ["pm_init Subroutine" on page 1069](#), ["pm_error Subroutine" on page 1024](#), ["pm_set_program_mygroup Subroutine" on page 1086](#), ["pm_get_program_mygroup Subroutine" on page 1055](#), ["pm_delete_program_mygroup Subroutine" on page 1019](#), ["pm_get_data_mygroup Subroutine" on page 1019](#), ["pm_get_tdata_mygroup or pm_get_Tdata_mygroup Subroutine" on page 1033](#), ["Description" on page 1104](#), ["pm_reset_data_mygroup Subroutine" on page 1075](#).

---

**pm_stop_mythread and pm_tstop_mythread Subroutine**

### Purpose

Stops Performance Monitor counting for the calling thread.

### Library

Performance Monitor APIs Library (**libpmapi.a**)

### Syntax

```c
#include <pmapi.h>

int pm_stop_mythread ()
int pm_tstop_mythread (timebasestruct_t *time);
```

### Description

The **pm_stop_mythread** subroutine stops Performance Monitor counting for the calling kernel thread.
The **pm_tstop_mythread** subroutine stops Performance Monitor counting for the calling kernel thread, and returns a timestamp indicating when the counting was stopped.

### Parameters

* `time` Pointer to a structure containing the timebase value when the counting was stopped. This can be converted to time using the **time_base_to_time** subroutine.

### Return Values

<table>
<thead>
<tr>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Operation completed successfully.</td>
</tr>
<tr>
<td>Positive error code</td>
<td>Refer to the &quot;pm_error Subroutine&quot; on page 1024 to decode the error code.</td>
</tr>
</tbody>
</table>

### Error Codes

Refer to the "pm_error Subroutine" on page 1024.

### Files

`/usr/include/pma.pi.h` Defines standard macros, data types, and subroutines.

### Related Information

The "pm_init Subroutine" on page 1069, "pm_error Subroutine" on page 1024, "pm_set_program_mythread Subroutine" on page 1089, "pm_get_program_mythread Subroutine" on page 1057, "pm_delete_program_mythread Subroutine" on page 1020, "pm_get_data_mythread, pm_get_tdata_mythread or pm_get_Tdata_mythread Subroutine" on page 1036, "pm_start_mythread and pm_tstart_mythread Subroutine" on page 1105, "pm_reset_data_mythread Subroutine" on page 1076.


### pm_stop_pgroup and pm_tstop_pgroup Subroutine

#### Purpose

Stops Performance Monitor counting for the group to which a target pthread belongs.

#### Library

Performance Monitor APIs Library (libpmapi.a)

#### Syntax

```c
#include <pmapi.h>

int pm_stop_pgroup (pid_t pid, tid_t tid, ptid_t ptid);
int pm_tstop_pgroup (pid_t pid, tid_t tid, ptid_t ptid, *time_t time);
```
tid_t tid;
ptid_t ptid;
timebasestruct_t *time;

Description
The **pm_stop_pgroup** subroutine stops Performance Monitor counting for a target pthread, the counting
group to which it belongs, and all the other pthread members of the same group. Counting stops
immediately for all the pthreads in the counting group. The target pthread must be stopped and must be
part of a debuggee process, under control of the calling process.

The **pm_tstop_pgroup** subroutine stops Performance Monitor counting for a target pthread, the counting
group to which it belongs, and all the other pthread members of the same group, and returns a timestamp
indicating when the counting was stopped.

If the pthread is running in 1:1 mode, only the *tid* parameter must be specified. If the pthread is running in
m:n mode, only the *ptid* parameter must be specified. If both the *ptid* and *tid* parameters are specified,
they must be referring to a single pthread with the *ptid* parameter specified and currently running on a
kernel thread with specified *tid* parameter.

Parameters

*pid*  
Process ID of target pthread. Target process must be a
debuggee of the caller process.

*tid*  
Thread ID of target pthread. To ignore this parameter, set
it to 0.

*ptid*  
Pthread ID of the target pthread. To ignore this parameter,
set it to 0.

*time*  
Pointer to a structure containing the timebase value when
the counting was stopped. This can be converted to time
using the **time_base_to_time** subroutine.

Return Values

0  Operation completed successfully.

Positive error code  Refer to the "**pm_error Subroutine** on page 1024" to decode the error code.

Error Codes
Refer to the "**pm_error Subroutine** on page 1024."  

Files

*/usr/include/pmapi.h*  Defines standard macros, data types, and subroutines.

Related Information

The "**pm_delete_program_pgroup Subroutine** on page 1021," "**pm_error Subroutine** on page 1024,
"**pm_get_data_pgroup, pm_get_tdata_pgroup and pm_get_Tdata_pgroup Subroutine** on page 1039,
"**pm_get_program_pgroup Subroutine** on page 1060," "**pm_initialize Subroutine** on page 1071,
"**pm_reset_data_pgroup Subroutine** on page 1076," "**pm_set_program_pgroup Subroutine** on page 1092,
"**pm_start_pgroup and pm_tstart_pgroup Subroutine** on page 1106;"

Performance Monitor API Programming Concepts in AIX 5L Version 5.3 Performance Tools Guide and
Reference.
pm_stop_pthread and pm_tstop_pthread Subroutine

Purpose
Stops Performance Monitor counting for a target pthread.

Library
Performance Monitor APIs Library (libpmapi.a)

Syntax
#include <pmapi.h>

int pm_stop_pthread (pid, tid, ptid)
pid_t pid;
tid_t tid;
ptid_t ptid;

int pm_tstop_pthread (pid, tid, ptid, *time)
pid_t pid;
tid_t tid;
ptid_t ptid;
timebasestruct_t *time;

Description
The pm_stop_pthread subroutine stops Performance Monitor counting for a target pthread. The pthread must be stopped and must be part of a debuggee process, under the control of the calling process.

The pm_tstop_pthread subroutine stops Performance Monitor counting for a target pthread, and returns a timestamp indicating when the counting was stopped.

If the pthread is running in 1:1 mode, only the tid parameter must be specified. If the pthread is running in m:n mode, only the ptid parameter must be specified. If both the ptid and tid parameters are specified, they must be referring to a single pthread with the ptid parameter specified and currently running on a kernel thread with specified tid parameter.

Parameters

pid Process ID of target pthread. Target process must be a debuggee of the caller process.

tid Thread ID of target pthread. To ignore this parameter, set it to 0.

ptid Pthread ID of the target pthread. To ignore this parameter, set it to 0.

*time Pointer to a structure containing the timebase value when the counting was stopped. This can be converted to time using the time_base_to_time subroutine.

Return Values

0 Operation completed successfully.
Positive error code Refer to the "pm_error Subroutine" on page 1024 to decode the error code.

Error Codes
Refer to the "pm_error Subroutine" on page 1024.
Files

/usr/include/pmapi.h

Defines standard macros, data types, and subroutines.

Related Information

The "pm_delete_program_pthread Subroutine" on page 1022, "pm_error Subroutine" on page 1024, "pm_get_data_pthread, pm_get_tdata_pthread or pm_get_Tdata_pthread Subroutine" on page 1043, "pm_get_program_pthread Subroutine" on page 1063, "pm_initialize Subroutine" on page 1071, "pm_reset_data_pthread Subroutine" on page 1078, "pm_set_program_pthread Subroutine" on page 1095, "pm_start_pthread and pm_tstart_pthread Subroutine" on page 1107.


pm_stop_thread and pm_tstop_thread Subroutine

Purpose

Stops Performance Monitor counting for a target thread.

Library

Performance Monitor APIs Library (libpmapi.a)

Syntax

#include <pmapi.h>

int pm_stop_thread (pid, tid)
pid_t pid;
tid_t tid;

int pm_tstop_thread (pid, tid, *time)
pid_t pid;
tid_t tid;
timebasestruct_t *time;

Description

This subroutine supports only the 1:1 threading model. It has been superseded by the pm_stop_pthread subroutine, which supports both the 1:1 and the M:N threading models. A call to this subroutine is equivalent to a call to the pm_stop_pthread subroutine with a ptid parameter equal to 0.

The pm_stop_thread subroutine stops Performance Monitor counting for a target kernel thread. The thread must be stopped and must be part of a debuggee process, under the control of the calling process.

The pm_tstop_thread subroutine stops Performance Monitor counting for a target kernel thread, and returns a timestamp indicating when the counting was stopped.

Parameters

pid Process ID of target thread. Target process must be a debuggee of the caller process.

tid Thread ID of target thread.
**poll Subroutine**

**Purpose**
Checks the I/O status of multiple file descriptors and message queues.

**Library**
Standard C Library (`libc.a`)

**Syntax**
```c
#include <sys/poll.h>
#include <sys/select.h>
#include <sys/types.h>

int poll(ListPointer, Nfdsmsgs, Timeout);
void *ListPointer;
unsigned long Nfdsmsgs;
long Timeout;
```

**Description**
The `poll` subroutine checks the specified file descriptors and message queues to see if they are ready for reading (receiving) or writing (sending), or to see if they have an exceptional condition pending. Even though there are `OPEN_MAX` number of file descriptors available, only `FD_SETSIZE` number of file descriptors are accessible with this subroutine.
Note: The **poll** subroutine applies only to character devices, pipes, message queues, and sockets. Not all character device drivers support it. See the descriptions of individual character devices for information about whether and how specific device drivers support the **poll** and **select** subroutines.

For compatibility with previous releases of this operating system and with BSD systems, the **select** subroutine is also supported.

**Parameters**

**ListPointer**

Specifies a pointer to an array of **pollfd** structures, **pollmsg** structures, or to **apollist** structure. Each structure specifies a file descriptor or message queue ID and the events of interest for this file or message queue. The **pollfd**, **pollmsg**, and **pollist** structures are defined in the `/usr/include/sys/poll.h` file. If a **pollist** structure is to be used, a structure similar to the following should be defined in a user program. The **pollfd** structure must precede the **pollmsg** structure.

```
struct pollist {
  struct pollfd fds[3];
  struct pollmsg msgs[2];
} list;
```

The structure can then be initialized as follows:

```
list.fds[0].fd = file_descriptorA;
list.fds[0].events = requested_events;
list.msgs[0].msgid = message_id;
list.msgs[0].events = requested_events;
```

The rest of the elements in the `fds` and `msgs` arrays can be initialized the same way. The **poll** subroutine can then be called, as follows:

```
nfds = 3; /* number of pollfd structs */
nmsgs = 2; /* number of pollmsg structs */
timeout = 1000 /* number of milliseconds to timeout */
poll(&list, (nmsgs<<16)|(nfds), 1000);
```

The exact number of elements in the `fds` and `msgs` arrays must be used in the calculation of the `Nfdsmsgs` parameter.

**Nfdsmsgs**

Specifies the number of file descriptors and the exact number of message queues to check.

The low-order 16 bits give the number of elements in the array of **pollfd** structures, while the high-order 16 bits give the exact number of elements in the array of **pollmsg** structures. If either half of the `Nfdsmsgs` parameter is equal to a value of 0, the corresponding array is assumed not to be present.

**Timeout**

Specifies the maximum length of time (in milliseconds) to wait for at least one of the specified events to occur. If the **Timeout** parameter value is -1, the **poll** subroutine does not return until at least one of the specified events has occurred. If the value of the **Timeout** parameter is 0, the **poll** subroutine does not wait for an event to occur but returns immediately, even if none of the specified events have occurred.

**poll Subroutine STREAMS Extensions**

In addition to the functions described above, the **poll** subroutine multiplexes input/output over a set of file descriptors that reference open streams. The **poll** subroutine identifies those streams on which you can send or receive messages, or on which certain events occurred.

You can receive messages using the **read** subroutine or the **getmsg** system call. You can send messages using the **write** subroutine or the **putmsg** system call. Certain **streamio** operations, such as **I_RECVFD** and **I_SENDFD** can also be used to send and receive messages. See the **streamio** operations.
The *ListPointer* parameter specifies the file descriptors to be examined and the events of interest for each file descriptor. It points to an array having one element for each open file descriptor of interest. The array’s elements are *pollfd* structures. In addition to the *pollfd* structure in the `/usr/include/sys/poll.h` file, STREAMS supports the following members:

```c
int fd;    /* file descriptor */
short events; /* requested events */
short revents; /* returned events */
```

The *fd* field specifies an open file descriptor and the *events* and *revents* fields are bit-masks constructed by ORing any combination of the following event flags:

- **POLLIN** A nonpriority or file descriptor-passing message is present on the stream-head read queue. This flag is set even if the message is of 0 length. In the *revents* field this flag is mutually exclusive with the *POLLPRI* flag. See the *I_RECVFD* command.
- **POLLRDNORM** A nonpriority message is present on the stream-head read queue.
- **POLLRDBAND** A priority message (band > 0) is present on the stream-head read queue.
- **POLLPRI** A high-priority message is present on the stream-head read queue. This flag is set even if the message is of 0 length. In the *revents* field, this flag is mutually exclusive with the *POLLIN* flag.
- **POLLOUT** The first downstream write queue in the stream is not full. Normal priority messages can be sent at any time. See the *putmsg* system call.
- **POLLWRNORM** The same as *POLLOUT*.
- **POLLWRBAND** A priority band greater than 0 exists downstream and priority messages can be sent at anytime.
- **POLLMMSG** A message containing the *SIGPOLL* signal has reached the front of the stream-head read queue.

**Return Values**

On successful completion, the *poll* subroutine returns a value that indicates the total number of file descriptors and message queues that satisfy the selection criteria. The return value is similar to the *Nfdsmsgs* parameter in that the low-order 16 bits give the number of file descriptors, and the high-order 16 bits give the number of message queue identifiers that had nonzero *revents* values. The *NFDS* and *NMSGS* macros, found in the *sys/select.h* file, can be used to separate these two values from the return value. The *NFDS* macro returns *NFDS#*, where the number returned indicates the number of files reporting some event or error, and the *NMSGS* macro returns *NMSGS#*, where the number returned indicates the number of message queues reporting some event or error.

A value of 0 indicates that the *poll* subroutine timed out and that none of the specified files or message queues indicated the presence of an event (all *revents* fields were values of 0).

If unsuccessful, a value of -1 is returned and the global variable *errno* is set to indicate the error.

**Error Codes**

The *poll* subroutine does not run successfully if one or more of the following are true:

- **EAGAIN** Allocation of internal data structures was unsuccessful.
- **EINVAL** The number of *pollfd* structures specified by the *Nfdsmsgs* parameter is greater than *FD_SETSIZE*. This error is also returned if the number of *pollmsg* structures specified by the *Nfdsmsgs* parameter is greater than the maximum number of allowable message queues.
- **EFAULT** The *ListPointer* parameter in conjunction with the *Nfdsmsgs* parameter addresses a location outside of the allocated address space of the process.
Related Information
The `read` subroutine, `select` subroutine, `write` subroutine.

The `getmsg` system call, `putmsg` system call.

The `streamio` operations.

The STREAMS Overview and the Input and Output Handling Programmer's Overview in AIX 5L Version 5.3 General Programming Concepts: Writing and Debugging Programs.

pollset_create, pollset_ctl, pollset_destroy, pollset_poll, and pollset_query Subroutines

Purpose
Check I/O status of multiple file descriptors.

Library
Standard C Library (libc.a)

Syntax
```c
#include <sys/poll.h>
#include <sys/pollset.h>
#include <sys/fcntl.h>

pollset_t ps = pollset_create(int maxfd)
int rc = pollset_destroy(pollset_t ps)
int rc = pollset_ctl(pollset_t ps, struct poll_ctl *pollctl_array, int array_length)
int rc = pollset_query(pollset_t ps, struct pollfd *pollfd_query)
int nfound = pollset_poll(pollset_t ps, struct pollfd *polldata_array, int array_length, int timeout)
```

Description
The pollset application programming interface (API) efficiently poll a large file descriptor set. This interface is best used when the file descriptor set is not frequently updated. The pollset subroutine can provide a significant performance enhancement over traditional select and poll APIs. Improvements are most visible when the number of events returned per poll operation is small in relation to the number of file descriptors polled.

The pollset API uses system calls to accomplish polling. A file descriptor set (or pollset) is established with a successful call to pollset_create. File descriptors and poll events are added, removed, or updated using the pollset_ctl subroutine. The pollset_poll subroutine is called to perform the poll operation. A pollset_query subroutine is called to query if a file descriptor is contained in the current poll set.

A pollset is established with a successful call to pollset_create. The pollset is initially empty following this system call. Each call to pollset_create creates a new and independent pollset. This can be useful to applications that monitor distinct sets of file descriptors. The maximum number of file descriptors that can belong to the pollset is specified by maxfd. If maxfd has a value of -1, the maximum number of file descriptors that can belong to the pollset is bound by OPEN_MAX as defined in <sys/limits.h> (the AIX limit of open file descriptors per process). AIX imposes a system-wide limit of 245025 active pollsets at one time. Upon failure, this system call returns -1 with errno set appropriately. Upon success, a pollset ID of type pollset_t is returned:
```c
typedef int pollset_t
```
The pollset ID must not be altered by the application. The pollset API verifies that the ID is not -1. In addition, the process ID of the application must match the process ID stored at pollset creation time.

A pollset is destroyed with a successful call to `pollset_destroy`. Upon success, this system call returns 0. Upon failure, the `pollset_destroy` subroutine returns -1 with `errno` set to the appropriate code. An `errno` of `EINVAL` indicates an invalid pollset ID.

File descriptors must be added to the pollset with the `pollset_ctl` subroutine before they can be monitored. An array of `poll_ctl` structures is passed to `pollset_ctl` through `pollctl_array`:

```c
struct poll_ctl {
    short cmd;
    short events;
    int fd;
}
```

Each `poll_ctl` structure contains an `fd`, `events`, and `cmd` field. The `fd` field defines the file descriptor to operate on. The `events` field contains events of interest. When `cmd` is `PS_ADD`, the `pollset_ctl` call adds a valid open file descriptor to the pollset. If a file descriptor is already in the pollset, `PS_ADD` causes `pollset_ctl` to return an error. When `cmd` is `PS_MOD` and the file descriptor is already in the pollset, bits in the `events` field are added (ORed) to the monitored events. If the file descriptor is not already in the pollset, `PS_MOD` adds a valid open file descriptor to the pollset.

Although poll events can be added by specifying an existing file descriptor, the file descriptor must be removed and then added again to remove an event. When `cmd` is `PS_DELETE` and the file descriptor is already in the pollset, `pollset_ctl` removes the file descriptor from the pollset. If the file descriptor is not already in the pollset, then `PS_DELETE` causes `pollset_ctl` to return an error.

The `pollset_query` interface can be used to determine information about a file descriptor with respect to the pollset. If the file descriptor is in the pollset, `pollset_query` returns 1 and `events` is set to the currently monitored events.

The `pollset_poll` subroutine determines which file descriptors in the pollset that have events pending. The `polldata_array` parameter contains a buffer address where `pollfd` structures are returned for file descriptors that have pending events. The number of events returned by a poll is limited by `array_length`. The `timeout` parameter specifies the amount of time to wait if no events are pending. Setting `timeout` to 0 guarantees that the `pollset_poll` subroutine returns immediately. Setting `timeout` to -1 specifies an infinite timeout. Other nonzero positive values specify the time to wait in milliseconds.

When events are returned from a `pollset_poll` operation, each `pollfd` structure contains an `fd` member with the file descriptor set, an `events` member with the requested events, and an `revents` member with the events that have occurred.

A single pollset can be accessed by multiple threads in a multithreaded process. When multiple threads are polling one pollset and an event occurs for a file descriptor, only one thread can be prompted to receive the event. After a file descriptor is returned to a thread, new events will not be generated until the next `pollset_poll` call. This behavior prevents all threads from being prompted on each event. Multiple threads can perform `pollset_poll` operations at one time, but modifications to the pollset require exclusive access. A thread that tries to modify the pollset is blocked until all threads currently in poll operations have exited `pollset_poll`. In addition, a thread calling `pollset_destroy` is blocked until all threads have left the other system calls (`pollset_ctl`, `pollset_query`, and `pollset_poll`).

A process can call `fork` after calling `pollset_create`. The child process will already have a pollset ID per pollset, but `pollset_destroy`, `pollset_ctl`, `pollset_query`, and `pollset_poll` operations will fail with an `errno` value of `EACCES`. 
After a file descriptor is added to a pollset, the file descriptor will not be removed until a pollset_ctl call with the cmd of PS_DELETE is executed. The file descriptor remains in the pollset even if the file descriptor is closed. A pollset_poll operation on a pollset containing a closed file descriptor returns a POLLNVAL event for that file descriptor. If the file descriptor is later allocated to a new object, the new object will be polled on future pollset_poll calls.

Parameters

array_length
Specifies the length of the array parameters.

maxfd
Specifies the maximum number of file descriptors that can belong to the pollset.

pollctl_array
The pointer to an array of poll_ctl structures that describes the file descriptors (through the polfd structure) and the unique operation to perform on each file descriptor (add, remove, or modify).

polldata_array
Returns the requested events that have occurred on the pollset.

pollfd_query
Points to a file descriptor that might or might not belong to the pollset. If it belongs to the pollset, then the requested events field of this parameter is updated to reflect what is currently being monitored for this file descriptor.

ps
Specifies the pollset ID.

timeout
Specifies the amount of time in milliseconds to wait for any monitored events to occur. A value of -1 blocks until some monitored event occurs.

Return Values

Upon success, the pollset_destroy returns 0. Upon failure, the pollset_destroy subroutine returns -1 with errno set to the appropriate code.

Upon success, the pollset_create subroutine returns a pollset ID of type pollset_t. Upon failure, this system call returns -1 with errno set appropriately.

Upon success, pollset_ctl returns 0. Upon failure, pollset_ctl returns the 0-based problem element number of the pollctl_array (for example, 2 is returned for element 3). If the first element is the problem element, or some other error occurs prior to processing the array of elements, -1 is returned and errno is set to the appropriate code. The calling application must acknowledge that elements in the array prior to the problem element were successfully processed and should attempt to call pollset_ctl again with the elements of pollctl_array beyond the problematic element.

If a file descriptor is not a member of the pollset, pollset_query returns 0. If the file descriptor is in the pollset, pollset_query returns 1 and events is set to the currently monitored events. If an error occurs after there is an attempt to determine if the file descriptor is a member of the pollset, then pollset_query returns -1 with errno set to the appropriate return code.

The pollset_poll subroutine returns the number of file descriptors on which requested events occurred. When no requested events occurred on any of the file descriptors, 0 is returned. A value of -1 is returned when an error occurs and errno is set to the appropriate code.

Error Codes

EACCES Process does not have permission to access a pollset.
EAGAIN System resource temporarily not available.
EFAULT Address supplied was not valid.
EINTR A signal was received during the system call.
EINVAL Invalid parameter.
ENOMEM Insufficient system memory available.
ENOSPC Maximum number of pollsets in use.
EPERM Process does not have permission to create a pollset.
popen Subroutine

Purpose
Initiates a pipe to a process.

Library
Standard C Library (libc.a)

Syntax
```c
#include <stdio.h>

FILE *popen (const char *Command, *Type);
```

Description
The `popen` subroutine creates a pipe between the calling program and a shell command to be executed.

Note: The `popen` subroutine runs only **sh** shell commands. The results are unpredictable if the `Command` parameter is not a valid **sh** shell command. If the terminal is in a trusted state, the **tsh** shell commands are run.

If streams opened by previous calls to the `popen` subroutine remain open in the parent process, the `popen` subroutine closes them in the child process.

The `popen` subroutine returns a pointer to a `FILE` structure for the stream.

**Attention:** If the original processes and the process started with the `popen` subroutine concurrently read or write a common file, neither should use buffered I/O. If they do, the results are unpredictable.

Some problems with an output filter can be prevented by flushing the buffer with the `fflush` subroutine.

Parameters
- `Command` Points to a null-terminated string containing a shell command line.
- `Type` Points to a null-terminated string containing an I/O mode. If the `Type` parameter is the value `r`, you can read from the standard output of the command by reading from the file `Stream`. If the `Type` parameter is the value `w`, you can write to the standard input of the command by writing to the file `Stream`.

Because open files are shared, a type `r` command can be used as an input filter and a type `w` command as an output filter.

Return Values
The `popen` subroutine returns a null pointer if files or processes cannot be created, or if the shell cannot be accessed.

Error Codes
The `popen` subroutine may set the `EINVAL` variable if the `Type` parameter is not valid. The `popen` subroutine may also set `errno` global variables as described by the `fork` or `pipe` subroutines.
posix_fadvise Subroutine

Purpose
Provides advisory information to the system regarding future behavior of the application with respect to a given file.

Syntax
```
#include <fcntl.h>
int posix_fadvise(int fd, off_t offset, size_t len, int advice);
```

Description
This function advises the system on the expected future behavior of the application with regards to a given file. The system can take this advice into account when performing operations on file data specified by this function. The advice is given over the range covered by the `offset` parameter and continuing for the number of bytes specified by the `len` parameter. If the value of the `len` parameter is 0, then all data following the `offset` parameter is covered.

The `advice` parameter must be one of the following:
- POSIX_FADV_NORMAL
- POSIX_FADV_SEQUENTIAL
- POSIX_FADV_RANDOM
- POSIX_FADV_WILLNEED
- POSIX_FADV_DONTNEED
- POSIX_FADV_NOREUSE

Parameters
- `fd` File descriptor of the file to be advised
- `offset` Represents the beginning of the address range
- `len` Determines the length of the address range
- `advice` Defines the advice to be given

Return Values
Upon successful completion, the `posix_fadvise` subroutine returns 0. Otherwise, one of the following error codes will be returned.

Error Codes

<table>
<thead>
<tr>
<th>Error Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EBADF</td>
<td>The <code>fd</code> parameter is not a valid file descriptor</td>
</tr>
</tbody>
</table>
posix_fallocate Subroutine

Purpose
Reserve storage space for a given file descriptor.

Syntax
#include <fcntl.h>
int posix_fallocate (int fd, off_t offset, size_t len);

Description
This function reserves adequate space on the file system for the file data range beginning at the value specified by the offset parameter and continuing for the number of bytes specified by the len parameter. Upon successful return, subsequent writes to this file data range will not fail due to lack of free space on the file system media. Space allocated by the posix_fallocate subroutine can be freed by a successful call to the creat subroutine or open subroutine, or by the ftruncate subroutine, which truncates the file size to a size smaller than the sum of the offset parameter and the len parameter.

Parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>fd</td>
<td>File descriptor of the file to reserve</td>
</tr>
<tr>
<td>offset</td>
<td>Represents the beginning of the address range</td>
</tr>
<tr>
<td>len</td>
<td>Determines the length of the address range</td>
</tr>
</tbody>
</table>

Return Values
Upon successful completion, the posix_fallocate subroutine returns 0. Otherwise, one of the following error codes will be returned.

Error Codes

<table>
<thead>
<tr>
<th>Error Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EBADF</td>
<td>The fd parameter is not a valid file descriptor</td>
</tr>
<tr>
<td>EBADF</td>
<td>The fd parameter references a file that was opened without write permission.</td>
</tr>
<tr>
<td>EFBIG</td>
<td>The value of the offset parameter plus the len parameter is greater than the maximum file size</td>
</tr>
<tr>
<td>EINTR</td>
<td>A signal was caught during execution</td>
</tr>
<tr>
<td>EIO</td>
<td>An I/O error occurred while reading from or writing to a file system</td>
</tr>
<tr>
<td>ENODEV</td>
<td>The fd parameter does not refer to a regular file.</td>
</tr>
<tr>
<td>EINVAL</td>
<td>The value of the advice parameter is invalid</td>
</tr>
<tr>
<td>ENOSPC</td>
<td>There is insufficient free space remaining on the file system storage media</td>
</tr>
<tr>
<td>ESPIPE</td>
<td>The fd parameter is associated with a pipe of FIFO</td>
</tr>
</tbody>
</table>
posix_madvise Subroutine

Purpose
Provides advisory information to the system regarding future behavior of the application with respect to a given range of memory.

Syntax
```
#include <sys/mman.h>
int posix_madvise (void *addr, size_t len, int advice);
```

Description
This function advises the system on the expected future behavior of the application with regard to a given range of memory. The system can take this advice into account when performing operations on the data in memory specified by this function. The advice is given over the range covered by the `offset` parameter and continuing for the number of bytes specified by the `addr` parameter and continuing for the number of bytes specified by the `len` parameter.

The `advice` parameter must be one of the following:
- `POSIX_MADV_NORMAL`
- `POSIX_MADV_SEQUENTIAL`
- `POSIX_MADV_RANDOM`
- `POSIX_MADV_WILLNEED`
- `POSIX_MADV_DONTNEED`

Parameters
- `addr` Defines the beginning of the range of memory to be advised
- `len` Determines the length of the address range
- `advice` Defines the advice to be given

Return Values
Upon successful completion, the `posix_fadvise` subroutine returns 0. Otherwise, one of the following error codes will be returned.

Error Codes

<table>
<thead>
<tr>
<th>Error Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EINVAL</td>
<td>The value of the <code>advice</code> parameter is invalid</td>
</tr>
<tr>
<td>ENOMEM</td>
<td>Addresses in the range specified by the <code>addr</code> parameter and the <code>len</code> parameter are partially or completely outside the range of the process’s address space.</td>
</tr>
</tbody>
</table>
posix_openpt Subroutine

Purpose
Opens a pseudo-terminal device.

Library
Standard C library (libc.a)

Syntax
```
#include <stdlib.h>
#include <fcntl.h>

int posix_openpt (oflag)
    int oflag;
```

Description
The `posix_openpt` subroutine establishes a connection between a master device for a pseudo terminal and a file descriptor. The file descriptor is used by other I/O functions that refer to that pseudo terminal.

The file status flags and file access modes of the open file description are set according to the value of the `oflag` parameter.

Parameters
`oflag` Values for the `oflag` parameter are constructed by a bitwise-inclusive OR of flags from the following list, defined in the `<fcntl.h>` file:

- **O_RDWR**
  Open for reading and writing.

- **O_NOCTTY**
  If set, the `posix_openpt` subroutine does not cause the terminal device to become the controlling terminal for the process.

The behavior of other values for the `oflag` parameter is unspecified.

Return Values
Upon successful completion, the `posix_openpt` subroutine opens a master pseudo-terminal device and returns a non-negative integer representing the lowest numbered unused file descriptor. Otherwise, -1 is returned and the `errno` global variable is set to indicate the error.

Error Codes
The `posix_openpt` subroutine will fail if:

- **EMFILE** OPEN_MAX file descriptors are currently open in the calling process.
- **ENFILE** The maximum allowable number of files is currently open in the system.

The **posix_openpt** subroutine may fail if:

<table>
<thead>
<tr>
<th>Error Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EINVAL</td>
<td>The value of the <em>oflag</em> parameter is not valid.</td>
</tr>
<tr>
<td>EAGAIN</td>
<td>Out of pseudo-terminal resources.</td>
</tr>
<tr>
<td>ENOSR</td>
<td>Out of STREAMS resources.</td>
</tr>
</tbody>
</table>

**Examples**
The following example describes how to open a pseudo-terminal and return the name of the slave device and file descriptor

```c
#include <fcntl.h>
#include <stdio.h>

int masterfd, slavefd;
char *slavedevice;

masterfd = posix_openpt(O_RDWR|O_NOCTTY);
if (masterfd == -1 ||
   grantpt (masterfd) == -1 ||
   unlockpt (masterfd) == -1 ||
   (slavedevice = ptsname (masterfd)) == NULL)
   return -1;

printf("slave device is: \%s\n", slavedevice);

slavefd = open(slave, O_RDWR|O_NOCTTY);
if (slavefd < 0)
   return -1;
```

**Related Information**
- “grantpt Subroutine” on page 480
- “open, openx, open64, creat, or creat64 Subroutine” on page 925
- “ptsname Subroutine” on page 1299
- `<fcntl.h>` file in AIX 5L Version 5.3 Files Reference.

**posix_spawn or posix_spawnp Subroutine**

**Purpose**
Spawns a process.

**Syntax**

```c
int posix_spawn(pid_t *restrict pid, const char *restrict path,
                const posix_spawn_file_actions_t *file_actions,
                const posix_spawnattr_t *restrict attrp,
                char *const argv[restrict], char *const envp[restrict]);
int posix_spawnp(pid_t *restrict pid, const char *restrict path,
                 const posix_spawn_file_actions_t *file_actions,
                 const posix_spawnattr_t *restrict attrp,
                 char *const argv[restrict], char * const envp[restrict]);
```
Description
The `posix_spawn` and `posix_spawnp` subroutines create a new process (child process) from the specified process image. The new process image is constructed from a regular executable file called the new process image file.

When a C program is executed as the result of this call, the program is entered as a C-language function call as follows:
```c
int main(int argc, char *argv[]);
```
where `argc` is the argument count and `argv` is an array of character pointers to the arguments themselves. In addition, the following variable:
```c
extern char **environ;
```
is initialized as a pointer to an array of character pointers to the environment strings.

The `argv` parameter is an array of character pointers to null-terminated strings. The last member of this array is a null pointer and is not counted in `argc`. These strings constitute the argument list available to the new process image. The value in `argv[0]` should point to a file name that is associated with the process image being started by the `posix_spawn` or `posix_spawnp` function.

The argument `envp` is an array of character pointers to null-terminated strings. These strings constitute the environment for the new process image. The environment array is terminated by a null pointer.

The number of bytes available for the child process’ combined argument and environment lists is `{ARG_MAX}`. The implementation specifies in the system documentation whether any list overhead, such as length words, null terminators, pointers, or alignment bytes, is included in this total.

The path argument to `posix_spawn` is a path name that identifies the new process image file to execute.

The file parameter to `posix_spawnp` is used to construct a path name that identifies the new process image file. If the file parameter contains a slash character (`/`), the file parameter is used as the path name for the new process image file. Otherwise, the path prefix for this file is obtained by a search of the directories passed as the environment variable `PATH`. If this environment variable is not defined, the results of the search are implementation-defined.

If `file_actions` is a null pointer, file descriptors that are open in the calling process remain open in the child process, except for those whose `FD_CLOEXEC` flag is set (see "fcntl, dup, or dup2 Subroutine" on page 254). For those file descriptors that remain open, all attributes of the corresponding open file descriptions, including file locks, remain unchanged.

If `file_actions` is not a null pointer, the file descriptors open in the child process are those open in the calling process as modified by the spawn file actions object pointed to by `file_actions` and the `FD_CLOEXEC` flag of each remaining open file descriptor after the spawn file actions have been processed. The effective order of processing the spawn file actions is as follows:
1. The set of open file descriptors for the child process is initially the same set as is open for the calling process. All attributes of the corresponding open file descriptions, including file locks (see "fcntl, dup, or dup2 Subroutine" on page 254), remain unchanged.
2. The signal mask, signal default actions, and the effective user and group IDs for the child process are changed as specified in the attributes object referenced by `attrp`.
3. The file actions specified by the spawn file actions object are performed in the order in which they were added to the spawn file actions object.
4. Any file descriptor that has its `FD_CLOEXEC` flag set is closed.

The `posix_spawnattr_t` spawn attributes object type is defined in the `spawn.h` header file. Its attributes are defined as follows:
• If the **POSIX_SPAWN_SETPGROUP** flag is set in the **spawn-flags** attribute of the object referenced by \textit{attrp}, and the **spawn-pgroup** attribute of the same object is non-zero, the child’s process group is as specified in the **spawn-pgroup** attribute of the object referenced by \textit{attrp}.

• As a special case, if the **POSIX_SPAWN_SETPGROUP** flag is set in the **spawn-flags** attribute of the object referenced by \textit{attrp}, and the **spawn-pgroup** attribute of the same object is set to 0, then the child is in a new process group with a process group ID equal to its process ID.

• If the **POSIX_SPAWN_SETPGROUP** flag is not set in the **spawn-flags** attribute of the object referenced by \textit{attrp}, the new child process inherits the parent’s process group.

• If the **POSIX_SPAWN_SETSCHEDPARAM** flag is set in the **spawn-flags** attribute of the object referenced by \textit{attrp}, but **POSIX_SPAWN_SETSCHEDULER** is not set, the new process image initially has the scheduling policy of the calling process with the scheduling parameters specified in the **spawn-schedparam** attribute of the object referenced by \textit{attrp}.

• If the **POSIX_SPAWN_SETSCHEDULER** flag is set in the **spawn-flags** attribute of the object referenced by \textit{attrp} (regardless of the setting of the **POSIX_SPAWN_SETSCHEDPARAM** flag), the new process image initially has the scheduling policy specified in the **spawn-schedpolicy** attribute of the object referenced by \textit{attrp} and the scheduling parameters specified in the **spawn-schedparam** attribute of the same object.

• The **POSIX_SPAWN_RESETIDS** flag in the **spawn-flags** attribute of the object referenced by \textit{attrp} governs the effective user ID of the child process. If this flag is not set, the child process inherits the parent process’ effective user ID. If this flag is set, the child process’ effective user ID is reset to the parent’s real user ID. In either case, if the set-user-ID mode bit of the new process image file is set, the effective user ID of the child process becomes that file’s owner ID before the new process image begins execution.

• The **POSIX_SPAWN_RESETIDS** flag in the **spawn-flags** attribute of the object referenced by \textit{attrp} also governs the effective group ID of the child process. If this flag is not set, the child process inherits the parent process’ effective group ID. If this flag is set, the child process’ effective group ID is reset to the parent’s real group ID. In either case, if the set-group-ID mode bit of the new process image file is set, the effective group ID of the child process becomes that file’s group ID before the new process image begins execution.

• If the **POSIX_SPAWN_SETSIGMASK** flag is set in the **spawn-flags** attribute of the object referenced by \textit{attrp}, the child process initially has the signal mask specified in the **spawn-sigmask** attribute of the object referenced by \textit{attrp}.

• If the **POSIX_SPAWN_SETSIGDEF** flag is set in the **spawn-flags** attribute of the object referenced by \textit{attrp}, the signals specified in the **spawn-sigdefault** attribute of the same object is set to their default actions in the child process. Signals set to the default action in the parent process are set to the default action in the child process. Signals set to be caught by the calling process are set to the default action in the child process.

• Except for **SIGCHLD**, signals set to be ignored by the calling process image are set to be ignored by the child process, unless otherwise specified by the **POSIX_SPAWN_SETSIGDEF** flag being set in the **spawn-flags** attribute of the object referenced by \textit{attrp} and the signals being indicated in the **spawn-sigdefault** attribute of the object referenced by \textit{attrp}.

• If the **SIGCHLD** signal is set to be ignored by the calling process, it is unspecified whether the **SIGCHLD** signal is set to be ignored or set to the default action in the child process. This is true unless otherwise specified by the **POSIX_SPAWN_SETSIGDEF** flag being set in the **spawn-flags** attribute of the object referenced by \textit{attrp} and the **SIGCHLD** signal being indicated in the **spawn-sigdefault** attribute of the object referenced by \textit{attrp}.

• If the value of the \textit{attrp} pointer is NULL, then the default values are used.

All process attributes, other than those influenced by the attributes set in the object referenced by \textit{attrp} in the preceding list or by the file descriptor manipulations specified in **file_actions**, are displayed in the new process image as though \textit{fork} had been called to create a child process and then a member of the \textit{exec} family of functions had been called by the child process to execute the new process image.
By default, fork handlers are not run in `posix_spawn` or `posix_spawnp` routines. To enable fork handlers, set the `POSIX_SPAWN_FORK_HANDLERS` flag in the attribute.

**Return Values**

Upon successful completion, `posix_spawn` and `posix_spawnp` return the process ID of the child process to the parent process, in the variable pointed to by a non-NULL `pid` argument, and return 0 as the function return value. Otherwise, no child process is created, the value stored into the variable pointed to by a non-NULL `pid` is unspecified, and an error number is returned as the function return value to indicate the error. If the `pid` argument is a null pointer, the process ID of the child is not returned to the caller.

**Error Codes**

The `posix_spawn` and `posix_spawnp` subroutines will fail if the following is true:

EINVAL  
The value specified by `file_actions` or `attrp` is invalid.

The error codes for the `posix_spawn` and `posix_spawnp` subroutines are affected by the following conditions:

- If this error occurs after the calling process successfully returns from the `posix_spawn` or `posix_spawnp` function, the child process might exit with exit status 127.
- If `posix_spawn` or `posix_spawnp` fail for any of the reasons that would cause `fork` or one of the `exec` family of functions to fail, an error value is returned as described by `fork` and `exec`, respectively (or, if the error occurs after the calling process successfully returns, the child process exits with exit status 127).
- If `POSIX_SPAWN_SETPGROUP` is set in the `spawn-flags` attribute of the object referenced by `attrp`, and `posix_spawn` or `posix_spawnp` fails while changing the child’s process group, an error value is returned as described by `setpgid` (or, if the error occurs after the calling process successfully returns, the child process exits with exit status 127).
- If `POSIX_SPAWN_SETSCHEDPARAM` is set and `POSIX_SPAWN_SETSCHEDULER` is not set in the `spawn-flags` attribute of the object referenced by `attrp`, then if `posix_spawn` or `posix_spawnp` fails for any of the reasons that would cause `sched_setparam` to fail, an error value is returned as described by `sched_setparam` (or, if the error occurs after the calling process successfully returns, the child process exits with exit status 127).
- If `POSIX_SPAWN_SETSCHEDULER` is set in the `spawn-flags` attribute of the object referenced by `attrp`, and if `posix_spawn` or `posix_spawnp` fails for any of the reasons that would cause `sched_setscheduler` to fail, an error value is returned as described by `sched_setscheduler` (or, if the error occurs after the calling process successfully returns, the child process exits with exit status 127).
- If the `file_actions` argument is not NULL and specifies any `close`, `dup2`, or `open` actions to be performed, and if `posix_spawn` or `posix_spawnp` fails for any of the reasons that would cause `close`, `dup2`, or `open` to fail, an error value is returned as described by `close`, `dup2`, and `open`, respectively (or, if the error occurs after the calling process successfully returns, the child process exits with exit status 127). An open file action might, by itself, result in any of the errors described by `close` or `dup2`, in addition to those described by `open`.

**Related Information**

The “getinterval, incinterval, absinterval, resinc, resabs, alarm, ualarm, getitimer or setitimer Subroutine” on page 382, “chmod or fchmod Subroutine” on page 148, “close Subroutine” on page 175, “fgetl, dup, or dup2 Subroutine” on page 254, “exec, execl, execl, execvp, execve, execvp, or execl Subroutine” on page 235, “exit, execl, execlp, execve, execv, or execlp Subroutine” on page 242, “fork, f_fork, or vfork Subroutine” on page 287, “kill or killpg Subroutine” on page 575, “open, openx, open64, creat, or creat64 Subroutine” on page 925, “posix spawn file actions addclose or posix spawn file actions addopen Subroutine” on page 1133, “posix spawn file actions adddup2 Subroutine” on page 1134, “posix spawn file actions destroy or posix spawn file actions init Subroutine” on page 1135.
posix_spawn_file_actions_addclose or posix_spawn_file_actions_addopen Subroutine

Purpose

Adds close or open action to the spawn file actions object.

Syntax

```c
#include <spawn.h>

int posix_spawn_file_actions_addclose(posix_spawn_file_actions_t *file_actions, int fildes);
int posix_spawn_file_actions_addopen(posix_spawn_file_actions_t *restrict file_actions, int fildes,
const char *restrict path, int oflag, mode_t mode);
```

Description

The `posix_spawn_file_actions_addclose` and `posix_spawn_file_actions_addopen` subroutines close or open action to a spawn file actions object.

A spawn file actions object is of type `posix_spawn_file_actions_t` (defined in the `spawn.h` header file) and is used to specify a series of actions to be performed by a `posix_spawn` or `posix_spawnp` operation in order to arrive at the set of open file descriptors for the child process given the set of open file descriptors of the parent. Comparison or assignment operators for the type `posix_spawn_file_actions_t` are not defined.

A spawn file actions object, when passed to `posix_spawn` or `posix_spawnp`, specifies how the set of open file descriptors in the calling process is transformed into a set of potentially open file descriptors for the spawned process. This transformation is as if the specified sequence of actions was performed exactly once, in the context of the spawned process (prior to running the new process image), in the order in which the actions were added to the object. Additionally, when the new process image is run, any file descriptor (from this new set) that has its FD_CLOEXEC flag set is closed (see "posix_spawn or posix_spawnp Subroutine" on page 1129).

The `posix_spawn_file_actions_addclose` function adds a close action to the object referenced by `file_actions` that causes the file descriptor `fildes` to be closed (as if `close( fildes )` had been called) when a new process is spawned using this file actions object.

The `posix_spawn_file_actions_addopen` function adds an open action to the object referenced by `file_actions` that causes the file named by `path` to be opened, as if `open( path, oflag, mode )` had been called, and the returned file descriptor, if not `fildes`, had been changed to `fildes` when a new process is spawned using this file actions object. If `fildes` was already an open file descriptor, it closes before the new file is opened.

The string described by `path` is copied by the `posix_spawn_file_actions_addopen` function.
Return Values
Upon successful completion, the `posix_spawn_file_actions_addclose` and `posix_spawn_file_actions_addopen` subroutines return 0; otherwise, an error number is returned to indicate the error.

Error Codes
The `posix_spawn_file_actions_addclose` and `posix_spawn_file_actions_addopen` subroutines fail if the following is true:

EBADF The value specified by `fildes` is negative, or greater than or equal to (OPEN_MAX).

The `posix_spawn_file_actions_addclose` and `posix_spawn_file_actions_addopen` subroutines might fail if the following are true:

EINVAL The value specified by `file_actions` is invalid.
ENOMEM Insufficient memory exists to add to the spawn file actions object.

It is not an error for the `fildes` argument passed to these functions to specify a file descriptor for which the specified operation could not be performed at the time of the call. Any such error will be detected when the associated file actions object is used later during a `posix_spawn` or `posix_spawnp` operation.

Related Information
The "close Subroutine" on page 175, "fcntl, dup, or dup2 Subroutine" on page 254, "open, openx, open64, creat, or creat64 Subroutine" on page 925, "posix_spawn or posix_spawnp Subroutine" on page 1129, "posix_spawn_file_actions_adddup2 Subroutine," "posix_spawn_file_actions_destroy or posix_spawn_file_actions_init Subroutine" on page 1135.

posix_spawn_file_actions_adddup2 Subroutine
Purpose
Adds `dup2` action to the spawn file actions object.

Syntax
```c
#include <spawn.h>

int posix_spawn_file_actions_adddup2(posix_spawn_file_actions_t * file_actions, int fildes, int newfildes);
```

Description
The `posix_spawn_file_actions_adddup2` subroutine adds a `dup2` action to the object referenced by `file_actions` that causes the file descriptor `fildes` to be duplicated as `newfildes` when a new process is spawned using this file actions object. This functions as if `dup2(fildes, newfildes)` had been called.

A spawn file actions object is as defined in `posix_spawn_file_actions_addclose`.

Return Values
Upon successful completion, the `posix_spawn_file_actions_adddup2` subroutine returns 0; otherwise, an error number is returned to indicate the error.
Error Codes
The `posix_spawn_file_actions_adddup2` subroutine will fail if the following are true:

- **EBADF** The value specified by `fildes` or `newfildes` is negative, or greater than or equal to `{OPEN_MAX}`.
- **ENOMEM** Insufficient memory exists to add to the spawn file actions object.

The `posix_spawn_file_actions_adddup2` subroutine might fail if the following is true:

- **EINVAL** The value specified by `file_actions` is invalid.

It is not an error for the `fildes` argument passed to this subroutine to specify a file descriptor for which the specified operation could not be performed at the time of the call. Any such error will be detected when the associated file actions object is used later during a `posix_spawn` or `posix_spawnp` operation.

Related Information
The "fcntl, dup, or dup2 Subroutine" on page 254, "posix_spawn or posix_spawnp Subroutine" on page 1129, "posix_spawn_file_actions_addclose or posix_spawn_file_actions_addopen Subroutine" on page 1133, "posix_spawn_file_actions_destroy or posix_spawn_file_actions_init Subroutine."

`posix_spawn_file_actions_destroy` or `posix_spawn_file_actions_init` Subroutine

**Purpose**
Destroys and initializes a spawn file actions object.

**Syntax**
```c
#include <spawn.h>

int posix_spawn_file_actions_destroy(posix_spawn_file_actions_t * file_actions);
int posix_spawn_file_actions_init(posix_spawn_file_actions_t * file_actions);
```

**Description**
The `posix_spawn_file_actions_destroy` subroutine destroys the object referenced by `file_actions`; the object becomes, in effect, uninitialized. An implementation can cause `posix_spawn_file_actions_destroy` to set the object referenced by `file_actions` to an invalid value. A destroyed spawn file actions object can be reinitialized using `posix_spawn_file_actions_init`; the results of otherwise referencing the object after it has been destroyed are undefined.

The `posix_spawn_file_actions_init` function initializes the object referenced by `file_actions` to contain no file actions for `posix_spawn` or `posix_spawnp` to perform.

A spawn file actions object is as defined in `posix_spawn_file_actions_addclose`. The effect of initializing a previously initialized spawn file actions object is undefined.

**Return Values**
Upon successful completion, the `posix_spawn_file_actions_destroy` and `posix_spawn_file_actions_init` subroutines return 0; otherwise, an error number is returned to indicate the error.
Error Codes

The `posix_spawn_file_actions_init` subroutine will fail if the following is true:

- **ENOMEM**: Insufficient memory exists to initialize the spawn file actions object.

The `posix_spawn_file_actions_destroy` subroutine might fail if the following is true:

- **EINVAL**: The value specified by `file_actions` is invalid.

Related Information

The [“posix_spawn or posix_spawnp Subroutine” on page 1129](#).

### posix_spawnattr_destroy or posix_spawnattr_init Subroutine

**Purpose**

Destroys and initializes a spawn attributes object.

**Syntax**

```c
#include <spawn.h>

int posix_spawnattr_destroy(posix_spawnattr_t *attr);
int posix_spawnattr_init(posix_spawnattr_t *attr);
```

**Description**

The `posix_spawnattr_destroy` subroutine destroys a spawn attributes object. A destroyed `attr` attributes object can be reinitialized using `posix_spawnattr_init`; the results of otherwise referencing the object after it has been destroyed are undefined. An implementation can cause `posix_spawnattr_destroy` to set the object referenced by `attr` to an invalid value.

The `posix_spawnattr_init` subroutine initializes a spawn attributes object `attr` with the default value for all of the individual attributes used by the implementation. Results are undefined if `posix_spawnattr_init` is called specifying an `attr` attributes object that is already initialized.

A spawn attributes object is of type `posix_spawnattr_t` (defined in the `spawn.h` header file) and is used to specify the inheritance of process attributes across a spawn operation. Comparison or assignment operators for the type `posix_spawnattr_t` are not defined.

Each implementation documents the individual attributes it uses and their default values unless these values are defined by IEEE Std 1003.1-2001. Attributes not defined by IEEE Std 1003.1-2001, their default values, and the names of the associated functions to get and set those attribute values are implementation-defined.

The resulting spawn attributes object (possibly modified by setting individual attribute values), is used to modify the behavior of `posix_spawn` or `posix_spawnp`. After a spawn attributes object has been used to spawn a process by a call to a `posix_spawn` or `posix_spawnp`, any function affecting the attributes object (including destruction) will not affect any process that has been spawned in this way.

**Return Values**

Upon successful completion, the `posix_spawnattr_destroy` and `posix_spawnattr_init` subroutines return 0; otherwise, an error number is returned to indicate the error.
Error Codes
The `posix_spawnattr_destroy` subroutine might fail if the following is true:

EINVAL  The value specified by `attr` is invalid.

Related Information
The [posix_spawn or posix_spawnp Subroutine](#) on page 1129, [posix_spawnattr_getsigdefault or posix_spawnattr_setsigdefault Subroutine](#) on page 1141, [posix_spawnattr_getflags or posix_spawnattr_setflags Subroutine](#) on page 1138, [posix_spawnattr_getschedparam or posix_spawnattr_setschedparam Subroutine](#) on page 1139, [posix_spawnattr_getschedpolicy or posix_spawnattr_setschedpolicy Subroutine](#) on page 1140, [posix_spawnattr_getsigmask or posix_spawnattr_setsigmask Subroutine](#) on page 1142.

**posixspawnattr_getflags or posixspawnattr_setflags Subroutine**

**Purpose**
Gets and sets the `spawn-flags` attribute of a spawn attributes object.

**Syntax**

```
#include <spawn.h>

int posix_spawnattr_getflags(const posix_spawnattr_t *restrict attr,
                             short *restrict flags);

int posix_spawnattr_setflags(posix_spawnattr_t *attr, short flags);
```

**Description**
The `posix_spawnattr_getflags` subroutine obtains the value of the `spawn-flags` attribute from the attributes object referenced by `attr`. The `posix_spawnattr_setflags` subroutine sets the `spawn-flags` attribute in an initialized attributes object referenced by `attr`. The `spawn-flags` attribute is used to indicate which process attributes are to be changed in the new process image when invoking `posix_spawn` or `posix_spawnp`. It is the bitwise-inclusive OR of 0 or more of the following flags:

- `POSIX_SPAWN_RESETIDS`
- `POSIX_SPAWN_SETPGROUP`
- `POSIX_SPAWN_SETSIGDEF`
- `POSIX_SPAWN_SETSIGMASK`
- `POSIX_SPAWN_SETSCHEDPARAM`
- `POSIX_SPAWN_SETSCHEDULER`

These flags are defined in the `spawn.h` header file. The default value of this attribute is as if no flags were set.

**Return Values**

Upon successful completion, the `posix_spawnattr_getflags` subroutine returns 0 and stores the value of the `spawn-flags` attribute of `attr` into the object referenced by the `flags` parameter; otherwise, an error number is returned to indicate the error.

Upon successful completion, the `posix_spawnattr_setflags` subroutine returns 0; otherwise, an error number is returned to indicate the error.
Error Codes
The `posix_spawnattr_getflags` and `posix_spawnattr_setflags` subroutines will fail if the following is true:

EINVAL The value specified by attr is invalid.

The `posix_spawnattr_setflags` subroutine might fail if the following is true:

EINVAL The value of the attribute being set is not valid.

Related Information
The "posix_spawn or posix_spawnp Subroutine" on page 1129, "posix_spawn_file_actions_addclose or posix_spawn_file_actions_addopen Subroutine" on page 1133, "posix_spawn_file_actions_adddup2 Subroutine" on page 1134, "posix_spawn_file_actions_destroy or posix_spawn_file_actions_init Subroutine" on page 1135, "posix_spawnattr_destroy or posix_spawnattr_init Subroutine" on page 1136, "posix_spawnattr_getsigdefault or posix_spawnattr_setsigdefault Subroutine" on page 1141, "posix_spawnattr_getpgroup or posix_spawnattr_setpgroup Subroutine," "posix_spawnattr_getschedparam or posix_spawnattr_setschedparam Subroutine" on page 1139, "posix_spawnattr_getschedpolicy or posix_spawnattr_setschedpolicy Subroutine" on page 1140, "posix_spawnattr_getsigmask or posix_spawnattr_setsigmask Subroutine" on page 1142

`posix_spawnattr_getpgroup` or `posix_spawnattr_setpgroup` Subroutine

Purpose
Gets and sets the spawn-pgroup attribute of a spawn attributes object.

Syntax
```
#include <spawn.h>

int posix_spawnattr_getpgroup(const posix_spawnattr_t *restrict attr, pid_t *restrict pgroup);
int posix_spawnattr_setpgroup(posix_spawnattr_t *attr, pid_t pgroup);
```

Description
The `posix_spawnattr_getpgroup` subroutine gets the value of the spawn-pgroup attribute from the attributes object referenced by attr.

The `posix_spawnattr_setpgroup` subroutine sets the spawn-pgroup attribute in an initialized attributes object referenced by attr.

The spawn-pgroup attribute represents the process group to be joined by the new process image in a spawn operation (if POSIX_SPAWN_SETPGROUP is set in the spawn-flags attribute). The default value of this attribute is 0.

Return Values
Upon successful completion, the `posix_spawnattr_getpgroup` subroutine returns 0 and stores the value of the spawn-pgroup attribute of attr into the object referenced by the pgroup parameter; otherwise, an error number is returned to indicate the error.

Upon successful completion, the `posix_spawnattr_setpgroup` subroutine returns 0; otherwise, an error number is returned to indicate the error.
Error Codes
The `posix_spawnattr_getpgroup` and `posix_spawnattr_setpgroup` subroutines might fail if the following is true:

EINVAL The value specified by `attr` is invalid.

The `posix_spawnattr_setpgroup` subroutine might fail if the following is true:

EINVAL The value of the attribute being set is not valid.

Related Information
The `posix_spawn or posix_spawnp Subroutine” on page 1129, "posix_spawn_file_actions_addclose or posix_spawn_file_actions_addopen Subroutine” on page 1133, "posix_spawn_file_actions_adddup2 Subroutine” on page 1134, "posix_spawn_file_actions_destroy or posix_spawn_file_actions_init Subroutine” on page 1135, "posix_spawnattr_destroy or posix_spawnattr_init Subroutine” on page 1136, "posix_spawnattr_getpgroup or posix_spawnattr_setpgroup Subroutine” on page 1137, "posix_spawnattr_getsigdefault or posix_spawnattr_setsigdefault Subroutine” on page 1141, "posix_spawnattr_getflags or posix_spawnattr_setflags Subroutine” on page 1140, "posix_spawnattr_getsigmask or posix_spawnattr_setsigmask Subroutine” on page 1142

`posix_spawnattr_getschedparam` or `posix_spawnattr_setschedparam` Subroutine

Purpose
Gets and sets the `spawn-schedparam` attribute of a spawn attributes object.

Syntax
```c
#include <spawn.h>
#include <sched.h>

int posix_spawnattr_getschedparam(const posix_spawnattr_t * restrict attr, struct sched_param *restrict schedparam);
int posix_spawnattr_setschedparam(posix_spawnattr_t *restrict attr, const struct sched_param *restrict schedparam);
```

Description
The `posix_spawnattr_getschedparam` subroutine gets the value of the `spawn-schedparam` attribute from the attributes object referenced by `attr`.

The `posix_spawnattr_setschedparam` subroutine sets the `spawn-schedparam` attribute in an initialized attributes object referenced by `attr`.

The `spawn-schedparam` attribute represents the scheduling parameters to be assigned to the new process image in a spawn operation (if `POSIX_SPAWN_SETSCHEDULER` or `POSIX_SPAWN_SETSCHEDPARAM` is set in the `spawn-flags` attribute). The default value of this attribute is unspecified.

Return Values
Upon successful completion, the `posix_spawnattr_getschedparam` subroutine returns 0 and stores the value of the `spawn-schedparam` attribute of `attr` into the object referenced by the `schedparam` parameter; otherwise, an error number is returned to indicate the error.
Upon successful completion, the `posix_spawnattr_setschedparam` subroutine returns 0; otherwise, an error number is returned to indicate the error.

### Error Codes

The `posix_spawnattr_getschedparam` and `posix_spawnattr_setschedparam` subroutines might fail if the following is true:

- **EINVAL**: The value specified by `attr` is invalid.

The `posix_spawnattr_setschedparam` subroutine might fail if the following is true:

- **EINVAL**: The value of the attribute being set is not valid.

### Related Information

The [“posix_spawn or posix_spawnp Subroutine” on page 1129](#), [“posix_spawn_file_actions_addclose or posix_spawn_file_actions_addopen Subroutine” on page 1133](#), [“posix_spawn_file_actions_adddup2 Subroutine” on page 1134](#), [“posix_spawn_file_actions_destroy or posix_spawn_file_actions_init Subroutine” on page 1135](#), [“posix_spawnattr_destroy or posix_spawnattr_init Subroutine” on page 1136](#), [“posix_spawnattr_getsigdefault or posix_spawnattr_setsigdefault Subroutine” on page 1141](#), [“posix_spawnattr_getflags or posix_spawnattr_setflags Subroutine” on page 1137](#), [“posix_spawnattr_getpgroup or posix_spawnattr_setpgroup Subroutine” on page 1138](#), [“posix_spawnattr_getschedpolicy or posix_spawnattr_setschedpolicy Subroutine” on page 1139](#), [“posix_spawnattr_getsigmask or posix_spawnattr_setsigmask Subroutine” on page 1142](#)

---

### posix_spawnattr_getschedpolicy or posix_spawnattr_setschedpolicy Subroutine

#### Purpose

Gets and sets the `spawn-schedpolicy` attribute of a spawn attributes object.

#### Syntax

```c
#include <spawn.h>
#include <sched.h>

int posix_spawnattr_getschedpolicy(const posix_spawnattr_t *attr, int *restrict schedpolicy);

int posix_spawnattr_setschedpolicy(posix_spawnattr_t *attr, int schedpolicy);
```

#### Description

The `posix_spawnattr_getschedpolicy` subroutine gets the value of the `spawn-schedpolicy` attribute from the attributes object referenced by `attr`.

The `posix_spawnattr_setschedpolicy` subroutine sets the `spawn-schedpolicy` attribute in an initialized attributes object referenced by `attr`.

The `spawn-schedpolicy` attribute represents the scheduling policy to be assigned to the new process image in a spawn operation (if `POSIX_SPAWN_SETSCHEDULER` is set in the spawn-flags attribute). The default value of this attribute is unspecified.
Return Values
Upon successful completion, the `posix_spawnattr_getschedpolicy` subroutine returns 0 and stores the value of the `spawn-schedpolicy` attribute of `attr` into the object referenced by the `schedpolicy` parameter; otherwise, an error number is returned to indicate the error.

Upon successful completion, `posix_spawnattr_setschedpolicy` returns 0; otherwise, an error number is returned to indicate the error.

Error Codes
The following `posix_spawnattr_getschedpolicy` and `posix_spawnattr_setschedpolicy` subroutines might fail if the following is true:

- **EINVAL** The value specified by `attr` is invalid.

The `posix_spawnattr_setschedpolicy` subroutine might fail if the following is true:

- **EINVAL** The value of the attribute being set is not valid.

Related Information
The following subroutines might fail if the following is true:

- **EINVAL** The value specified by `attr` is invalid.

```
posix_spawnattr_getsigdefault or posix_spawnattr_setsigdefault Subroutine
```

Purpose
Gets and sets the `spawn-sigdefault` attribute of a spawn attributes object.

Syntax
```
#include <signal.h>
#include <spawn.h>

int posix_spawnattr_getsigdefault(const posix_spawnattr_t *attr, sigset_t *restrict sigdefault);
int posix_spawnattr_setsigdefault(posix_spawnattr_t *restrict attr, const sigset_t *restrict sigdefault);
```

Description
The `posix_spawnattr_getsigdefault` subroutine gets the value of the `spawn-sigdefault` attribute from the attributes object referenced by `attr`.

The `posix_spawnattr_setsigdefault` subroutine sets the `spawn-pgroup` attribute in an initialized attributes object referenced by `attr`. 
The `spawn-sigdefault` attribute represents the set of signals to be forced to default signal handling in the new process image by a spawn operation (if `POSIX_SPAWN_SETSIGDEF` is set in the `spawn-flags` attribute). The default value of this attribute is an empty signal set.

**Return Values**

Upon successful completion, the `posix_spawnattr_getsigdefault` subroutine returns 0 and stores the value of the `spawn-sigdefault` attribute of `attr` into the object referenced by the `sigdefault` parameter; otherwise, an error number is returned to indicate the error.

Upon successful completion, the `posix_spawnattr_setsigdefault` subroutine returns 0; otherwise, an error number is returned to indicate the error.

**Error Codes**

The `posix_spawnattr_getsigdefault` and `posix_spawnattr_setsigdefault` subroutines might fail if the following is true:

- **EINVAL** The value specified by `attr` is invalid.

The `posix_spawnattr_setsigdefault` subroutine might fail if the following is true:

- **EINVAL** The value of the attribute being set is not valid.

**Related Information**

The following documentation is related:
- "posix_spawn or posix_spawnp Subroutine" on page 1129
- "posix_spawn_file_actions_addclose or posix_spawn_file_actions_addopen Subroutine" on page 1133
- "posix_spawn_file_actions_adddup2 Subroutine" on page 1134
- "posix_spawn_file_actions_destroy or posix_spawn_file_actions_init Subroutine" on page 1135
- "posix_spawnattr_destroy or posix_spawnattr_init Subroutine" on page 1136
- "posix_spawnattr_getflags or posix_spawnattr_setflags Subroutine" on page 1137
- "posix_spawnattr_getpgroup or posix_spawnattr_setpgroup Subroutine" on page 1138
- "posix_spawnattr_getschedparam or posix_spawnattr_setschedparam Subroutine" on page 1139
- "posix_spawnattr_getschedpolicy or posix_spawnattr_setschedpolicy Subroutine" on page 1140
- "posix_spawnattr_getsigmask or posix_spawnattr_setsigmask Subroutine" on page 1141

---

**posix_spawnattr_getsigmask or posix_spawnattr_setsigmask Subroutine**

**Purpose**

Gets and sets the `spawn-sigmask` attribute of a spawn attributes object.

**Syntax**

```c
#include <signal.h>
#include <spawn.h>

int posix_spawnattr_getsigmask(const posix_spawnattr_t *restrict attr, 
                                sigset_t *restrict sigmask);

int posix_spawnattr_setsigmask(posix_spawnattr_t *restrict attr, 
                                const sigset_t *restrict sigmask);
```

**Description**

The `posix_spawnattr_getsigmask` subroutine gets the value of the `spawn-sigmask` attribute from the attributes object referenced by `attr`. 
The **posix_spawnattr_setsigmask** subroutine sets the **spawn-sigmask** attribute in an initialized attributes object referenced by **attr**.

The **spawn-sigmask** attribute represents the signal mask in effect in the new process image of a spawn operation (if **POSIX_SPAWN_SETSIGMASK** is set in the **spawn-flags** attribute). The default value of this attribute is unspecified.

**Return Values**
Upon successful completion, the **posix_spawnattr_getsigmask** subroutine returns 0 and stores the value of the **spawn-sigmask** attribute of **attr** into the object referenced by the **sigmask** parameter; otherwise, an error number is returned to indicate the error.

Upon successful completion, the **posix_spawnattr_setsigmask** subroutine returns 0; otherwise, an error number is returned to indicate the error.

**Error Codes**
The **posix_spawnattr_getsigmask** and **posix_spawnattr_setsigmask** subroutines might fail if the following is true:

- **EINVAL** The value specified by **attr** is invalid.

The **posix_spawnattr_setsigmask** subroutine might fail if the following is true:

- **EINVAL** The value of the attribute being set is not valid.

**Related Information**
The **posix_trace_getnext_event**, **posix_trace_timedgetnext_event**, **posix_trace_trygetnext_event** Subroutines

**posix_trace_getnext_event**, **posix_trace_timedgetnext_event**, **posix_trace_trygetnext_event** Subroutine

**Purpose**
Retrieves a trace event (TRACING).

**Syntax**
```c
#include <sys/types.h>
#include <trace.h>

int posix_trace_getnext_event(trace_id_t trid,
   struct posix_trace_event_info *restrict event,
   void *restrict data, size_t num_bytes,
   size_t *restrict data_len, int *restrict unavailable);

int posix_trace_timedgetnext_event(trace_id_t trid,
   struct posix_trace_event_info *restrict event,
```

Base Operating System (BOS) Runtime Services (A-P) 1143
The `posix_trace_getnext_event()` function reports a recorded trace event either from an active trace stream without log or a prerecorded trace stream identified by the `trid` argument. The `posix_trace_trygetnext_event()` function reports a recorded trace event from an active trace stream without log identified by the `trid` argument.

The trace event information associated with the recorded trace event is copied by the function into the structure pointed to by the `event` argument, and the data associated with the trace event is copied into the buffer pointed to by the `data` argument.

The `posix_trace_getnext_event()` function blocks if the `trid` argument identifies an active trace stream and there is currently no trace event ready to be retrieved. When returning, if a recorded trace event was reported, the variable pointed to by the `unavailable` argument is set to 0. Otherwise, the variable pointed to by the `unavailable` argument is set to a value different from 0.

The `posix_trace_timedgetnext_event()` function attempts to get another trace event from an active trace stream without log, as in the `posix_trace_getnext_event()` function. However, if no trace event is available from the trace stream, the implied wait terminates when the timeout specified by the argument `abs_timeout` expires, and the function returns the error [ETIMEDOUT].

The timeout expires when the absolute time specified by `abs_timeout` passes—as measured by the clock upon which timeouts are based (that is, when the value of that clock equals or exceeds `abs_timeout`)—or when the absolute time specified by `abs_timeout` has already passed at the time of the call.

If the Timers option is supported, the timeout is based on the CLOCK_REALTIME clock; if the Timers option is not supported, the timeout is based on the system clock as returned by the `time()` function. The resolution of the timeout matches the resolution of the clock on which it is based. The `timespec` data type is defined in the `<time.h>` header.

The function never fails with a timeout if a trace event is immediately available from the trace stream. The validity of the `abs_timeout` argument does not need to be checked if a trace event is immediately available from the trace stream.

The behavior of this function for a prerecorded trace stream is unspecified.

The `posix_trace_trygetnext_event()` function does not block. This function returns an error if the `trid` argument identifies a prerecorded trace stream. If a recorded trace event was reported, the variable pointed to by the `unavailable` argument is set to 0. Otherwise, if no trace event was reported, the variable pointed to by the `unavailable` argument is set to a value different from 0.

The `num_bytes` argument equals the size of the buffer pointed to by the `data` argument. The `data_len` argument reports to the application the length, in bytes, of the data record just transferred. If `num_bytes` is greater than or equal to the size of the data associated with the trace event pointed to by the `event` argument, all the recorded data is transferred. In this case, the truncation-status member of the trace event structure is either POSIX_TRACE_NOT_TRUNCATED (if the trace event data was recorded without truncation while tracing) or POSIX_TRACE_TRUNCATED_RECORD (if the trace event data was truncated when it was recorded). If the `num_bytes` argument is less than the length of recorded trace event data, the data transferred is truncated to a length of `num_bytes`, the value stored in the variable pointed to by
data_len equals num_bytes, and the truncation-status member of the event structure argument is set to POSIX_TRACE_TRUNCATED_READ (see the posix_trace_event_info structure defined in <trace.h>.

The report of a trace event is sequential starting from the oldest recorded trace event. Trace events are reported in the order in which they were generated, up to an implementation-defined time resolution that causes the ordering of trace events occurring very close to each other to be unknown. After it is reported, a trace event cannot be reported again from an active trace stream. After a trace event is reported from an active trace stream without log, the trace stream makes the resources associated with that trace event available to record future generated trace events.

Return Values
Upon successful completion, these functions return a value of 0. Otherwise, they return the corresponding error number.

If successful, these functions store:
• The recorded trace event in the object pointed to by event
• The trace event information associated with the recorded trace event in the object pointed to by data
• The length of this trace event information in the object pointed to by data_len
• The value of 0 in the object pointed to by unavailable

Error Codes
These functions fail if:

EINVAL The trace stream identifier argument trid is invalid.

The posix_trace_getnext_event() and posix_trace_timedgetnext_event() functions fail if:

EINTR The operation was interrupted by a signal, and so the call had no effect.

The posix_trace_trygetnext_event() function fails if:

EINVAL The trace stream identifier argument trid does not correspond to an active trace stream.

The posix_trace_timedgetnext_event() function fails if:

EINVAL There is no trace event immediately available from the trace stream, and the timeout argument is invalid.
ETIMEDOUT No trace event was available from the trace stream before the specified timeout expired.

Related Information
"mq_receive, mq_timedreceive Subroutine" on page 861, "mq_send, mq_timedsend Subroutine" on page 863, "pthread_mutex_timedlock Subroutine" on page 1251, "pthread_rwlock_timedrdlock Subroutine" on page 1266, "pthread_rwlock_timedwrlock Subroutine" on page 1268.


The pthread.h and time.h files in AIX 5L Version 5.3 Files Reference.
powf, powl, or pow Subroutine

Purpose
Computes power.

Syntax
#include <math.h>

float powf (x, y)
float x;
float y;

long double powl (x, y)
long double x, y;

double pow (x, y)
double x, y;

Description
The powf, powl, and pow subroutines compute the value of x raised to the power y, $x^y$. If x is negative, the application ensures that y is an integer value.

An application wishing to check for error situations should set errno to zero and call feclearexcept(FE_ALL_EXCEPT) before calling these subroutines. Upon return, if errno is nonzero or fetestexcept(FE_INVALID | FE_DIVBYZERO | FE_OVERFLOW | FE_UNDERFLOW) is nonzero, an error has occurred.

Parameters
x Specifies the value of the base.
y Specifies the value of the exponent.

Return Values
Upon successful completion, the pow, powf and powl subroutines return the value of x raised to the power y.

For finite values of $x < 0$, and finite non-integer values of y, a domain error occurs and a NaN is returned.

If the correct value would cause overflow, a range error occurs and the pow, powf, and powl subroutines return HUGE_VAL, HUGE_VALF, and HUGE_VALL, respectively.

If the correct value would cause underflow, and is not representable, a range error may occur, and 0.0 is returned.

If x or y is a NaN, a NaN is returned (unless specified elsewhere in this description).

For any value of y (including NaN), if x is +1, 1.0 is returned.

For any value of x (including NaN), if y is ±0, 1.0 is returned.

For any odd integer value of y>0, if x is ±0, ±0 is returned.

For $y > 0$ and not an odd integer, if x is ±0, +0 is returned.
If \( x \) is -1, and \( y \) is \( \pm \text{Inf} \), 1.0 is returned.

For \( |x| < 1 \), if \( y \) is \( -\text{Inf} \), \( +\text{Inf} \) is returned.

For \( |x| > 1 \), if \( y \) is \( -\text{Inf} \), \( +0 \) is returned.

For \( |x| < 1 \), if \( y \) is \( +\text{Inf} \), \( +0 \) is returned.

For \( |x| > 1 \), if \( y \) is \( +\text{Inf} \), \( +\text{Inf} \) is returned.

For \( y \) an odd integer < 0, if \( x \) is \( -\text{Inf} \), \( -0 \) is returned.

For \( y < 0 \) and not an odd integer, if \( x \) is \( -\text{Inf} \), \( +0 \) is returned.

For \( y \) an odd integer > 0, if \( x \) is \( -\text{Inf} \), \( -\text{Inf} \) is returned.

For \( y > 0 \) and not an odd integer, if \( x \) is \( -\text{Inf} \), \( +\text{Inf} \) is returned.

For \( y < 0 \), if \( x \) is \( +\text{Inf} \), \( +0 \) is returned.

For \( y > 0 \), if \( x \) is \( +\text{Inf} \), \( +\text{Inf} \) is returned.

For \( y \) an odd integer < 0, if \( x \) is \( \pm 0 \), a pole error occurs and \( \pm \text{HUGE\_VAL} \), \( \pm \text{HUGE\_VALF} \), and \( \pm \text{HUGE\_VALL} \) is returned for \texttt{pow}, \texttt{powf}, and \texttt{powl}, respectively.

For \( y < 0 \) and not an odd integer, if \( x \) is \( \pm 0 \), a pole error occurs and \texttt{HUGE\_VAL}, \texttt{HUGE\_VALF} and \texttt{HUGE\_VALL} is returned for \texttt{pow}, \texttt{powf}, and \texttt{powl}, respectively.

If the correct value would cause underflow, and is representable, a range error may occur and the correct value is returned.

**Error Codes**

When using the \texttt{libm.a} library:

- \texttt{pow}
  - If the correct value overflows, the \texttt{pow} subroutine returns a \texttt{HUGE\_VAL} value and sets \texttt{errno} to \texttt{ERANGE}. If the \( x \) parameter is negative and the \( y \) parameter is not an integer, the \texttt{pow} subroutine returns a \texttt{NaNQ} value and sets \texttt{errno} to \texttt{EDOM}. If \( x=0 \) and the \( y \) parameter is negative, the \texttt{pow} subroutine returns a \texttt{HUGE\_VAL} value but does not modify \texttt{errno}.

- \texttt{powl}
  - If the correct value overflows, the \texttt{powl} subroutine returns a \texttt{HUGE\_VAL} value and sets \texttt{errno} to \texttt{ERANGE}. If the \( x \) parameter is negative and the \( y \) parameter is not an integer, the \texttt{powl} subroutine returns a \texttt{NaNQ} value and sets \texttt{errno} to \texttt{EDOM}. If \( x=0 \) and the \( y \) parameter is negative, the \texttt{powl} subroutine returns a \texttt{HUGE\_VAL} value but does not modify \texttt{errno}. 


When using libmsaa.a(-lmsaa):

**pow**

If \( x=0 \) and the \( y \) parameter is not positive, or if the \( x \) parameter is negative and the \( y \) parameter is not an integer, the **pow** subroutine returns 0 and sets **errno** to **EDOM**. In these cases a message indicating DOMAIN error is output to standard error. When the correct value for the **pow** subroutine would overflow or underflow, the **pow** subroutine returns:

\[ +\text{HUGE\_VAL} \]

OR

\[ -\text{HUGE\_VAL} \]

OR

0

When using either the **libm.a** library or the **libsaa.a** library:

**powl**

If the correct value overflows, **powl** returns **HUGE\_VAL** and **errno** to **ERANGE**. If \( x \) is negative and \( y \) is not an integer, **powl** returns **NaNQ** and sets **errno** to **EDOM**. If \( x = 0 \) and \( y \) is negative, **powl** returns a **HUGE\_VAL** value but does not modify **errno**.

**Related Information**

"exp, expf, or expl Subroutine" on page 244, "fclearexcept Subroutine" on page 262, "fetestexcept Subroutine" on page 270, and "class, _class, finite, isnan, or unordered Subroutines" on page 167.

**math.h** in AIX 5L Version 5.3 Files Reference.

**printf, fprintf, sprintf, snprintf, vsprintf, vfprintf, vprintf, vsprintf, or vwsprintf Subroutine**

**Purpose**

Prints formatted output.

**Library**

Standard C Library (**libc.a**) or the Standard C Library with 128-Bit long doubles (**libc128.a**)

**Syntax**

```c
#include <stdio.h>

int printf (Format, [Value, ...])
const char *Format;

int fprintf (Stream, Format, [Value, ...])
FILE *Stream;
const char *Format;

int sprintf (String, Format, [Value, ...])
char *String;
const char *Format;

int snprintf (String, Number, Format, [Value, ...])
char *String;
int Number;
const char *Format;
#include <stdarg.h>

int vprintf (Format, Value)
```
const char *Format;
va_list Value;

int vfprintf (Stream, Format, Value)
FILE *Stream;
const char *Format;
va_list Value;

int vsprintf (String, Format, Value)
char *String;
const char *Format;
va_list Value;
#include <wchar.h>

int wvsprintf (String, Format, Value)
wchar_t *String;
const char *Format;
va_list Value;

int wprintf (String, Format, [Value, ...])
wchar_t *String;
const char *Format;

Description
The printf subroutine converts, formats, and writes the Value parameter values, under control of the Format parameter, to the standard output stream. The printf subroutine provides conversion types to handle code points and wchar_t wide character codes.

The fprintf subroutine converts, formats, and writes the Value parameter values, under control of the Format parameter, to the output stream specified by the Stream parameter. This subroutine provides conversion types to handle code points and wchar_t wide character codes.

The sprintf subroutine converts, formats, and stores the Value parameter values, under control of the Format parameter, into consecutive bytes, starting at the address specified by the String parameter. The sprintf subroutine places a null character (\0) at the end. You must ensure that enough storage space is available to contain the formatted string. This subroutine provides conversion types to handle code points and wchar_t wide character codes.

The snprintf subroutine converts, formats, and stores the Value parameter values, under control of the Format parameter, into consecutive bytes, starting at the address specified by the String parameter. The snprintf subroutine places a null character (\0) at the end. You must ensure that enough storage space is available to contain the formatted string. This subroutine provides conversion types to handle code points and wchar_t wide character codes. The snprintf subroutine is identical to the sprintf subroutine with the addition of the Number parameter, which states the size of the buffer referred to by the String parameter.

The wsprintf subroutine converts, formats, and stores the Value parameter values, under control of the Format parameter, into consecutive wchar_t characters starting at the address specified by the String parameter. The wsprintf subroutine places a null character (\0) at the end. The calling process should ensure that enough storage space is available to contain the formatted string. The field width unit is specified as the number of wchar_t characters. The wsprintf subroutine is the same as the printf subroutine, except that the String parameter for the wsprintf subroutine uses a string of wchar_t wide-character codes.

All of the above subroutines work by calling the _doprnt subroutine, using variable-length argument facilities of the varargs macros.

The vprintf, vfprintf, vsprintf, and wvsprintf subroutines format and write varargs macros parameter lists. These subroutines are the same as the printf, fprintf, sprintf, snprintf, and wsprintf subroutines,
respectively, except that they are not called with a variable number of parameters. Instead, they are called
with a parameter-list pointer as defined by the varargs macros.

Parameters

Number
Specifies the number of bytes in a string to be copied or transformed.

Value
Specifies 0 or more arguments that map directly to the objects in the Format parameter.

Stream
Specifies the output stream.

String
Specifies the starting address.

Format
A character string that contains two types of objects:

- Plain characters, which are copied to the output stream.
- Conversion specifications, each of which causes 0 or more items to be retrieved from the Value parameter list. In the case of the vprintf, vfprintf, vsprintf, and vwsprintf subroutines, each conversion specification causes 0 or more items to be retrieved from the varargs parameter lists.

If the Value parameter list does not contain enough items for the Format parameter, the results are unpredictable. If more parameters remain after the entire Format parameter has been processed, the subroutine ignores them.

Each conversion specification in the Format parameter has the following elements:

- A % (percent sign).
- 0 or more options, which modify the meaning of the conversion specification. The option characters and their meanings are:
  - A % (percent sign).
  - 0 or more options, which modify the meaning of the conversion specification. The option characters and their meanings are:
    - Formats the integer portions resulting from i, d, u, f, g and G decimal conversions with thousands_sep grouping characters. For other conversions the behavior is undefined. This option uses the nonmonetary grouping character.
    - Left-justifies the result of the conversion within the field.
    - Begins the result of a signed conversion with a + (plus sign) or - (minus sign).

  space character
  Prefixes a space character to the result if the first character of a signed conversion is not a sign. If both the space-character and + option characters appear, the space-character option is ignored.

  #
  Converts the value to an alternate form. For c, d, s, and u conversions, the option has no effect. For o conversion, it increases the precision to force the first digit of the result to be a 0. For x and X conversions, a nonzero result has a 0x or 0X prefix. For e, E, f, g, and G conversions, the result always contains a decimal point, even if no digits follow it. For g and G conversions, trailing 0’s are not removed from the result.

  0
  Pads to the field width with leading 0’s (following any indication of sign or base) for d, i, o, u, x, X, e, E, f, g, and G conversions; the field is not space-padded. If the 0 and - options both appear, the 0 option is ignored. For d, i, o u, x, and X conversions, if a precision is specified, the 0 option is also ignored. If the 0 and ’ options both appear, grouping characters are inserted before the field is padded. For other conversions, the results are unreliable.

  B
  Specifies a no-op character.

  N
  Specifies a no-op character.

  J
  Specifies a no-op character.
• An optional decimal digit string that specifies the minimum field width. If the converted value has fewer characters than the field width, the field is padded on the left to the length specified by the field width. If the - (left-justify) option is specified, the field is padded on the right.

• An optional precision. The precision is a . (dot) followed by a decimal digit string. If no precision is specified, the default value is 0. The precision specifies the following limits:
  – Minimum number of digits to appear for the d, i, o, u, x, or X conversions.
  – Number of digits to appear after the decimal point for the e, E, and f conversions.
  – Maximum number of significant digits for g and G conversions.
  – Maximum number of bytes to be printed from a string in s and S conversions.
  – Maximum number of bytes, converted from the wchar_t array, to be printed from the S conversions. Only complete characters are printed.

• An optional l (lowercase L), ll (lowercase LL), h, or L specifier indicates one of the following:
  – An optional h specifying that a subsequent d, i, u, o, x, or X conversion specifier applies to a short int or unsigned short int Value parameter (the parameter will have been promoted according to the integral promotions, and its value will be converted to a short int or unsigned short int before printing).
  – An optional h specifying that a subsequent n conversion specifier applies to a pointer to a short int parameter.
  – An optional l (lowercase L) specifying that a subsequent d, i, u, o, x, or X conversion specifier applies to a long int or unsigned long int parameter.
  – An optional l (lowercase L) specifying that a subsequent n conversion specifier applies to a pointer to a long int parameter.
  – An optional ll (lowercase LL) specifying that a subsequent d, i, u, o, x, or X conversion specifier applies to a long long int or unsigned long long int parameter.
  – An optional ll (lowercase LL) specifying that a subsequent n conversion specifier applies to a pointer to a long long int parameter.
  – An optional L specifying that a following e, E, f, g, or G conversion specifier applies to a long double parameter. If linked with libc.a, long double is the same as double (64bits). If linked with libc128.a and libc.a, long double is 128 bits.

• An optional vl, lv, vh, hv or v specifier indicates one of the following vector data type conversions:
  – An optional v specifying that a following e, E, f, g, G, a, or A conversion specifier applies to a vector float parameter. It consumes one argument and interprets the data as a series of four 4-byte floating point components.
  – An optional v specifying that a following c, d, i, u, o, x, or X conversion specifier applies to a vector signed char, vector unsigned char, or vector bool char parameter. It consumes one argument and interprets the data as a series of sixteen 1-byte components.
  – An optional vl or lv specifying that a following d, i, u, o, x, or X conversion specifier applies to a vector signed int, vector unsigned int, or vector bool parameter. It consumes one argument and interprets the data as a series of four 4-byte integer components.
  – An optional vh or hv specifying that a following d, i, u, o, x, or X conversion specifier applies to a vector signed short or vector unsigned short parameter. It consumes one argument and interprets the data as a series of eight 2-byte integer components.
  – For any of the preceding specifiers, an optional separator character can be specified immediately preceding the vector size specifier. If no separator is specified, the default separator is a space unless the conversion is c, in which case the default separator is null. The set of supported optional separators are , (comma), ; (semicolon), : (colon), and _ (underscore).

• The following characters indicate the type of conversion to be applied:
  % Performs no conversion. Prints (%).
d or i Accepts a Value parameter specifying an integer and converts it to signed decimal notation. The precision specifies the minimum number of digits to appear. If the value being converted can be represented in fewer digits, it is expanded with leading 0's. The default precision is 1. The result of converting a value of 0 with a precision of 0 is a null string. Specifying a field width with a 0 as a leading character causes the field-width value to be padded with leading 0's.

u Accepts a Value parameter specifying an unsigned integer and converts it to unsigned decimal notation. The precision specifies the minimum number of digits to appear. If the value being converted can be represented in fewer digits, it is expanded with leading 0's. The default precision is 1. The result of converting a value of 0 with a precision of 0 is a null string. Specifying a field width with a 0 as a leading character causes the field-width value to be padded with leading 0's.

o Accepts a Value parameter specifying an unsigned integer and converts it to unsigned octal notation. The precision specifies the minimum number of digits to appear. If the value being converted can be represented in fewer digits, it is expanded with leading 0's. The default precision is 1. The result of converting a value of 0 with a precision of 0 is a null string. Specifying a field width with a 0 as a leading character causes the field width value to be padded with leading 0's. An octal value for field width is not implied.

x or X Accepts a Value parameter specifying an unsigned integer and converts it to unsigned hexadecimal notation. The letters abcdef are used for the x conversion and the letters ABCDEF are used for the X conversion. The precision specifies the minimum number of digits to appear. If the value being converted can be represented in fewer digits, it is expanded with leading 0's. The default precision is 1. The result of converting a value of 0 with a precision of 0 is a null string. Specifying a field width with a 0 as a leading character causes the field width value to be padded with leading 0's.

f Accepts a Value parameter specifying a double and converts it to decimal notation in the format [-]ddd.ddd. The number of digits after the decimal point is equal to the precision specification. If no precision is specified, six digits are output. If the precision is 0, no decimal point appears.

e or E Accepts a Value parameter specifying a double and converts it to the exponential form [-]d.ddd e+/-dd. One digit exists before the decimal point, and the number of digits after the decimal point is equal to the precision specification. The precision specification can be in the range of 0-17 digits. If no precision is specified, six digits are output. If the precision is 0, no decimal point appears. The E conversion character produces a number with E instead of e before the exponent. The exponent always contains at least two digits.

g or G Accepts a Value parameter specifying a double and converts it in the style of the e, E, or f conversion characters, with the precision specifying the number of significant digits. Trailing 0's are removed from the result. A decimal point appears only if it is followed by a digit. The style used depends on the value converted. Style e (E, if G is the flag used) results only if the exponent resulting from the conversion is less than -4, or if it is greater or equal to the precision. If an explicit precision is 0, it is taken as 1.

c Accepts and prints a Value parameter specifying an integer converted to an unsigned char data type.

C Accepts and prints a Value parameter specifying a wchar_t wide character code. The wchar_t wide character code specified by the Value parameter is converted to an array of bytes representing a character and that character is written; the Value parameter is written without conversion when using the wprintf subroutine.

s Accepts a Value parameter as a string (character pointer), and characters from the string are printed until a null character (\0) is encountered or the number of bytes
indicated by the precision is reached. If no precision is specified, all bytes up to the first
null character are printed. If the string pointer specified by the Value parameter has a
null value, the results are unreliable.

S Accepts a corresponding Value parameter as a pointer to a wchar_t string. Characters
from the string are printed (without conversion) until a null character (\0) is encountered
or the number of wide characters indicated by the precision is reached. If no precision
is specified, all characters up to the first null character are printed. If the string pointer
specified by the Value parameter has a value of null, the results are unreliable.

p Accepts a pointer to void. The value of the pointer is converted to a sequence of
printable characters, the same as an unsigned hexadecimal (x).

n Accepts a pointer to an integer into which is written the number of characters
(wide-character codes in the case of the wsprintf subroutine) written to the output
stream by this call. No argument is converted.

A field width or precision can be indicated by an * (asterisk) instead of a digit string. In this case, an
integer Value parameter supplies the field width or precision. The Value parameter converted for output is
not retrieved until the conversion letter is reached, so the parameters specifying field width or precision
must appear before the value (if any) to be converted.

If the result of a conversion is wider than the field width, the field is expanded to contain the converted
result and no truncation occurs. However, a small field width or precision can cause truncation on the right.

The printf, fprintf, sprintf, snprintf, wprintf, vprintf, vsprintf, or vwprintf subroutine allows
the insertion of a language-dependent radix character in the output string. The radix character is defined
by language-specific data in the LC_NUMERIC category of the program’s locale. In the C locale, or in a
locale where the radix character is not defined, the radix character defaults to a . (dot).

After any of these subroutines runs successfully, and before the next successful completion of a call to the
fclose ("fclose or fflush Subroutine" on page 252) or fflush subroutine on the same stream or to the exit
("exit, atexit, unatexit, _exit, or _Exit Subroutine" on page 242) or abort ("abort Subroutine" on page 2)
subroutine, the st_ctime and st_mtime fields of the file are marked for update.

The e, E, f, g, and G conversion specifiers represent the special floating-point values as follows:

<table>
<thead>
<tr>
<th>Specification</th>
<th>Notation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Quiet NaN</td>
<td>+NaNQ or -NaNQ</td>
</tr>
<tr>
<td>Signaling NaN</td>
<td>+NaNS or -NaNS</td>
</tr>
<tr>
<td>+/-INF</td>
<td>+INF or -INF</td>
</tr>
<tr>
<td>+/-0</td>
<td>+0 or -0</td>
</tr>
</tbody>
</table>

The representation of the + (plus sign) depends on whether the + or space-character formatting option is
specified.

These subroutines can handle a format string that enables the system to process elements of the
parameter list in variable order. In such a case, the normal conversion character % (percent sign) is
replaced by %digest$, where digit is a decimal number in the range from 1 to the NL_ARGMAX value.
Conversion is then applied to the specified argument, rather than to the next unused argument. This
feature provides for the definition of format strings in an order appropriate to specific languages. When
variable ordering is used the * (asterisk) specification for field width in precision is replaced by %digest$.
If you use the variable-ordering feature, you must specify it for all conversions.

The following criteria apply:
• The format passed to the NLS extensions can contain either the format of the conversion or the explicit or implicit argument number. However, these forms cannot be mixed within a single format string, except for %n (double percent sign).
• The n value must have no leading zeros.
• If %n$ is used, %1$n to %n - 1$n inclusive must be used.
• The n in %n$ is in the range from 1 to the NL_ARGMAX value, inclusive. See the limits.h file for more information about the NL_ARGMAX value.
• Numbered arguments in the argument list can be referenced as many times as required.
• The * specification for field width or precision is not permitted with the variable order %n$ format; instead, the *m$ format is used.

Return Values
Upon successful completion, the printf, fprintf, vprintf, and vfprintf subroutines return the number of bytes transmitted (not including the null character \0 in the case of the sprintf, and vsprintf subroutines). If an error was encountered, a negative value is output.

Upon successful completion, the snprintf subroutine returns the number of bytes written to the String parameter (excluding the terminating null byte). If output characters are discarded because the output exceeded the Number parameter in length, then the snprintf subroutine returns the number of bytes that would have been written to the String parameter if the Number parameter had been large enough (excluding the terminating null byte).

Upon successful completion, the wprintf and vwprintf subroutines return the number of wide characters transmitted (not including the wide character null character \0). If an error was encountered, a negative value is output.

Error Codes
The printf, fprintf, sprintf, snprintf, or wprintf subroutine is unsuccessful if the file specified by the Stream parameter is unbuffered or the buffer needs to be flushed and one or more of the following are true:

EAGAIN The O_NONBLOCK or O_NDELAY flag is set for the file descriptor underlying the file specified by the Stream or String parameter and the process would be delayed in the write operation.
EBADF The file descriptor underlying the file specified by the Stream or String parameter is not a valid file descriptor open for writing.
EFBIG An attempt was made to write to a file that exceeds the file size limit of this process or the maximum file size. For more information, refer to the ulimit subroutine.
EINTR The write operation terminated due to receipt of a signal, and either no data was transferred or a partial transfer was not reported.

Note: Depending upon which library routine the application binds to, this subroutine may return EINTR. Refer to the signal subroutine regarding sa_restart.

EIO The process is a member of a background process group attempting to perform a write to its controlling terminal, the TOSTOP flag is set, the process is neither ignoring nor blocking the SIGSTO signal, and the process group of the process has no parent process.
ENOSPC No free space remains on the device that contains the file.
EPipe An attempt was made to write to a pipe or first-in-first-out (FIFO) that is not open for reading by any process. A SIGPIPE signal is sent to the process.

The printf, fprintf, sprintf, snprintf, or wprintf subroutine may be unsuccessful if one or more of the following are true:
Examples

The following example demonstrates how the `vfprintf` subroutine can be used to write an error routine:

```c
#include <stdio.h>
#include <stdarg.h>

/* The error routine should be called with the */
/* syntax: */
/* error(routine_name, Format */
/* [value, ...]); */
/* ** Note that the function name and */
/* Format arguments cannot be ** */
/* separately declared because of the ** */
/* definition of varargs. */ { */
va_list args;

va_start(args, fmt);
/* Display the name of the function */
/* that called the error routine */
/* fprintf(stderr, "ERROR in %s: ", */
/* va_arg(args, char *)); */
/* Display the remainder of the message */
/* fmt = va_arg(args, char *); */
vfprintf(fmt, args);
va_end(args);
abort(); }
```

Related Information

The `abort` subroutine, `conv` subroutine, `ecvt`, `fcvt`, or `gcvt` subroutine, `exit`, `atexit`, `unatexit`, `fclose` or `fflush` subroutine, `putc`, `putchar`, `fputc`, or `putw` subroutine, `putwc`, `putwchar`, or `fputwc` subroutine, `scanf`, `fscanf`, `sscanf`, or `wscanf` subroutine, `setlocale` subroutine.

Input and Output Handling and 128-Bit Long Double Floating-Point Data Type in AIX 5L Version 5.3

profil Subroutine

Purpose

Starts and stops program address sampling for execution profiling.

Library

Standard C Library (libc.a)

Syntax

```
#include <mon.h>
```
void profil (ShortBuffer, BufferSize, Offset, Scale)
OR
void profil (ProfBuffer, -1, 0, 0)

unsigned short *ShortBuffer;
struct prof *ProfBuffer;
unsigned int Buffersize, Scale;
unsigned long Offset;

Description
The profil subroutine arranges to record a histogram of periodically sampled values of the calling process program counter. If BufferSize is not -1:

- The parameters to the profil subroutine are interpreted as shown in the first syntax definition.
- After this call, the program counter (pc) of the process is examined each clock tick if the process is the currently active process. The value of the Offset parameter is subtracted from the pc. The result is multiplied by the value of the Scale parameter, shifted right 16 bits, and rounded up to the next half-word aligned value. If the resulting number is less than the BufferSize value divided by sizeof(short), the corresponding short inside the ShortBuffer parameter is incremented. If the result of this increment would overflow an unsigned short, it remains USHRT_MAX.
- The least significant 16 bits of the Scale parameter are interpreted as an unsigned, fixed-point fraction with a binary point at the left. The most significant 16 bits of the Scale parameter are ignored. For example:

<table>
<thead>
<tr>
<th>Octal</th>
<th>Hex</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>0177777</td>
<td>0xFFFF</td>
<td>Maps approximately each pair of bytes in the instruction space to a unique short in the ShortBuffer parameter.</td>
</tr>
<tr>
<td>077777</td>
<td>0x7FFF</td>
<td>Maps approximately every four bytes to a short in the ShortBuffer parameter.</td>
</tr>
<tr>
<td>02</td>
<td>0x0002</td>
<td>Maps all instructions to the same location, producing a noninterrupting core clock.</td>
</tr>
<tr>
<td>01</td>
<td>0x0001</td>
<td>Turns profiling off.</td>
</tr>
<tr>
<td>00</td>
<td>0x0000</td>
<td>Turns profiling off.</td>
</tr>
</tbody>
</table>

Note: Mapping each byte of the instruction space to an individual short in the ShortBuffer parameter is not possible.
- Profiling, using the first syntax definition, is rendered ineffective by giving a value of 0 for the BufferSize parameter.

If the value of the BufferSize parameter is -1:

- The parameters to the profil subroutine are interpreted as shown in the second syntax definition. In this case, the Offset and Scale parameters are ignored, and the ProfBuffer parameter points to an array of prof structures. The prof structure is defined in the mon.h file, and it contains the following members:

  caddr_t p_low;
caddr_t p_high;
HISTCOUNTER *p_buff;
int p_bufsize;
uint p_scale;

If the p_scale member has the value of -1, a value for it is computed based on p_low, p_high, and p_bufsize; otherwise p_scale is interpreted like the scale argument in the first synopsis. The p_high members in successive structures must be in ascending sequence. The array of structures is ended with a structure containing a p_high member set to 0; all other fields in this last structure are ignored.
The p_buff buffer pointers in the array of prof structures must point into a single contiguous buffer space.

- Profiling, using the second syntax definition, is turned off by giving a ProfBuffer argument such that the p_high element of the first structure is equal to 0.

In every case:
- Profiling remains on in both the child process and the parent process after a fork subroutine.
- Profiling is turned off when an exec subroutine is run.
- A call to the profi subroutine is ineffective if profiling has been previously turned on using one syntax definition, and an attempt is made to turn profiling off using the other syntax definition.
- A call to the profi subroutine is ineffective if the call is attempting to turn on profiling when profiling is already turned on, or if the call is attempting to turn off profiling when profiling is already turned off.

**Parameters**

- **ShortBuffer**
  Points to an area of memory in the user address space. Its length (in bytes) is given by the BufferSize parameter.

- **BufferSize**
  Specifies the length (in bytes) of the buffer.

- **Offset**
  Specifies the delta of program counter start and buffer; for example, a 0 Offset implies that text begins at 0. If the user wants to use the entry point of a routine for the Offset parameter, the syntax of the parameter is as follows:

  *(long *)RoutineName

- **Scale**
  Specifies the mapping factor between the program counter and ShortBuffer.

- **ProfBuffer**
  Points to an array of prof structures.

**Return Values**

The profi subroutine always returns a value of 0. Otherwise, the errno global variable is set to indicate the error.

**Error Codes**

The profi subroutine is unsuccessful if one or both of the following are true:

- **EFAULT**
  The address specified by the ShortBuffer or ProfBuffer parameters is not valid, or the address specified by a p_buff field is not valid. EFAULT can also occur if there are not sufficient resources to pin the profiling buffer in real storage.

- **EINVAL**
  The p_high fields in the prof structure specified by the ProfBuffer parameter are not in ascending order.

**Related Information**

The exec ("exec: execl, execle, execcl, execvp, or execve Subroutine" on page 235) subroutines, fork ("fork, f_fork, or vfork Subroutine" on page 287) subroutine, moncontrol ("moncontrol Subroutine" on page 840) subroutine, monitor ("monitor Subroutine" on page 841) subroutine, monstartup ("monstartup Subroutine" on page 847) subroutine.

The prof command.

**proj_execve Subroutine**

**Purpose**

Executes an application with the specified project assignment.
Library
The libaacct.a library.

Syntax

```c
int proj_execve(char *path, char *const arg[], char *const env[], projid_t projid, int force);
```

Description

The `proj_execve` system call assigns the requested project ID to the calling process and runs the given program. This subroutine checks whether the caller is allowed to assign the requested project ID to the application, using the available project assignment rules for the caller’s user ID, group ID, and application name. If the requested project assignment is not allowed, an error code is returned. However, the user with root authority or advanced accounting administrator capabilities can force the project assignment by setting the `force` parameter to 1.

Parameters

- `path` (Path for the application or program to be run.)
- `arg` (List of arguments for the new process.)
- `env` (Environment for the new process.)
- `projid` (Project ID to be assigned to the new process.)
- `force` (Option to override the allowed project list for the application, user, or group.)

Return Values

- `0` Upon success, does not return to the calling process.
- `-1` The subroutine failed.

Error Codes

- **EPERM** (Permission denied. A user without privileges attempted the call.)

Related Information

The “addproj Subroutine” on page 31, “chprojattr Subroutine” on page 158, “getproj Subroutine” on page 413.

Understanding the Advanced Accounting Subsystem

projdballoc Subroutine

Purpose

Allocates a project database handle.

Library

The libaacct.a library.
Syntax
<sys/aacct.h>

projballoc(void **handle)

Description
The projballoc subroutine allocates a handle to operate on the project database. By default, this handle is initialized to operate on the system project database; however, it can be reset with the projdbfinit subroutine to reference another project database.

Parameters

handle Pointer to a void pointer

Security
Only for privileged users. Privilege can be extended to nonroot users by granting the CAP_AACCT capability to a user.

Return Values

0 Success
-1 Failure

Error Codes

EINVAL The passed pointer is NULL
ENOMEM No space left on memory

Related Information
The "addprojdb Subroutine" on page 32, "chprojattrib Subroutine" on page 159, "getfirstprojdb Subroutine" on page 363, "getnextprojdb Subroutine" on page 391, "getprojdb Subroutine" on page 414, "projdbfinit Subroutine", "projdbfree Subroutine" on page 1160, "rmprojdb Subroutine"

projdbfinit Subroutine

Purpose
Sets the handle to use a local project database as specified in the dbfile pointer and opens the file with the specified mode.

Library
The libaacct.a library.

Syntax
<sys/aacct.h>

projdbfinit(void *handle, char *file, int mode)
Description
The projdbfinit subroutine sets the specified handle to use the specified project definition file. The file is opened in the specified mode. Subsequently, the project database, as represented by the handle parameter, will be referenced through file system primitives.

The project database must be initialized before calling this subroutine. The routines projdballoc and projdbfinit are provided for this purpose. The specified file is opened in the specified mode. File system calls are used to operate on these types of files. The struct projdb is filled as follows:
projdb.type = PROJ_LOCAL
projdb.fdes = value returned from open() call.

If the file parameter is NULL, then the system project database is opened.

Parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>handle</td>
<td>Pointer to handle</td>
</tr>
<tr>
<td>file</td>
<td>Indicate the project definition file name</td>
</tr>
<tr>
<td>mode</td>
<td>Indicates the mode in which the file is opened</td>
</tr>
</tbody>
</table>

Security
Only for privileged users. Privilege can be extended to nonroot users by granting the CAP_AACCT capability to a user.

Return Values

0          Success
-1         Failure

Error Codes

EINVAL    Passed handle or file is invalid

Related Information

The "addprojdb Subroutine" on page 32, "chprojattrdb Subroutine" on page 159, "getfirstprojdb Subroutine" on page 363, "getnextprojdb Subroutine" on page 391, "getproj Subroutine" on page 413, "getprojdb Subroutine" on page 414, "projdballoc Subroutine" on page 1158, "projdbfinit Subroutine" on page 1159, "projdbfree Subroutine," rmprojdb Subroutine

projdbfree Subroutine

Purpose
Frees an allocated project database handle.

Library
The libaacct.a library.
Syntax
<sys/aacct.h>

`projdbfree(void *handle)`

Description
The `projdbfree` subroutine releases the memory allocated to a project database handle. The closure operation is based on the type of project database. If a project database is local, then it is closed using system primitives. The project database must be initialized before calling this subroutine. The routines `projdballoc` and `projdbfinit` are provided for this purpose.

Parameters
`handle` Pointer to a void pointer

Security
Only for privileged users. Privilege can be extended to nonroot users by granting the CAP_AACCT capability to a user.

Return Values

<table>
<thead>
<tr>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Success</td>
</tr>
<tr>
<td>-1</td>
<td>Failure</td>
</tr>
</tbody>
</table>

Error Codes
EINVAL Passed pointer is NULL

Related Information
The `addprojdb Subroutine` on page 32, `chprojattrdb Subroutine` on page 159, `getfirstprojdb Subroutine` on page 363, `getnextprojdb Subroutine` on page 391, `getproj Subroutine` on page 413, `getprojdb Subroutine` on page 414, `projdballoc Subroutine` on page 1158, `projdbfinit Subroutine` on page 1159, `rmprojdb Subroutine`.

psdanger Subroutine

Purpose
Defines the amount of free paging space available.

Syntax
```
#include <signal.h>
#include <sys/vndinfo.h>

blkcmt_t psdanger (Signal);
int Signal;
```

Description
The `psdanger` subroutine returns the difference between the current number of free paging-space blocks and the paging-space thresholds of the system.
Parameters

*Signal*  
Defines the signal.

Return Values

If the value of the *Signal* parameter is 0, the return value is the total number of paging-space blocks defined in the system.

If the value of the *Signal* parameter is -1, the return value is the number of free paging-space blocks available in the system.

If the value of the *Signal* parameter is **SIGDANGER**, the return value is the difference between the current number of free paging-space blocks and the paging-space warning threshold. If the number of free paging-space blocks is less than the paging-space warning threshold, the return value is negative.

If the value of the *Signal* parameter is **SIGKILL**, the return value is the difference between the current number of free paging-space blocks and the paging-space kill threshold. If the number of free paging-space blocks is less than the paging-space kill threshold, the return value is negative.

Related Information

The `swapoff` subroutine, `swapon` subroutine, `swapqry` subroutine.

The `chps` command, `lsps` command, `mkps` command, `rmps` command, `swapoff` command, `swapon` command.

Paging space in *Operating system and device management*.

Subroutines Overview and Understanding Paging Space Programming Requirements in AIX 5L Version 5.3 General Programming Concepts: Writing and Debugging Programs.

**psignal Subroutine or sys_siglist Vector**

Purpose

Prints system signal messages.

Library

Standard C Library (*libc.a*)

Syntax

```c
psignal ( Signal, String )
unsigned Signal;
char *String;
char *sys_siglist[ ];
```

Description

The `psignal` subroutine produces a short message on the standard error file describing the indicated signal. First the *String* parameter is printed, then the name of the signal and a new-line character.

To simplify variant formatting of signal names, the `sys_siglist` vector of message strings is provided. The signal number can be used as an index in this table to get the signal name without the new-line character.
The **NSIG** defined in the **signal.h** file is the number of messages provided for in the table. It should be checked because new signals may be added to the system before they are added to the table.

**Parameters**

- **Signal** Specifies a signal. The signal number should be among those found in the **signal.h** file.
- **String** Specifies a string that is printed. Most usefully, the **String** parameter is the name of the program that incurred the signal.

**Related Information**

The **perror** subroutine, **sigvec** subroutine.

**pthdb_attr, pthdb_cond, pthdb_condattr, pthdb_key, pthdb_mutex, pthdb_mutexattr, pthdb_pthread, pthdb_pthread_key, pthdb_rwlock, or pthdb_rwlockattr Subroutine**

**Purpose**

Reports the pthread library objects.

**Library**

pthread debug library (**libpthdebug.a**)?

**Syntax**

```
#include <sys/pthdebug.h>

int pthdb_pthread (pthdb_session_t session, pthdb_pthread_t *pthreadp, int cmd);
int pthdb_pthread_key (pthdb_session_t session, pthread_key_t *keyp, int cmd);
int pthdb_attr(pthdb_session_t session, pthdb_attr_t *attrp, int cmd);
int pthdb_cond (pthdb_session_t session, pthdb_cond_t *condp, int cmd);
int pthdb_condattr (pthdb_session_t session, pthdb_condattr_t *condattrp, int cmd);
int pthdb_key(pthdb_session_t session, pthdb_pthread_t pthread, pthread_key_t *keyp, int cmd);
int pthdb_mutex (pthdb_session_t session, pthdb_mutex_t *mutexp, int cmd);
int pthdb_mutexattr (pthdb_session_t session, pthdb_mutexattr_t *mutexattrp, int cmd);
int pthdb_rwlock (pthdb_session_t session, pthdb_rwlock_t *rwlockp, int cmd);
```
int pthdb_rwlockattr (pthdb_session_t session,
    pthdb_rwlockattr_t *rwlockattrp,
    int cmd)

Description
The pthread library maintains internal lists of objects: pthreads, mutexes, mutex attributes, condition
variables, condition variable attributes, read/write locks, read/write lock attributes, attributes, pthread
specific keys, and active keys. The pthread debug library provides access to these lists one element at a
time via the functions listed above.

Each one of those functions acquire the next element in the list of objects. For example, the pthdb_attr
function gets the next attribute on the list of attributes.

A report of PTHDB_INVALID_OBJECT represents the empty list or the end of a list, where OBJECT is
equal to PTHREAD, ATTR, MUTEX, MUTEXATTR, COND, CONDATTR, RWLOCK, RWLOCKATTR,
KEY, or TID as appropriate.

Each list is reset to the top of the list when the pthdb_session_update function is called, or when the list
function reports a PTHDB_INVALID_* value. For example, when pthdb_attr reports an attribute of
PTHDB_INVALID_ATTR the list is reset to the beginning such that the next call reports the first attribute in
the list, if any.

When PTHDB_LIST_FIRST is passed for the cmd parameter, the first item in the list is retrieved.

Parameters

session    Session handle.
attrp      Attribute object.
cmd        Reset to the beginning of the list.
condp      Pointer to Condition variable object.
condattrp  Pointer to Condition variable attribute object.
keyp       Pointer to Key object.
mutexattrp Pointer to Mutex attribute object.
mutexp     Pointer to Mutex object.
pthreadp   pthread object.
rwlockp    Pointer to Read/Write lock object.
rwlockattrp Pointer to Read/Write lock attribute object.

Return Values
If successful, these functions return PTHDB_SUCCESS. Otherwise, an error code is returned.

Error Codes

PTHDB_BAD_SESSION     Invalid session handle.
PTHDB_BAD_PTHREAD     Invalid pthread handle.
PTHDB_BAD_CMD         Invalid command.
PTHDB_BAD_POINTER     Invalid buffer pointer.
PTHDB_INTERNAL        Error in library.
PTHDB_MEMORY          Not enough memory
Related Information

The `pthdebug.h` file.

The `pthread.h` file.

`pthdb_attr_detachstate`, `pthdb_attr_addr`, `pthdb_attr_guardsize`, `pthdb_attr Inheritsched`, `pthdb_attr_schedparam`, `pthdb_attr_schedpolicy`, `pthdb_attr_schedpriority`, `pthdb_attr_scope`, `pthdb_attr_stackaddr`, `pthdb_attr_stacksize`, or `pthdb_attr_suspendstate` subroutine

Purpose

Query the various fields of a pthread attribute and return the results in the specified buffer.

Library

pthread debug library (`libpthdebug.a`)

Syntax

```c
#include <sys/pthdebug.h>

int pthdb_attr_detachstate (pthdb_session_t session, pthdb_attr_t attr, pthdb_detachstate_t *detachstatep);
int pthdb_attr_addr (pthdb_session_t session, pthdb_attr_t attr, pthdb_addr_t *addrp);
int pthdb_attr_guardsize (pthdb_session_t session, pthdb_attr_t attr, pthdb_size_t *guardsizep);
int pthdb_attr Inheritsched (pthdb_session_t session, pthdb_attr_t attr, pthdb_inheritsched_t *inheritschedp);
int pthdb_attr_schedparam (pthdb_session_t session, pthdb_attr_t attr, struct sched_param *schedparamp);
int pthdb_attr_schedpolicy (pthdb_session_t session, pthdb_attr_t attr, pthdb_policy_t *schedpolicyp);
int pthdb_attr_schedpriority (pthdb_session_t session, pthdb_attr_t attr, int *schedpriority);
int pthdb_attr_scope (pthdb_session_t session, pthdb_attr_t attr, pthdb_scope_t *scopep);
int pthdb_attr_stackaddr (pthdb_session_t session, pthdb_attr_t attr, pthdb_size_t *stackaddrp);
int pthdb_attr_stacksize (pthdb_session_t session, pthdb_attr_t attr, pthdb_size_t *stacksizep);
int pthdb_attr_suspendstate (pthdb_session_t session, pthdb_attr_t attr, pthdb_suspendstate_t *suspendstatep);
```
Description
Each pthread is created using either the default pthread attribute or a user-specified pthread attribute. These functions query the various fields of a pthread attribute and, if successful, return the result in the buffer specified. In all cases, the values returned reflect the expected fields of a pthread created with the attribute specified.


**pthdb_attr_detachstate** reports if the created pthread is detachable (PDS_DETACHED) or joinable (PDS_JOINABLE). PDS_NOTSUP is reserved for unexpected results.

**pthdb_attr_addr** reports the address of the pthread_attr_t.

**pthdb_attr_guardsize** reports the guard size for the attribute.

**pthdb_attr_inheritsched** reports whether the created pthread will run with scheduling policy and scheduling parameters from the created pthread (PIS_INHERIT), or from the attribute (PIS_EXPLICIT). PIS_NOTSUP is reserved for unexpected results.

**pthdb_attr_schedparam** reports the scheduling parameters associated with the pthread attribute. See **pthdb_attr_inheritsched** for additional information.

**pthdb_attr_schedpolicy** reports whether the scheduling policy associated with the pthread attribute is other (SP_OTHER), first in first out (SP_FIFO), or round robin (SP_RR). SP_NOTSUP is reserved for unexpected results.

**pthdb_attr_schedpriority** reports the scheduling priority associated with the pthread attribute. See **pthdb_attr_inheritsched** for additional information.

**pthdb_attr_scope** reports whether the created pthread will have process scope (PS_PROCESS) or system scope (PS_SYSTEM). PS_NOTSUP is reserved for unexpected results.

**pthdb_attr_stackaddr** reports the address of the stack.

**pthdb_attr_stacksize** reports the size of the stack.

**pthdb_attr_suspendstate** reports whether the created pthread will be suspended (PSS_SUSPENDED) or not (PSS_UNSUSPENDED). PSS_NOTSUP is reserved for unexpected results.

Parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>addr</strong></td>
<td>Attributes address.</td>
</tr>
<tr>
<td><strong>attr</strong></td>
<td>Attributes handle.</td>
</tr>
<tr>
<td><strong>detachstatep</strong></td>
<td>Detach state buffer.</td>
</tr>
<tr>
<td><strong>guardsizep</strong></td>
<td>Attribute guard size.</td>
</tr>
<tr>
<td><strong>inheritschedp</strong></td>
<td>Inherit scheduling buffer.</td>
</tr>
<tr>
<td><strong>schedparamp</strong></td>
<td>Scheduling parameters buffer.</td>
</tr>
<tr>
<td><strong>schedpolicyp</strong></td>
<td>Scheduling policy buffer.</td>
</tr>
<tr>
<td><strong>schedpriorityp</strong></td>
<td>Scheduling priority buffer.</td>
</tr>
<tr>
<td><strong>scopep</strong></td>
<td>Contention scope buffer.</td>
</tr>
<tr>
<td><strong>session</strong></td>
<td>Session handle.</td>
</tr>
<tr>
<td><strong>stackaddrp</strong></td>
<td>Attributes stack address.</td>
</tr>
<tr>
<td><strong>stacksizep</strong></td>
<td>Attributes stack size.</td>
</tr>
<tr>
<td><strong>suspendstatep</strong></td>
<td>Suspend state buffer.</td>
</tr>
</tbody>
</table>
Return Values
If successful these functions return PTHDB_SUCCESS. Otherwise, an error code is returned.

Error Codes

<table>
<thead>
<tr>
<th>Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>PTHDB_BAD_SESSION</td>
<td>Invalid session handle.</td>
</tr>
<tr>
<td>PTHDB_BAD_ATTR</td>
<td>Invalid attribute handle.</td>
</tr>
<tr>
<td>PTHDB_BAD_POINTER</td>
<td>Invalid buffer pointer.</td>
</tr>
<tr>
<td>PTHDB_CALLBACK</td>
<td>Debugger call back error.</td>
</tr>
<tr>
<td>PTHDB_NOTSUP</td>
<td>Not supported.</td>
</tr>
<tr>
<td>PTHDB_INTERNAL</td>
<td>Internal library error.</td>
</tr>
</tbody>
</table>

Related Information
The pthdebug.h file.
The pthread.h file.

pthdb_condattr_pshared, or pthdb_condattr_addr Subroutine

Purpose
Gets the condition variable attribute pshared value.

Library
pthread debug library (libpthdebug.a)

Syntax

```c
#include <sys/pthdebug.h>

int pthdb_condattr_pshared (pthdb_session_t session, pthdb_condattr_t condattr, pthdb_pshared_t *psharedp);

int pthdb_condattr_addr (pthdb_session_t session, pthdb_condattr_t condattr, pthdb_addr_t *addrp);
```

Description
The **pthdb_condattr_pshared** function is used to get the condition variable attribute process shared value. The pshared value can be PSH_SHARED, PSH_PRIVATE, or PSH_NOTSUP.

The **pthdb_condattr_addr** function reports the address of the pthread_condattr_t.

Parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>addrp</td>
<td>Pointer to the address of the pthread_condattr_t.</td>
</tr>
<tr>
<td>condattr</td>
<td>Condition variable attribute handle</td>
</tr>
<tr>
<td>psharedp</td>
<td>Pointer to the pshared value.</td>
</tr>
<tr>
<td>session</td>
<td>Session handle.</td>
</tr>
</tbody>
</table>
Return Values
If successful this function returns PTHDB_SUCCESS. Otherwise, an error code is returned.

Error Codes

<table>
<thead>
<tr>
<th>Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>PTHDB_BAD_CONDATTR</td>
<td>Invalid condition variable attribute handle.</td>
</tr>
<tr>
<td>PTHDB_BAD_SESSION</td>
<td>Invalid session handle.</td>
</tr>
<tr>
<td>PTHDB_CALLBACK</td>
<td>Debugger call back error.</td>
</tr>
<tr>
<td>PTHDB_INTERNAL</td>
<td>Error in library.</td>
</tr>
<tr>
<td>PTHDB_POINTER</td>
<td>Invalid pointer</td>
</tr>
</tbody>
</table>

Error Codes

Related Information
The pthdebug.h file.
The pthread.h file.

**pthdb_cond_addr, pthdb_cond_mutex or pthdb_cond_pshared**

### Subroutine

### Purpose
Gets the condition variable's mutex handle and pshared value.

### Library
pthread debug library (libpthdebug.a)

### Syntax
```
#include <sys/pthdebug.h>

int pthdb_cond_addr (pthdb_session_t session, pthdb_cond_t cond, pthdb_addr_t *addrp);
int pthdb_cond_mutex (pthdb_session_t session, pthdb_cond_t cond, pthdb_mutex_t *mutexp);
int pthdb_cond_pshared (pthdb_session_t session, pthdb_cond_t cond, pthdb_pshared_t *psharedp);
```

### Description
The pthdb_cond_addr function reports the address of the pthdb_cond_t.

The pthdb_cond_mutex function is used to get the mutex handle associated with the particular condition variable, if the mutex does not exist then PTHDB_INVALID_MUTEX is returned from the mutex.

The pthdb_cond_pshared function is used to get the condition variable process shared value. The pshared value can be PSH_SHARED, PSH_PRIVATE, or PSH_NOTSUP.
Parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>addr</td>
<td>Condition variable address</td>
</tr>
<tr>
<td>cond</td>
<td>Condition variable handle</td>
</tr>
<tr>
<td>mutexp</td>
<td>Pointer to mutex</td>
</tr>
<tr>
<td>psharedp</td>
<td>Pointer to pshared value</td>
</tr>
<tr>
<td>session</td>
<td>Session handle</td>
</tr>
</tbody>
</table>

Return Values

If successful, these functions return `PTHDB_SUCCESS`. Otherwise, an error code is returned.

Error Codes

- `PTHDB_BAD_COND`: Invalid cond handle.
- `PTHDB_BAD_SESSION`: Invalid session handle.
- `PTHDB_CALLBACK`: Debugger call back error.
- `PTHDB_INVALID_MUTEX`: Invalid mutex.
- `PTHDB_INTERNAL`: Error in library.
- `PTHDB_POINTER`: Invalid pointer

Related Information

The `pthdebug.h` file.

The `pthread.h` file.

`pthdb_mutexattr_addr`, `pthdb_mutexattr_prioceiling`, `pthdb_mutexattr_protocol`, `pthdb_mutexattr_pshared` or `pthdb_mutexattr_type`

Subroutine

Purpose

Gets the mutex attribute pshared, priority ceiling, protocol, and type values.

Library

pthread debug library (`libpthdebug.a`)

Syntax

```c
#include <sys/pthdebug.h>

int pthdb_mutexattr_addr (pthdb_session_t session, pthdb_mutexattr_t mutexattr, pthdb_addr_t *addrp);

int pthdb_mutexattr_protocol (pthdb_session_t session, pthdb_mutexattr_t mutexattr, pthdb_protocol_t *protocolp);

int pthdb_mutexattr_pshared (pthdb_session_t session, pthdb_mutexattr_t mutexattr, pthdb_pshared_t *psharedp);
```
int pthdb_mutexattr_type (pthdb_session_t session,
                        pthdb_mutexattr_t mutexattr,
                        pthdb_mutex_type_t *typep)

Description
The pthdb_mutexattr_addr function reports the address of the pthread_mutexattr_t.

The pthdb_mutexattr_prioceiling function is used to get the mutex attribute priority ceiling value.

The pthdb_mutexattr_protocol function is used to get the mutex attribute protocol value. The protocol value can be MP_INHERIT, MP_PROTECT, MP_NONE, or MP_NOTSUP.

The pthdb_mutexattr_pshared function is used to get the mutex attribute process shared value. The pshared value can be PSH_SHARED, PSH_PRIVATE, or PSH_NOTSUP.

The pthdb_mutexattr_type is used to get the value of the mutex attribute type. The values for the mutex type can be MK_NONRECURSIVE_NP, MK_RECURSIVE_NP, MK_FAST_NP, MK_ERRORCHECK, MK_RECURSIVE, MK_NORMAL, or MK_NOTSUP.

Parameters
addr Mutex attribute address.
mutexattr Condition variable attribute handle
prioceiling Pointer to priority ceiling value.
protocolp Pointer to protocol value.
psharedp Pointer to pshared value.
session Session handle.
typep Pointer to type value.

Return Values
If successful, these functions return PTHDB_SUCCESS. Otherwise, an error code is returned.

Error Codes
PTHDB_BAD_MUTEXATTR Invalid mutex attribute handle.
PTHDB_BAD_SESSION Invalid session handle.
PTHDB_CALLBACK Debugger call back error.
PTHDB_INTERNAL Error in library.
PTHDB_NOSYS Not implemented
PTHDB_POINTER Invalid pointer

Related Information
The pthdebug.h file.
The pthread.h file.
**Purpose**
Gets the owner's pthread, mutex's pshared value, priority ceiling, protocol, lock state, and type.

**Library**
pthread debug library (libpthreaddebug.a)

**Syntax**
```
#include <sys/pthdebug.h>

int pthdb_mutex_addr (pthdb_session_t session, pthdb_mutex_t mutex, pthdb_addr_t *addrp);

int pthdb_mutex_owner (pthdb_session_t session, pthdb_mutex_t mutex, pthdb_pthread_t *ownerp);

int pthdb_mutex_lock_count (pthdb_session_t session, pthdb_mutex_t mutex, int *countp);

int pthdb_mutex_pshared (pthdb_session_t session, pthdb_mutex_t mutex, pthdb_pshared_t *psharedp);

int pthdb_mutex_prioceiling (pthdb_session_t session, pthdb_mutex_t mutex, pthdb_pshared_t *prioceilingp);

int pthdb_mutex_protocol (pthdb_session_t session, pthdb_mutex_t mutex, pthdb_pshared_t *protocolp);

int pthdb_mutex_state (pthdb_session_t session, pthdb_mutex_t mutex, pthdb_mutex_state_t *statep);

int pthdb_mutex_type (pthdb_session_t session, pthdb_mutex_t mutex, pthdb_mutex_type_t *typep);
```

**Description**
- **pthdb_mutex_addr** reports the address of the pthread_mutex_t.
- **pthdb_mutex_lock_count** reports the lock count of the mutex.
- **pthdb_mutex_owner** is used to get the pthread that owns the mutex.
The `pthdb_mutex_pshared` function is used to get the mutex process shared value. The pshared value can be `PSH_SHARED`, `PSH_PRIVATE`, or `PSH_NOTSUP`.

`pthdb_mutex_prioceiling` function is used to get the mutex priority ceiling value.

`pthdb_mutex_protocol` function is used to get the mutex protocol value. The protocol value can be `MP_INHERIT`, `MP_PROTECT`, `MP_NONE`, or `MP_NOTSUP`.

`pthdb_mutex_state` is used to get the value of the mutex lock state. The state can be `MS_LOCKED`, `MS_UNLOCKED` or `MS_NOTSUP`.

`pthdb_mutex_type` is used to get the value of the mutex type. The values for the mutex type can be `MK_NONRECURSIVE_NP`, `MK_RECURSIVE_NP`, `MK_FAST_NP`, `MK_ERRORCHECK`, `MK_RECURSIVE`, `MK_NORMAL`, or `MK_NOTSUP`.

**Parameters**

- `addr` Mutex address
- `countp` Mutex lock count
- `mutex` Mutex handle
- `ownerp` Pointer to mutex owner
- `psharedp` Pointer to pshared value
- `prioceilingp` Pointer to priority ceiling value
- `protocolp` Pointer to protocol value
- `session` Session handle.
- `statep` Pointer to mutex state
- `typep` Pointer to mutex type

**Return Values**

If successful, these functions return `PTHDB_SUCCESS`. Otherwise, an error code is returned.

**Error Codes**

- `PTHDB_BAD_MUTEX` Invalid mutex handle.
- `PTHDB_BAD_SESSION` Invalid session handle.
- `PTHDB_CALLBACK` Debugger call back error.
- `PTHDB_INTERNAL` Call failed.
- `PTHDB_NOSYS` Not implemented
- `PTHDB_POINTER` Invalid pointer

**Related Information**

The `pthdebug.h` file and the `pthread.h` file.

The `pthread.h` file.

**pthdb_mutex_waiter, pthdb_cond_waiter, pthdb_rwlock_read_waiter or pthdb_rwlock_write_waiter Subroutine**

**Purpose**

Gets the next waiter in the list of an object's waiters.
Library

pthread debug library (libpthdebug.a)

Syntax

```c
#include <sys/pthdebug.h>

int pthdb_mutex_waiter (pthdb_session_t session, pthdb_mutex_t mutex, pthdb_pthread_t *waiter, int cmd);

int pthdb_cond_waiter (pthdb_session_t session, pthdb_cond_t cond, pthdb_pthread_t *waiter, int cmd);

int *pthdb_rwlock_read_waiter (pthdb_session_t session, pthdb_rwlock_t rwlock, pthdb_pthread_t *waiter, int cmd);

int *pthdb_rwlock_write_waiter (pthdb_session_t session, pthdb_rwlock_t rwlock, pthdb_pthread_t *waiter, int cmd);
```

Description

The `pthdb_mutex_waiter` functions get the next waiter in the list of an object’s waiters.

Each list is reset to the top of the list when the `pthdb_session_update` function is called, or when the list function reports a `PTHDB_INVALID_`* value. For example, when `pthdb_attr` reports an attribute of `PTHDB_INVALID_ATTR` the list is reset to the beginning such that the next call reports the first attribute in the list, if any.

A report of `PTHDB_INVALID_OBJECT` represents the empty list or the end of a list, where `OBJECT` is one of these values: `PTHREAD`, `ATTR`, `MUTEX`, `MUTEXATTR`, `COND`, `CONDATTR`, `RWLOCK`, `RWLOCKATTR`, `KEY`, or `TID` as appropriate.

When `PTHDB_LIST_FIRST` is passed for the `cmd` parameter, the first item in the list is retrieved.

Parameters

- `session`: Session handle.
- `mutex`: Mutex object.
- `cond`: Condition variable object.
- `cmd`: Reset to the beginning of the list.
- `rwlock`: Read/Write lock object.
- `waiter`: Pointer to waiter.

Return Values

If successful, these functions return `PTHDB_SUCCESS`. Otherwise, an error code is returned.

Error Codes

- `PTHDB_BAD_SESSION`: Invalid session handle.
- `PTHDB_BAD_CMD`: Invalid command.
Related Information
The `pthdebug.h` file.

The `pthread.h` file.

**pthdb_pthread_arg Subroutine**

**Purpose**
Reports the information associated with a pthread.

**Library**
pthread debug library (`libpthdebug.a`)

**Syntax**

```c
#include <sys/pthdebug.h>

int pthdb_pthread_arg (pthdb_session_t session, pthdb_pthread_t pthread, pthdb_addr_t *argp)

int pthdb_pthread_addr (pthdb_session_t session, pthdb_pthread_t pthread, pthdb_addr_t *addrp)

int pthdb_pthread_cancelpend (pthdb_session_t session, pthdb_pthread_t pthread, int *cancelpendp)

int pthdb_pthread_cancelstate (pthdb_session_t session, pthdb_pthread_t pthread, pthdb_cancelstate_t *cancelstatep)

int pthdb_pthread_canceltype (pthdb_session_t session, pthdb_pthread_t pthread, pthdb_canceltype_t *canceltypep)

int pthdb_pthread_detachstate (pthdb_session_t session, pthdb_pthread_t pthread, pthdb_detachstate_t *detachstatep)

int pthdb_pthread_exit (pthdb_session_t session, pthdb_pthread_t pthread, pthdb_addr_t *exitp)

int pthdb_pthread_func (pthdb_session_t session, pthdb_pthread_t pthread, pthdb_addr_t *funcp)
```

1174  Technical Reference, Volume 1: Base Operating System and Extensions
int pthdb_pthread_ptid (pthdb_session_t session, pthdb_pthread_t pthread, pthread_t *ptidp);

int pthdb_pthread_schedparam (pthdb_session_t session, pthdb_pthread_t pthread, struct sched_param *schedparamp);

int pthdb_pthread_schedpolicy (pthdb_session_t session, pthdb_pthread_t pthread, pthdb_schedpolicy_t *schedpolicyp);

int pthdb_pthread_schedpriority (pthdb_session_t session, pthdb_pthread_t pthread, int *schedpriorityp);

int pthdb_pthread_scope (pthdb_session_t session, pthdb_pthread_t pthread, pthdb_scope_t *scopep);

int pthdb_pthread_state (pthdb_session_t session, pthdb_pthread_t pthread, pthdb_state_t *statep);

int pthdb_pthread_suspendstate (pthdb_session_t session, pthdb_pthread_t pthread, pthdb_suspendstate_t *suspendstatep);

int pthdb_ptid_pthread (pthdb_session_t session, pthread_t ptid, pthdb_pthread_t *pthreadp);

Description

pthdb_pthread_arg reports the initial argument passed to the pthread's start function.

pthdb_pthread_addr reports the address of the pthread_t.

pthdb_pthread_cancelpend reports non-zero if cancellation is pending on the pthread; if not, it reports zero.

pthdb_pthread_cancelstate reports whether cancellation is enabled (PCS_ENABLE) or disabled (PCS_DISABLE). PCS_NOTSUP is reserved for unexpected results.

pthdb_pthread_canceltype reports whether cancellation is deferred (PCT_DEFERRED) or asynchronous (PCTASYNCHRONOUS). PCT_NOTSUP is reserved for unexpected results.

pthdb_pthread_detachstate reports whether the pthread is detached (PDS_DETACHED) or joinable (PDS_JOINABLE). PDS_NOTSUP is reserved for unexpected results.

pthdb_pthread_exit reports the exit status returned by the pthread via pthread_exit. This is only valid if the pthread has exited (PST_TERM).

pthdb_pthread_func reports the address of the pthread's start function.

pthdb_pthread_ptid reports the pthread identifier (pthread_t) associated with the pthread.
**pthdb_pthread_schedparam** reports the pthread’s scheduling parameters. This currently includes policy and priority.

**pthdb_pthread_schedpolicy** reports whether the pthread’s scheduling policy is other (**SP_OTHER**), first in first out (**SP_FIFO**), or round robin (**SP_RR**). **SP_NOTSUP** is reserved for unexpected results.

**pthdb_pthread_schedpriority** reports the pthread’s scheduling priority.

**pthdb_pthread_scope** reports whether the pthread has process scope (**PS_PROCESS**) or system scope (**PS_SYSTEM**). **PS_NOTSUP** is reserved for unexpected results.

**pthdb_pthread_state** reports whether the pthread is being created (**PST_IDLE**), currently running (**PST_RUN**), waiting on an event (**PST_SLEEP**), waiting on a cpu (**PST_READY**), or waiting on a join or detach (**PST_TERM**). **PST_NOTSUP** is reserved for unexpected results.

**pthdb_pthread_suspendstate** reports whether the pthread is suspended (**PSS_SUSPENDED**) or not (**PSS_UNSUSPENDED**). **PSS_NOTSUP** is reserved for unexpected results.

**pthdb_ptid_pthread** reports the pthread for the ptid.

**Parameters**

- **addr**: pthread address
- **argp**: Initial argument buffer.
- **cancelpendp**: Cancel pending buffer.
- **cancelstatep**: Cancel state buffer.
- **canceltypep**: Cancel type buffer.
- **detachstatep**: Detach state buffer.
- **exitp**: Exit value buffer.
- **funcp**: Start function buffer.
- **pthread**: pthread handle.
- **pthreadp**: Pointer to pthread handle.
- **ptid**: pthread identifier.
- **ptidp**: pthread identifier buffer.
- **schedparamp**: Scheduling parameters buffer.
- **schedpolicyp**: Scheduling policy buffer.
- **schedpriorityp**: Scheduling priority buffer.
- **scopep**: Contention scope buffer.
- **session**: Session handle.
- **statep**: State buffer.
- **suspendstatep**: Suspend state buffer.

**Return Values**

If successful, these functions return **PTHDB_SUCCESS**, else an error code is returned.

**Error Codes**

- **PTHDB_BAD_SESSION**: Invalid session handle.
- **PTHDB_BAD_PTHREAD**: Invalid pthread handle.
- **PTHDB_BAD_POINTER**: Invalid buffer pointer.
- **PTHDB_BAD_PTID**: Invalid ptid.
- **PTHDB_CALLBACK**: Debugger call back error.
- **PTHDB_NOTSUP**: Not supported.
- **PTHDB_INTERNAL**: Error in library.
Related Information
The pthdebug.h file.
The pthread.h file.

pthdb_pthread_context or pthdb_pthread_setcontext Subroutine

Purpose
Provides access to the pthread context via the struct context64 structure.

Library
pthread debug library (libpthdebug.a)

Syntax
#include <sys/pthdebug.h>

```c
int pthdb_pthread_context (pthdb_session_t session,
                         pthdb_pthread_t pthread,
                         pthdb_context_t *context)

int pthdb_pthread_setcontext (pthdb_session_t session,
                           pthdb_pthread_t pthread,
                           pthdb_context_t *context)
```

Description
The pthread debug library provides access to the pthread context via the struct context64 structure, whether the process is 32-bit or 64-bit. The debugger should be able to convert from 32-bit to 64-bit and from 64-bit for 32-bit processes. The extent to which this structure is filled in depends on the presence of the PTHDB_FLAG_GPRS, PTHDB_FLAG_SPRS, and PTHDB_FLAG_FPRS session flags. It is necessary to use the pthread debug library to access the context of a pthread without a kernel thread. The pthread debug library can also be used to access the context of a pthread with a kernel thread, but this results in a call back to the debugger, meaning that the debugger is capable of obtaining this information by itself. The debugger determines if the kernel thread is running in user mode or kernel mode and then fills in the struct context64 appropriately. The pthread debug library does not use this information itself and is thus not sensitive to the correct implementation of the read_regs and write_regs call back functions.

```c
pthdb_pthread_context reports the context of the pthread based on the settings of the session flags. Uses the read_regs call back if the pthread has a kernel thread. If read_regs is not defined, then it returns PTHDB_NOTSUP.
```

```c
pthdb_pthread_setcontext sets the context of the pthread based on the settings of the session flags. Uses the write_data call back if the pthread does not have a kernel thread. Use the write_regs call back if the pthread has a kernel thread.
```

If the debugger does not define the read_regs and write_regs call backs and if the pthread does not have a kernel thread, then the pthdb_pthread_context and pthdb_pthread_setcontext functions succeed. But if a pthread does not have a kernel thread, then these functions fail and return PTHDB_CONTEXT.

Parameters

session    Session handle.
pthread handle.
context    Context buffer pointer.

Return Values
If successful, these functions return PTHDB_SUCCESS. Otherwise, an error code is returned.

Error Codes

PTHDB_BAD_SESSION     Invalid session handle.
PTHDB_BAD_PTHREAD     Invalid pthread handle.
PTHDB_BAD_POINTER     Invalid buffer pointer.
PTHDB_CALLBACK        Callback function failed.
PTHDB_CONTEXT         Could not determine pthread context.
PTHDB_MEMORY          Not enough memory

pthdb_pthread_(set)context returns PTHDB_NOTSUP if the read_regs, write_data or write_regs call backs are set to NULL.
PTHDB_INTERNAL Error in library.

Related Information
The pthdebug.h file.
The pthread.h file.

pthdb_pthread_hold, pthdb_pthread_holdstate or pthdb_pthread_unhold Subroutine

Purpose
Reports and changes the hold state of the specified pthread.

Library
pthread debug library (libpthdebug.a)

Syntax
#include <sys/pthdebug.h>

int pthdb_pthread_holdstate (pthdb_session_t session, pthdb_pthread_t pthread, pthdb_holdstate_t *holdstatep)
int pthdb_pthread_hold (pthdb_session_t session, pthdb_pthread_t pthread)
int pthdb_pthread_unhold (pthdb_session_t session, pthdb_pthread_t pthread)

Description
pthdb_pthread_holdstate reports if a pthread is held. The possible hold states are PHS_HELD, PHS_NOTHELD, or PHS_NOTSUP.

pthdb_pthread_hold prevents the specified pthread from running.
**pthdb_pthread_unhold** unholds the specified pthread. The pthread held earlier can be unheld by calling this function.

**Notes:**
1. You must always use the **pthdb_pthread_hold** and **pthdb_pthread_unhold** functions, regardless of whether or not a pthread has a kernel thread.
2. These functions are only supposted when the **PTHDB_FLAG_HOLD** is set.

**Parameters**

- **session**
  
  Session handle.

- **pthread**
  
  pthread handle. The specified pthread should have an attached kernel thread id.

- **holdstatep**
  
  Pointer to the hold state

**Return Values**

If successful, **pthdb_pthread_hold** returns **PTHDB_SUCCESS**. Otherwise, it returns an error code.

**Error Codes**

- **PTHDB_BAD_PTHREAD**
  
  Invalid pthread handle.

- **PTHDB_BAD_SESSION**
  
  Invalid session handle.

- **PTHDB_HELD**
  
  pthread is held.

- **PTHDB_INTERNAL**
  
  Error in library.

**Related Information**

The **pthdb_session_setflags** subroutine.

The **pthdebug.h** file.

The **pthread.h** file.

---

**pthdb_pthread_sigmask, pthdb_pthread_sigpend or pthdb_pthread_sigwait Subroutine**

**Purpose**

Returns the pthread signals pending, the signals blocked, the signals received, and awaited signals.

**Library**

pthread debug library (**libpthdebug.a**)

**Syntax**

```c
#include <sys/pthdebug.h>

int pthdb_pthread_sigmask (pthdb_session_t session, pthdb_pthread_t pthread, sigset_t *sigsetp);
int pthdb_pthread_sigpend (pthdb_session_t session, pthdb_pthread_t pthread, sigset_t *sigsetp);
int pthdb_pthread_sigwait (pthdb_session_t session, sigset_t *sigsetp);
```
Description

`pthdb_pthread_sigmask` reports the signals that the pthread has blocked.

`pthdb_pthread_sigpend` reports the signals that the pthread has pending.

`pthdb_pthread_sigwait` reports the signals that the pthread is waiting on.

Parameters

- `session`  
  Session handle.

- `pthread`  
  Pthread handle

- `sigsetp`  
  Signal set buffer.

Return Values

If successful, these functions return `PTHDB_SUCCESS`. Otherwise, an error code is returned.

Error Code

- `PTHDB_BAD_SESSION`  
  Invalid session handle.

- `PTHDB_BAD_PTHREAD`  
  Invalid pthread handle.

- `PTHDB_BAD_POINTER`  
  Invalid buffer pointer.

- `PTHDB_CALLBACK`  
  Debugger call back error.

- `PTHDB_INTERNAL`  
  Error in library.

Related Information

The `pthdebug.h` file.

The `pthread.h` file.

`pthdb_pthread_specific` Subroutine

Purpose

Reports the value associated with a pthreads specific data key.

Library

`pthdebug` debug library (`libpthdebug.a`)  

Syntax

```
#include <sys/pthdebug.h>

void *pthdb_pthread_specific(pthdb_session_t session, pthdb_pthread_t pthread, pthdb_key_t key, pthdb_addr_t *specificp)
```
Description
Each process has active pthread specific data keys. Each active pthread specific data key is in use by one or more pthreads. Each pthread can have its own value associated with each pthread specific data key. The `pthdb_pthread_specific` function provide access to those values.

`pthdb_pthread_specific` reports the specific data value for the pthread and key combination.

Parameters

- `session`: The session handle.
- `pthread`: Thepthread handle.
- `key`: The key.
- `specificp`: Specific data value buffer.

Return Values
If successful, `pthdb_pthread_specific` returns `PTHDB_SUCCESS`. Otherwise, an error code is returned.

Error Codes

- `PTHDB_BAD_SESSION`: Invalid session handle.
- `PTHDB_BAD_PTHREAD`: Invalid pthread handle.
- `PTHDB_BAD_KEY`: Invalid key.
- `PTHDB_BAD_POINTER`: Invalid buffer pointer.
- `PTHDB_CALLBACK`: Debugger call back error.
- `PTHDB_INTERNAL`: Error in library.

Related information
The `pthdebug.h` file.
The `pthread.h` file.

`pthdb_pthread_tid` or `pthdb_tid_pthread` Subroutine

Purpose
Gets the kernel thread associated with the pthread and the pthread associated with the kernel thread.

Library
pthread debug library (libpthdebug.a)

Syntax

```
#include <sys/pthdebug.h>

int pthdb_pthread_tid (pthdb_session_t session, pthdb_pthread_t pthread, tid_t *tidp);
int pthdb_tid_pthread (pthdb_session_t session, tid_t tid, pthdb_pthread_t *pthreadp);
```
Description

`pthdb_pthread_tid` gets the kernel thread id associated with the pthread.

`pthdb_tid_pthread` is used to get the pthread associated with the kernel thread.

Parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>session</td>
<td>Session handle.</td>
</tr>
<tr>
<td>pthread</td>
<td>Pthread handle.</td>
</tr>
<tr>
<td>pthreadp</td>
<td>Pointer to pthread handle</td>
</tr>
<tr>
<td>tid</td>
<td>Kernel thread id</td>
</tr>
<tr>
<td>tidp</td>
<td>Pointer to kernel thread id</td>
</tr>
</tbody>
</table>

Return Values

If successful, these functions return `PTHDB_SUCCESS`. Otherwise, an error code is returned.

Error Codes

- `PTHDB_BAD_PTHREAD`: Invalid pthread handle.
- `PTHDB_BAD_SESSION`: Invalid session handle.
- `PTHDB_BAD_TID`: Invalid tid.
- `PTHDB_CALLBACK`: Debugger call back error.
- `PTHDB_INTERNAL`: Error in library.
- `PTHDB_INVALID_TID`: Empty list or the end of a list.

Related Information

The `pthdebug.h` file.

The `pthread.h` file.

`pthdb_rwlockattr_addr`, or `pthdb_rwlockattr_pshared` Subroutine

Purpose

Gets the rwlock attribute pshared values.

Library

pthread debug library (libpthdebug.a)

Syntax

```c
#include <sys/pthdebug.h>

int pthdb_rwlockattr_addr (pthdb_session_t session, pthdb_rwlockattr_t rwlockattr, pthdb_addr_t *addrp)

int pthdb_rwlockattr_pshared (pthdb_session_t session, pthdb_rwlockattr_t rwlockattr, pthdb_pshared_t *psharedp)
```
Description

`pthdb_rwlockattr_addr` reports the address of the `pthread_rwlockattr_t`.

`pthdb_rwlockattr_pshared` is used to get the rwlock attribute process shared value. The pshared value can be `PSH_SHARED`, `PSH_PRIVATE`, or `PSH_NOTSUP`.

Parameters

- `addr` : Read/Write lock attribute address.
- `psharedp` : Pointer to the pshared value.
- `rwlockattr` : Read/Write lock attribute handle
- `session` : Session handle.

Return Values

If successful, these functions return `PTHDB_SUCCESS`. Otherwise, an error code is returned.

Error Codes

- `PTHDB_BAD_RWLOCKATTR` : Invalid rwlock attribute handle.
- `PTHDB_BAD_SESSION` : Invalid session handle.
- `PTHDB_CALLBACK` : Debugger call back error.
- `PTHDB_INTERNAL` : Error in library.
- `PTHDB_POINTER` : Invalid pointer

Related Information

The `pthdebug.h` file.
The `pthread.h` file.

`pthdb_rwlock_addr`, `pthdb_rwlock_lock_count`, `pthdb_rwlock_owner`, `pthdb_rwlock_pshared` or `pthdb_rwlock_state` Subroutine

Purpose

Gets the owner, the pshared value, or the state of the read/write lock.

Library

pthread debug library (`libpthdebug.a`)

Syntax

```c
#include <sys/pthdebug.h>

int pthdb_rwlock_addr (pthdb_session_t session, pthdb_rwlock_t rwlock, pthdb_addr_t *addrp);

int pthdb_rwlock_lock_count (pthdb_session_t session, pthdb_rwlock_t rwlock, int *countp);
```
int pthdb_rwlock_owner (pthdb_session_t session, pthdb_rwlock_t rwlock, pthdb_pthread_t *ownerp, int cmd)

int pthdb_rwlock_pshared (pthdb_session_t session, pthdb_rwlock_t rwlock, pthdb_pshared_t *psharedp)

int pthdb_rwlock_state (pthdb_session_t session, pthdb_rwlock_t rwlock, pthdb_rwlock_state_t *statep)

Description
The pthdb_rwlock_addr function reports the address of the pthdb_rwlock_t.

The pthdb_rwlock_lock_count function reports the lock count for the rwlock.

The pthdb_rwlock_owner function is used to get the read/write lock owner's pthread handle.

The pthdb_rwlock_pshared function is used to get the rwlock attribute process shared value. The pshared value can be PSH_SHARED, PSH_PRIVATE, or PSH_NOTSUP.

The pthdb_rwlock_state is used to get the read/write locks state. The state can be RWLS_NOTSUP, RWLS_WRITE, RWLS_FREE, and RWLS_READ.

Parameters
addrp Read write lock address.
countp Read write lock count.
cmd cmd can be PTHDB_LIST_FIRST to get the first owner in the list of owners or PTHDB_LIST_NEXT to get the next owner in the list of owners. The list is empty or ended by *owner == PTHDB_INVALID_PTHREAD.

ownerp Pointer to pthread which owns the rwlock
psharedp Pointer to pshared value
rwlock Read write lock handle
session Session handle.
statep Pointer to state value

Return Values
If successful, these functions return PTHDB_SUCCESS. Otherwise, an error code is returned.

Error Codes
PTHDB_BAD_SESSION Invalid session handle.
PTHDB_BAD_CMD Invalid command passed.
PTHDB_CALLBACK Debugger call back error.
PTHDB_INTERNAL Error in library.
PTHDB_POINTER Invalid pointer
Related Information
The `pthdebug.h` file.
The `pthread.h` file.

**pthdb_session_committed** Subroutines

**Purpose**
Facilitates examining and modifying multi-threaded application’s pthread library object data.

**Library**
pthread debug library (**libpthread.a**)

**Syntax**

```c
#include <sys/pthdebug.h>

int pthdb_session_committed (pthdb_session_t session, char ** name);
int pthdb_session_concurrency (pthdb_session_t session, int * concurrency);
int pthdb_session_destroy (pthdb_session_t session);
int pthdb_session_flags (pthdb_session_t session, unsigned long long * flagsp);
int pthdb_session_init (pthdb_user_t user, pthdb_exec_mode_t exec_mode, unsigned long long flags, pthdb_callbacks_t * callbacks, pthdb_session_t * sessionp);
int pthdb_session_pthreaded (pthdb_user_t user, unsigned long long flags, pthdb_callbacks_t * callbacks, char ** name);
int pthdb_session_continue_tid (pthdb_session_t session, tid_t * tidp, int cmd);
int pthdb_session_stop_tid (pthdb_session_t session, tid_t tid);
int pthdb_session_commit_tid (pthdb_session_t session, tid_t * tidp, int cmd);
int pthdb_session_setflags (pthdb_session_t session, unsigned long long flags);
int pthdb_session_update (pthdb_session_t session);
```

**Description**
To facilitate debugging multiple processes, the pthread debug library supports multiple sessions, one per process. Functions are provided to initialize, destroy, and customize the behavior of these sessions. In addition, functions are provided to query global fields of the pthread library. All functions in the library require a session handle associated with an initialized session except **pthdb_session_init**, which initializes sessions, and **pthdb_session_pthreaded**, which can be called before the session has been initialized.
**pthdb_session_committed** reports the symbol name of a function called after the hold/unhold commit operation has completed. This symbol name can be used to set a breakpoint to notify the debugger when the hold/unhold commit has completed. The actual symbol name reported may change at any time. The function name returned is implemented in assembly with the following code:

```
ori 0,0, 0  # no-op
blr       # return to caller
```

This allows the debugger to overwrite the no-op with a trap instruction and leave it there by stepping over it. This function is only supported when the **PTHDB_FLAG_HOLD** flag is set.

**pthdb_session_concurrency** reports the concurrency level of the pthread library. The concurrency level is the M:N ratio, where N is always 1.

**pthdb_session_destroy** notifies the pthread debug library that the debugger or application is finished with the session. This deallocates any memory associated with the session and allows the session handle to be reused.

**pthdb_session_setflags** changes the flags for a session. With these flags, a debugger can customize the session. Flags consist of the following values or-ed together:

- **PTHDB_FLAG_GPRS**: The general purpose registers should be included in any context read or write, whether internal to the library or via call backs to the debugger.
- **PTHDB_FLAG_SPRS**: The special purpose registers should be included in any context read or write whether internal to the library or via call backs to the debugger.
- **PTHDB_FLAG_FPRS**: The floating point registers should be included in any context read or write whether internal to the library or via call backs to the debugger.
- **PTHDB_FLAG_REGS**: All registers should be included in any context read or write whether internal to the library or via call backs to the debugger. This is equivalent to **PTHDB_FLAG_GPRS|PTHDB_FLAG_GPRS|PTHDB_FLAG_GPRS**.
- **PTHDB_FLAG_HOLD**: The debugger will be using the pthread debug library hold/unhold facilities to prevent the execution of pthreads. This flag cannot be used with **PTHDB_FLAG_SUSPEND**. This flag should be used by debuggers, only.
- **PTHDB_FLAG_SUSPEND**: Applications will be using the pthread library suspend/continue facilities to prevent the execution of pthreads. This flag cannot be used with **PTHDB_FLAG_HOLD**. This flag is for introspective mode and should be used by applications, only.

**Note**: **PTHDB_FLAG_HOLD** and **PTHDB_FLAG_SUSPEND** can only be passed to the **pthdb_session_init** function. Neither **PTHDB_FLAG_HOLD** nor **PTHDB_FLAG_SUSPEND** should be passed to **pthdb_session_init** when debugging a core file.

The **pthdb_session_flags** function gets the current flags for the session.

The **pthdb_session_init** function tells the pthread debug library to initialize a session associated with the unique given user handle. **pthdb_session_init** will assign a unique session handle and return it to the debugger. If the application's execution mode is 32 bit, then the debugger should initialize the **exec_mode** to **PEM_32BIT**. If the application's execution mode is 64 bit, then the debugger should initialize **mode** to **PEM_64BIT**. The **flags** are documented above with the **pthdb_session_setflags** function. The **callback** parameter is a list of call back functions. (Also see the **pthdebug.h** header file.) The **pthdb_session_init** function calls the **symbol_addrs** function to get the starting addresses of the symbols and initializes these symbols' starting addresses within the pthread debug library.

**pthdb_session_pthreaded** reports the symbol name of a function called after the pthread library has been initialized. This symbol name can be used to set a breakpoint to notify the debugger when to initialize a pthread debug library session and begin using the pthread debug library to examine pthread library state. The actual symbol name reported may change at any time. This function is the only pthread
debug library function that can be called before the pthread library is initialized. The function name returned is implemented in assembly with the following code:

```
ori 0,0,0  # no-op
blr      # return to caller
```

This is conveniently allows the debugger to overwrite the no-op with a trap instruction and leave it there by stepping over it.

The `pthdb_session_continue_tid` function allows the debugger to obtain the list of threads that must be continued before it proceeds with single stepping a single pthread or continuing a group of pthreads. This function reports one tid at a time. If the list is empty or the end of the list has been reached, `PTHDB_INVALID_TID` is reported. The debugger will need to continue any pthreads with kernel threads that it wants. The debugger is responsible for parking the stop thread and continuing the stop thread. The `cmd` parameter can be either `PTHDB_LIST_NEXT` or `PTHDB_LIST_FIRST`; if `PTHDB_LIST_FIRST` is passed, then the internal counter will be reset and the first tid in the list will be reported.

**Note:** This function is only supported when the `PTHDB_FLAG_HOLD` flag is set.

The `pthdb_session_stop_tid` function informs the pthread debug library, which informs the pthread library the tid of the thread that stopped the debugger.

**Note:** This function is only supported when the `PTHDB_FLAG_HOLD` flag is set.

`pthdb_session_commit_tid` reports subsequent kernel thread identifiers which must be continued to commit the hold and unhold changes. This function reports one tid at a time. If the list is empty or the end of the list has been reached, `PTHDB_INVALID_TID` is reported. The `cmd` parameter can be either `PTHDB_LIST_NEXT` or `PTHDB_LIST_FIRST`, if `PTHDB_LIST_FIRST` is passed then the internal counter will be reset and first tid in the list will be reported.

**Note:** This function is only supported when the `PTHDB_FLAG_HOLD` flag is set.

`pthdb_session_update` tells the pthread debug library to update it's internal information concerning the state of the pthread library. This should be called each time the process stops before any other pthread debug library functions to ensure their results are reliable.

Each list is reset to the top of the list when the `pthdb_session_update` function is called, or when the list function reports a `PTHDB_INVALID_*` value. For example, when `pthdb_attr` reports an attribute of `PTHDB_INVALID_ATTR` the list is reset to the beginning such that the next call reports the first attribute in the list, if any.

A report of `PTHDB_INVALID_OBJECT` represents the empty list or the end of a list, where `OBJECT` is one of these values: `PTHREAD`, `ATTR`, `MUTEX`, `MUTEXATTR`, `COND`, `CONDATTR`, `RWLOCK`, `RWLOCKATTR`, `KEY`, or `TID` as appropriate.

### Parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>session</code></td>
<td>Session handle.</td>
</tr>
<tr>
<td><code>user</code></td>
<td>Debugger user handle.</td>
</tr>
<tr>
<td><code>sessionp</code></td>
<td>Pointer to session handle.</td>
</tr>
<tr>
<td><code>name</code></td>
<td>Symbol name buffer.</td>
</tr>
<tr>
<td><code>cmd</code></td>
<td>Reset to the beginning of the list.</td>
</tr>
<tr>
<td><code>concurrencyp</code></td>
<td>Library concurrency buffer.</td>
</tr>
<tr>
<td><code>flags</code></td>
<td>Session flags.</td>
</tr>
<tr>
<td><code>flagsp</code></td>
<td>Pointer to session flags.</td>
</tr>
<tr>
<td><code>exec_mode</code></td>
<td>Debuggee execution mode: <code>PEM_32BIT</code> for 32-bit processes or <code>PEM_64BIT</code> for 64-bit processes.</td>
</tr>
</tbody>
</table>
callbacks
Call backs structure.
tid
Kernel thread id.
tidp
Kernel thread id buffer.

Return Values
If successful, these functions return PTHDB_SUCCESS. Otherwise, they return an error value.

Error Codes

<table>
<thead>
<tr>
<th>Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>PTHDB_BAD_SESSION</td>
<td>Invalid session handle.</td>
</tr>
<tr>
<td>PTHDB_BAD_VERSION</td>
<td>Invalid pthread debug library or pthread library version.</td>
</tr>
<tr>
<td>PTHDB_BAD_MODE</td>
<td>Invalid execution mode.</td>
</tr>
<tr>
<td>PTHDB_BAD_FLAGS</td>
<td>Invalid session flags.</td>
</tr>
<tr>
<td>PTHDB_BAD_CALLBACK</td>
<td>Insufficient call back functions.</td>
</tr>
<tr>
<td>PTHDB_BAD_CMD</td>
<td>Invalid command.</td>
</tr>
<tr>
<td>PTHDB_BAD_POINTER</td>
<td>Invalid buffer pointer.</td>
</tr>
<tr>
<td>PTHDB_BAD_USER</td>
<td>Invalid user handle.</td>
</tr>
<tr>
<td>PTHDB_CALLBACK</td>
<td>Debugger call back error.</td>
</tr>
<tr>
<td>PTHDB_MEMORY</td>
<td>Not enough memory.</td>
</tr>
<tr>
<td>PTHDB_NOSYS</td>
<td>Function not implemented.</td>
</tr>
<tr>
<td>PTHDB_NOT_PTHREADED</td>
<td>pthread library not initialized.</td>
</tr>
<tr>
<td>PTHDB_SYMBOL</td>
<td>pthread library symbol not found.</td>
</tr>
<tr>
<td>PTHDB_INTERNAL</td>
<td>Error in library.</td>
</tr>
</tbody>
</table>

Related Information
The pthdebug.h file.

The pthread.h file.

pthread_atfork Subroutine

Purpose
Registers fork handlers.

Library
Threads Library (libpthread.a)

Syntax

```c
#include <sys/types.h>
#include <unistd.h>

int pthread_atfork (prepare, parent, child)
void (*prepare)(void);
void (*parent)(void);
void (*child)(void);
```

Description
The pthread_atfork subroutine registers fork cleanup handlers. The prepare handler is called before the processing of the fork subroutine commences. The parent handler is called after the processing of the fork subroutine completes in the parent process. The child handler is called after the processing of the fork subroutine completes in the child process.
When the `fork` subroutine is called, only the calling thread is duplicated in the child process, but all synchronization variables are duplicated. The `pthread_atfork` subroutine provides a way to prevent state inconsistencies and resulting deadlocks. The expected usage is that the `prepare` handler acquires all mutexes, and the two other handlers release them in the parent and child processes.

The prepare handlers are called in LIFO (Last In First Out) order; whereas the parent and child handlers are called in FIFO (first-in first-out) order. Thereafter, the order of calls to the `pthread_atfork` subroutine is significant.

**Note:** The `pthread.h` header file must be the first included file of each source file using the threads library.

### Parameters

- **prepare**: Points to the pre-fork cleanup handler. If no pre-fork handling is desired, the value of this pointer should be set to `NULL`.
- **parent**: Points to the parent post-fork cleanup handler. If no parent post-fork handling is desired, the value of this pointer should be set to `NULL`.
- **child**: Points to the child post-fork cleanup handler. If no child post-fork handling is desired, the value of this pointer should be set to `NULL`.

### Return Values

Upon successful completion, the `pthread_atfork` subroutine returns a value of zero. Otherwise, an error number is returned to indicate the error.

### Error Codes

The `pthread_atfork` subroutine will fail if:

- **ENOMEM** Insufficient table space exists to record the fork handler addresses.

The `pthread_atfork` subroutine will not return an error code of `EINTR`.

### Related Information

- **The `fork` subroutine**, `f_fork`, or `vfork` subroutine, `atexit` subroutine, `exit`, `atexit`, `unatexit`, `_exit`, or `_Exit` subroutine.

- The **`posix_spawn` or `posix_spawnp` subroutine** on page 1129.

- `Process Duplication and Termination` in *AIX 5L Version 5.3 General Programming Concepts: Writing and Debugging Programs*.

### `pthread_attr_destroy` Subroutine

#### Purpose

Deletes a thread attributes object.

#### Library

Threads Library (`libpthreads.a`)
## Syntax

```c
#include <pthread.h>

int pthread_attr_destroy (attr)
    pthread_attr_t *attr;
```

## Description

The `pthread_attr_destroy` subroutine destroys the thread attributes object `attr`, reclaiming its storage space. It has no effect on the threads previously created with that object.

## Parameters

- `attr` Specifies the thread attributes object to delete.

## Return Values

Upon successful completion, 0 is returned. Otherwise, an error code is returned.

## Error Codes

The `pthread_attr_destroy` subroutine is unsuccessful if the following is true:

- **EINVAL** The `attr` parameter is not valid.

This function will not return an error code of [EINVAL].

## Related Information

- [Creating Threads in AIX 5L Version 5.3 General Programming Concepts: Writing and Debugging Programs](#)

---

### pthread_attr_getguardsize or pthread_attr_setguardsize Subroutines

#### Purpose

Gets or sets the thread guardsize attribute.

#### Library

Threads Library (libthreads.a)

#### Syntax

```c
#include <pthread.h>

int pthread_attr_getguardsize (attr, guardsize)
    const pthread_attr_t *attr;
    size_t *guardsize;

int pthread_attr_setguardsize (attr, guardsize)
    pthread_attr_t *attr;
    size_t guardsize;
```
Description
The `guardsize` attribute controls the size of the guard area for the created thread’s stack. The `guardsize` attribute provides protection against overflow of the stack pointer. If a thread’s stack is created with guard protection, the implementation allocates extra memory at the overflow end of the stack as a buffer against stack overflow of the stack pointer. If an application overflows into this buffer an error results (possibly in a SIGSEGV signal being delivered to the thread).

The `guardsize` attribute is provided to the application for two reasons:

- Overflow protection can potentially result in wasted system resources. An application that creates a large number of threads, and which knows its threads will never overflow their stack, can save system resources by turning off guard areas.
- When threads allocate large data structures on the stack, large guard areas may be needed to detect stack overflow.

The `pthread_attr_getguardsize` function gets the `guardsize` attribute in the `attr` object. This attribute is returned in the `guardsize` parameter.

The `pthread_attr_setguardsize` function sets the `guardsize` attribute in the `attr` object. The new value of this attribute is obtained from the `guardsize` parameter. If `guardsize` is zero, a guard area will not be provided for threads created with `attr`. If `guardsize` is greater than zero, a guard area of at least size `guardsize` bytes is provided for each thread created with `attr`.

A conforming implementation is permitted to round up the value contained in `guardsize` to a multiple of the configurable system variable `PAGESIZE` (see `sys/mman.h`). If an implementation rounds up the value of `guardsize` to a multiple of `PAGESIZE`, a call to `pthread_attr_getguardsize` specifying `attr` will store in the `guardsize` parameter the guard size specified by the previous `pthread_attr_setguardsize` function call. The default value of the `guardsize` attribute is `PAGESIZE` bytes. The actual value of `PAGESIZE` is implementation-dependent and may not be the same on all implementations.

If the `stackaddr` attribute has been set (that is, the caller is allocating and managing its own thread stacks), the `guardsize` attribute is ignored and no protection will be provided by the implementation. It is the responsibility of the application to manage stack overflow along with stack allocation and management in this case.

Parameters

- `attr` Specifies the thread attributes object.
- `guardsize` Controls the size of the guard area for the created thread’s stack, and protects against overflow of the stack pointer.

Return Values
If successful, the `pthread_attr_getguardsize` and `pthread_attr_setguardsize` functions return zero. Otherwise, an error number is returned to indicate the error.

Error Codes
The `pthread_attr_getguardsize` and `pthread_attr_setguardsize` functions will fail if:

- `EINVAL` The attribute `attr` is invalid.
- `EINVAL` The `guardsize` parameter is invalid.
- `EINVAL` The `guardsize` parameter contains an invalid value.
pthread_attr_getinheritsched, pthread_attr_setinheritsched Subroutine

**Purpose**
Gets and sets the `inheritsched` attribute (REALTIME THREADS).

**Syntax**
```c
#include <pthread.h>
#include <time.h>

int pthread_attr_getinheritsched(const pthread_attr_t *restrict attr,
                                int *restrict inheritsched);
int pthread_attr_setinheritsched(pthread_attr_t *attr,
                                 int inheritsched);
```

**Description**
The `pthread_attr_getinheritsched()` and `pthread_attr_setinheritsched()` functions, respectively, get and set the `inheritsched` attribute in the `attr` argument.

When the attributes objects are used by `pthread_create()`, the `inheritsched` attribute determines how the other scheduling attributes of the created thread are set.

- **PTHREAD_INHERIT_SCHED**
  Specifies that the thread scheduling attributes is inherited from the creating thread, and the scheduling attributes in this `attr` argument are ignored.

- **PTHREAD_EXPLICIT_SCHED**
  Specifies that the thread scheduling attributes are set to the corresponding values from this attributes object.

The PTHREAD_INHERIT_SCHED and PTHREAD_EXPLICIT_SCHED symbols are defined in the `<pthread.h>` header.

The following thread scheduling attributes defined by IEEE Std 1003.1-2001 are affected by the `inheritsched` attribute: scheduling policy (schedpolicy), scheduling parameters (schedparam), and scheduling contention scope (contentionscope).

**Application Usage**
After these attributes have been set, a thread can be created with the specified attributes using `pthread_create()`. Using these routines does not affect the current running thread.

**Return Values**
If successful, the `pthread_attr_getinheritsched()` and `pthread_attr_setinheritsched()` functions return 0; otherwise, an error number is returned to indicate the error.

**Error Codes**
The `pthread_attr_setschedpolicy()` function might fail if:

- **EINVAL**
  The value of `inheritsched` is not valid.
- **ENOTSUP**
  An attempt was made to set the attribute to an unsupported value.

These functions do not return an error code of EINTR.
Related Information


The pthread.h and sched.h files in AIX 5L Version 5.3 Files Reference.

pthread_attr_getschedparam Subroutine

Purpose

Returns the value of the schedparam attribute of a thread attributes object.

Library

Threads Library (libpthreads.a)

Syntax

#include <pthread.h>
#include <sys/sched.h>

int pthread_attr_getschedparam (attr, schedparam)
const pthread_attr_t *attr;
struct sched_param *schedparam;

Description

The pthread_attr_getschedparam subroutine returns the value of the schedparam attribute of the thread attributes object attr. The schedparam attribute specifies the scheduling parameters of a thread created with this attributes object. The sched_priority field of the sched_param structure contains the priority of the thread. It is an integer value.

Note: The pthread.h header file must be the first included file of each source file using the threads library. Otherwise, the -D THREAD_SAFE compilation flag should be used, or the cc_r compiler used. In this case, the flag is automatically set.

Parameters

attr Specifies the thread attributes object.
schedparam Points to where the schedparam attribute value will be stored.

Return Values

Upon successful completion, the value of the schedparam attribute is returned via the schedparam parameter, and 0 is returned. Otherwise, an error code is returned.

Error Codes

The pthread_attr_getschedparam subroutine is unsuccessful if the following is true:

EINVAL The attr parameter is not valid.

This function does not return EINTR.
pThread_attr_getschedpolicy, pThread_attr_setschedpolicy Subroutine

**Purpose**

Gets and sets the schedpolicy attribute (REALTIME THREADS).

**Syntax**

```c
#include <pthread.h>
#include <time.h>

int pthread_attr_getschedpolicy(const pthread_attr_t *restrict attr,
                                int *restrict policy);
int pthread_attr_setschedpolicy(pthread_attr_t *attr, int policy);
```

**Description**

The `pthread_attr_getschedpolicy()` and `pthread_attr_setschedpolicy()` functions, respectively, get and set the schedpolicy attribute in the `attr` argument.

The supported values of policy include SCHED_FIFO, SCHED_RR, and SCHED_OTHER, which are defined in the `<sched.h>` header. When threads executing with the scheduling policy SCHED_FIFO, SCHED_RR, or SCHED_SPORADIC are waiting on a mutex, they acquire the mutex in priority order when the mutex is unlocked.

**Application Usage**

After these attributes have been set, a thread can be created with the specified attributes using `pthread_create()`. Using these routines does not affect the current running thread.

**Return Values**

If successful, the `pthread_attr_getschedpolicy()` and `pthread_attr_setschedpolicy()` functions return 0; otherwise, an error number is returned to indicate the error.

**Error Codes**

The `pthread_attr_setschedpolicy()` function might fail if:

- **EINVAL** The value of policy is not valid.
- **ENOTSUP** An attempt was made to set the attribute to an unsupported value.

These functions do not return an error code of EINTR.
**Related Information**


The `pthread.h` and `time.h` files in AIX 5L Version 5.3 Files Reference.

---

**pthread_attr_getstackaddr Subroutine**

**Purpose**

Returns the value of the stackaddr attribute of a thread attributes object.

**Library**

Threads Library (libpthread.a)

**Syntax**

```c
#include <pthread.h>

int pthread_attr_getstackaddr(const pthread_attr_t *attr, void **stackaddr);
```

**Description**

The `pthread_attr_getstackaddr` subroutine returns the value of the stackaddr attribute of the thread attributes object `attr`. This attribute specifies the stack address of the thread created with this attributes object.

**Note:** The `pthread.h` header file must be the first included file of each source file using the threads library. Otherwise, the `-D_THREAD_SAFE` compilation flag should be used, or the `cc_r` compiler used. In this case, the flag is automatically set.

**Parameters**

- **attr** Specifies the thread attributes object.
- **stackaddr** Points to where the stackaddr attribute value will be stored.

**Return Values**

Upon successful completion, the value of the stackaddr attribute is returned via the `stackaddr` parameter, and 0 is returned. Otherwise, an error code is returned.

**Error Codes**

The `pthread_attr_getstackaddr` subroutine is unsuccessful if the following is true:

- **EINVAL** The `attr` parameter is not valid.

This function will not return EINTR.
pthread_attr_getstacksize Subroutine

Purpose
Returns the value of the stacksize attribute of a thread attributes object.

Library
Threads Library (libpthreads.a)

Syntax
#include <pthread.h>

int pthread_attr_getstacksize
    (attr, stacksize);

Description
The pthread_attr_getstacksize subroutine returns the value of the stacksize attribute of the thread attributes object attr. This attribute specifies the minimum stacksize of a thread created with this attributes object. The value is given in bytes. For 32-bit compiled applications, the default stacksize is 96 KB (defined in the pthread.h file). For 64-bit compiled applications, the default stacksize is 192 KB (defined in the pthread.h file).

Note: The pthread.h header file must be the first included file of each source file using the threads library. Otherwise, the -D_THREAD_SAFE compilation flag should be used, or the cc_r compiler used. In this case, the flag is automatically set.

Parameters
attr Specifies the thread attributes object.
stacksize Points to where the stacksize attribute value will be stored.

Return Values
Upon successful completion, the value of the stacksize attribute is returned via the stacksize parameter, and 0 is returned. Otherwise, an error code is returned.

Error Codes
The pthread_attr_getstacksize subroutine is unsuccessful if the following is true:

EINVAL The attr or stacksize parameters are not valid.

This function will not return an error code of [EINTR].
Related Information

The pthread_attr_setstacksize subroutine, pthread_attr_init subroutine, the pthread.h file.

Advanced Attributes in AIX 5L Version 5.3 General Programming Concepts: Writing and Debugging Programs.

Threads Library Options in AIX 5L Version 5.3 General Programming Concepts: Writing and Debugging Programs.

pthread_attr_init Subroutine

Purpose
Creates a thread attributes object and initializes it with default values.

Library
Threads Library (libpthreads.a)

Syntax
#include <pthread.h>

int pthread_attr_init (attr)
    pthread_attr_t * attr;

Description
The pthread_attr_init subroutine creates a new thread attributes object attr. The new thread attributes object is initialized with the following default values:

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Default value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Detachstate</td>
<td>PTHREAD_CREATE_JOINABLE</td>
</tr>
<tr>
<td>Contention-scope</td>
<td>PTHREAD_SCOPE_PROCESS the default ensures compatibility with implementations that do not support this POSIX option.</td>
</tr>
<tr>
<td>Inheritsched</td>
<td>PTHREAD_INHERITSCHED</td>
</tr>
<tr>
<td>Schedparam</td>
<td>A sched_param structure which sched_prio field is set to 1, the least favored priority.</td>
</tr>
<tr>
<td>Schedpolicy</td>
<td>SCHED_OTHER</td>
</tr>
<tr>
<td>Stacksize</td>
<td>PTHREAD_STACK_MIN</td>
</tr>
<tr>
<td>Guardsize</td>
<td>PAGESIZE</td>
</tr>
</tbody>
</table>

The resulting attribute object (possibly modified by setting individual attribute values), when used by pthread_create, defines the attributes of the thread created. A single attributes object can be used in multiple simultaneous calls to pthread_create.

Parameters

attr Specifies the thread attributes object to be created.
Return Values
Upon successful completion, the new thread attributes object is filled with default values and returned via the attr parameter, and 0 is returned. Otherwise, an error code is returned.

Error Codes
The pthread_attr_init subroutine is unsuccessful if the following is true:

EINVAL  The attr parameter is not valid.
ENOMEM   There is not sufficient memory to create the thread attribute object.

This function will not return an error code of [EINTR].

Related Information
The pthread_attr_setdetachstate ("pthread_attr_getdetachstate or pthread_attr_setdetachstate Subroutines") subroutine, pthread_attr_setstackaddr ("pthread_attr_setstackaddr Subroutine" on page 1202) subroutine, pthread_attr_setstacksize ("pthread_attr_setstacksize Subroutine" on page 1203) subroutine, pthread_create ("pthread_create Subroutine" on page 1222) subroutine, pthread_attr_destroy ("pthread_attr_destroy Subroutine" on page 1189) subroutine, pthread_attr_setguardsize ("pthread_attr_getguardsize or pthread_attr_setguardsize Subroutines" on page 1190) subroutine.

The pthread.h file.

Creating Threads in AIX 5L Version 5.3 General Programming Concepts: Writing and Debugging Programs.

Threads Library Options in AIX 5L Version 5.3 General Programming Concepts: Writing and Debugging Programs.

pthread_attr_getdetachstate or pthread_attr_setdetachstate Subroutines

Purpose
Sets and returns the value of the detachstate attribute of a thread attributes object.

Library
Threads Library (libpthreads.a)

Syntax
#include <pthread.h>

int pthread_attr_setdetachstate (attr, detachstate)
pthread_attr_t *attr;
int detachstate;

int pthread_attr_getdetachstate (attr, detachstate)
const pthread_attr_t *attr;
int *detachstate;
Description
The detachstate attribute controls whether the thread is created in a detached state. If the thread is
created detached, then use of the ID of the newly created thread by the `pthread_detach` or `pthread_join`
function is an error.

The `pthread_attr_setdetachstate` and `pthread_attr_getdetachstate`, respectively, set and get the
detachstate attribute in the `attr` object.

The detachstate attribute can be set to either PTHREAD_CREATE_DETACHED or
PTHREAD_CREATE_JOINABLE. A value of PTHREAD_CREATE_DETACHED causes all threads created
with `attr` to be in the detached state, whereas using a value of PTHREAD_CREATE_JOINABLE causes all
threads created with `attr` to be in the joinable state. The default value of the detachstate attribute is
PTHREAD_CREATE_JOINABLE.

Parameters

- **attr** Specifies the thread attributes object.
- **detachstate** Points to where the detachstate attribute value will be stored.

Return Values
Upon successful completion, `pthread_attr_setdetachstate` and `pthread_attr_getdetachstate` return a
value of 0. Otherwise, an error number is returned to indicate the error.

The `pthread_attr_getdetachstate` function stores the value of the detachstate attribute in the `detachstate`
parameter if successful.

Error Codes
The `pthread_attr_setdetachstate` function will fail if:

- **EINVAL** The value of `detachstate` was not valid.

The `pthread_attr_getdetachstate` and `pthread_attr_setdetachstate` functions will fail if:

- **EINVAL** The attribute parameter is invalid.

These functions will not return an error code of EINTR.

Related Information
The `pthread_attr_setstackaddr Subroutine` on page 1202, `pthread_attr_setstacksize Subroutine` on page
1203, `pthread_create Subroutine` on page 1222, and `pthread_attr_init Subroutine` on page 1197.

The `pthread.h` file in AIX 5L Version 5.3 Files Reference

Creating Threads in AIX 5L Version 5.3 General Programming Concepts: Writing and Debugging
Programs.

`pthread_attr_getscope` and `pthread_attr_setscope` Subroutines

Purpose
Gets and sets the scope attribute in the `attr` object.
Library
Threads Library (libthreads.a)

Syntax

```
#include <pthread.h>

int pthread_attr_setscope(pthread_attr_t *attr, int contentionscope);
int *contentionscope;

int pthread_attr_getscope(const pthread_attr_t *attr, int *contentionscope);
```

Description

The scope attribute controls whether a thread is created in system or process scope.

The `pthread_attr_getscope` and `pthread_attr_setscope` subroutines get and set the scope attribute in the `attr` object.

The scope can be set to `PTHREAD_SCOPE_SYSTEM` or `PTHREAD_SCOPE_PROCESS`. A value of `PTHREAD_SCOPE_SYSTEM` causes all threads created with the `attr` parameter to be in system scope, whereas a value of `PTHREAD_SCOPE_PROCESS` causes all threads created with the `attr` parameter to be in process scope.

The default value of the `contentionscope` parameter is `PTHREAD_SCOPE_PROCESS`.

Parameters

- `attr` Specifies the thread attributes object.
- `contentionscope` Points to where the scope attribute value will be stored.

Return Values

Upon successful completion, the `pthread_attr_getscope` and `pthread_attr_setscope` subroutines return a value of 0. Otherwise, an error number is returned to indicate the error.

Error Codes

- `EINVAL` The value of the attribute being set/read is not valid.
- `ENOTSUP` An attempt was made to set the attribute to an unsupported value.

Related Information

- The `pthread_create Subroutine` on page 1222, and `pthread_attr_init Subroutine` on page 1197.
- The `pthread.h` file in `AIX 5L Version 5.3 Files Reference`.
- `Creating Threads` in `AIX 5L Version 5.3 General Programming Concepts: Writing and Debugging Programs`.

1200 Technical Reference, Volume 1: Base Operating System and Extensions
pthread_attr_setschedparam Subroutine

Purpose
Sets the value of the schedparam attribute of a thread attributes object.

Library
Threads Library (libpthreads.a)

Syntax
#include <pthread.h>
#include <sys/sched.h>

int pthread_attr_setschedparam (attr, schedparam)
pthread_attr_t *attr;
const struct sched_param *schedparam;

Description
The pthread_attr_setschedparam subroutine sets the value of the schedparam attribute of the thread attributes object attr. The schedparam attribute specifies the scheduling parameters of a thread created with this attributes object. The sched_priority field of the sched_param structure contains the priority of the thread.

Note: The pthread.h header file must be the first included file of each source file using the threads library. Otherwise, the -D_THREAD_SAFE compilation flag should be used, or the cc_r compiler used. In this case, the flag is automatically set.

Parameters
attr Specifies the thread attributes object.
schedparam Points to where the scheduling parameters to set are stored. The sched_priority field must be in the range from 1 to 127, where 1 is the least favored priority, and 127 the most favored.

Return Values
Upon successful completion, 0 is returned. Otherwise, an error code is returned.

Error Codes
The pthread_attr_setschedparam subroutine is unsuccessful if the following is true:

EINVAL The attr parameter is not valid.
ENOSYS The priority scheduling POSIX option is not implemented.
ENOTSUP The value of the schedparam attribute is not supported.

Related Information
The pthread_attr_getschedparam subroutine, pthread_attr_init subroutine, pthread_create subroutine, the pthread.h file.

Threads Scheduling in AIX 5L Version 5.3 General Programming Concepts: Writing and Debugging Programs.
### pthread_attr_setstackaddr Subroutine

#### Purpose
Sets the value of the stackaddr attribute of a thread attributes object.

#### Library
Threads Library (libpthreads.a)

#### Syntax
```c
#include <pthread.h>

int pthread_attr_setstackaddr (attr, stackaddr);

pthread_attr_t *attr;
void *stackaddr;
```

#### Description
The `pthread_attr_setstackaddr` subroutine sets the value of the stackaddr attribute of the thread attributes object `attr`. This attribute specifies the stack address of a thread created with this attributes object.

**Note:** The `pthread.h` header file must be the first included file of each source file using the threads library. Otherwise, the `-D_THREAD_SAFE` compilation flag should be used, or the cc_r compiler used. In this case, the flag is automatically set.

A Provision has been made in `libpthreads` to create guardpages for the user stack internally. This is used for debugging purposes only. By default, it is turned off and can be invoked by exporting the following environment variable:

```
AIXTHREAD_GUARDPAGES_FOR_USER_STACK=n
```

(Where `n` is the decimal number of guard pages.)

**Note:** Even if it is exported, guard pages will only be constructed if both the stackaddr and stacksize attributes have been set by the caller for the thread. Also, the guard pages and alignment pages will be created out of the user’s stack (which will reduce the stack size). If the new stack size after creating guard pages is less than the minimum stack size (PTHREAD_STACK_MIN), then the guard pages will not be constructed.

#### Parameters
- `attr` Specifies the thread attributes object.
- `stackaddr` Specifies the stack address to set. It is a void pointer. The address that needs to be passed is not the beginning of the malloc generated address but the beginning of the stack. For example:

  ```c
  stackaddr = malloc(stacksize);
  pthread_attr_setstackaddr(&thread, stackaddr + stacksize);
  ```

#### Return Values
Upon successful completion, 0 is returned. Otherwise, an error code is returned.
Error Codes
The pthread_attr_setstackaddr subroutine is unsuccessful if the following is true:

EINVAL  The attr parameter is not valid.
ENOSYS  The stack address POSIX option is not implemented.

Related Information
The pthread_attr_getstackaddr Subroutine” on page 1195, "pthread_attr_init Subroutine” on page 1197, pthread.h file.

Advanced Attributes in AIX 5L Version 5.3 General Programming Concepts: Writing and Debugging Programs.

Threads Library Options in AIX 5L Version 5.3 General Programming Concepts: Writing and Debugging Programs.

pthread_attr_setstacksize Subroutine

Purpose
Sets the value of the stacksize attribute of a thread attributes object.

Library
Threads Library (libpthreads.a)

Syntax
#include <pthread.h>

int pthread_attr_setstacksize (attr, stacksize)

pthread_attr_t *attr;
size_t stacksize;

Description
The pthread_attr_setstacksize subroutine sets the value of the stacksize attribute of the thread attributes object attr. This attribute specifies the minimum stack size, in bytes, of a thread created with this attributes object.

The allocated stack size is always a multiple of 8K bytes, greater or equal to the required minimum stack size of 56K bytes (PTHREAD_STACK_MIN). The following formula is used to calculate the allocated stack size: if the required stack size is lower than 56K bytes, the allocated stack size is 56K bytes; otherwise, if the required stack size belongs to the range from (56 + (n - 1) * 16) K bytes to (56 + n * 16) K bytes, the allocated stack size is (56 + n * 16) K bytes.

Note: The pthread.h header file must be the first included file of each source file using the threads library. Otherwise, the -D_THREAD_SAFE compilation flag should be used, or the cc_r compiler used. In this case, the flag is automatically set.

Parameters
attr  Specifies the thread attributes object.
stacksize  Specifies the minimum stack size, in bytes, to set. The default stack size is PTHREAD_STACK_MIN. The minimum stack size should be greater or equal than this value.
Return Values
Upon successful completion, 0 is returned. Otherwise, an error code is returned.

Error Codes
The `pthread_attr_setstacksize` subroutine is unsuccessful if the following is true:

- **EINVAL**  The `attr` parameter is not valid, or the value of the `stacksize` parameter exceeds a system imposed limit.
- **ENOSYS** The stack size POSIX option is not implemented.

Related Information
The `pthread_attr_getstacksize` subroutine, `pthread_attr_init` subroutine, `pthread_create` subroutine, the `pthread.h` file.

`pthread_attr_setsuspendstate_np` and `pthread_attr_getsuspendstate_np` Subroutine

Purpose
Controls whether a thread is created in a suspended state.

Library
Threads Library (`libpthreads.a`)

Syntax
```
#include <pthread.h>

int pthread_attr_setsuspendstate_np (`attr`, `suspendstate`)  
pthread_attr_t *attr;  
int `suspendstate`;

int pthread_attr_getsuspendstate_np (`attr`, `suspendstate`)  
pthread_attr_t *attr;  
int *`suspendstate`;
```

Description
The `suspendstate` attribute controls whether the thread is created in a suspended state. If the thread is created suspended, the thread start routine will not execute until `pthread_continue_np` is run on the thread. The `pthread_attr_setsuspendstate_np` and `pthread_attr_getsuspendstate_np` routines, respectively, set and get the `suspendstate` attribute in the `attr` object.

The `suspendstate` attribute can be set to either `PTHREAD_CREATE_SUSPENDED_NP` or `PTHREAD_CREATE_UNSUSPENDED_NP`. A value of `PTHREAD_CREATE_SUSPENDED_NP` causes all threads created with `attr` to be in the suspended state, whereas using a value of `PTHREAD_CREATE_UNSUSPENDED_NP` causes all threads created with `attr` to be in the unsuspended state. The default value of the `suspendstate` attribute is `PTHREAD_CREATE_UNSUSPENDED_NP`.
Parameters

attr
Specifies the thread attributes object.
suspendstate
Points to where the suspendstate attribute value will be stored.

Return Values

Upon successful completion, pthread_attr_setsuspendstate_np and pthread_attr_getsuspendstate_np return a value of 0. Otherwise, an error number is returned to indicate the error.

The pthread_attr_getsuspendstate_np function stores the value of the suspendstate attribute in suspendstate if successful.

Error Codes

The pthread_attr_setsuspendstate_np function will fail if:

EINVAL
The value of suspendstate is not valid.

pthread_barrier_destroy or pthread_barrier_init Subroutine

Purpose

Destroys or initializes a barrier object.

Syntax

#include <pthread.h>

int pthread_barrier_destroy(pthread_barrier_t *barrier);
int pthread_barrier_init(pthread_barrier_t *restrict barrier,
const pthread_barrierattr_t *restrict attr,
unsigned count);

Description

The pthread_barrier_destroy subroutine destroys the barrier referenced by the barrier parameter and releases any resources used by the barrier. The effect of subsequent use of the barrier is undefined until the barrier is reinitialized by another call to the pthread_barrier_init subroutine. An implementation can use this subroutine to set the barrier parameter to an invalid value. The results are undefined if the pthread_barrier_destroy subroutine is called when any thread is blocked on the barrier, or if this function is called with an uninitialized barrier.

The pthread_barrier_init subroutine allocates any resources required to use the barrier referenced by the barrier parameter and initializes the barrier with attributes referenced by the attr parameter. If the attr parameter is NULL, the default barrier attributes are used; the effect is the same as passing the address of a default barrier attributes object. The results are undefined if pthread_barrier_init subroutine is called when any thread is blocked on the barrier (that is, has not returned from the pthread_barrier_wait call). The results are undefined if a barrier is used without first being initialized. The results are undefined if the pthread_barrier_init subroutine is called specifying an already initialized barrier.

The count argument specifies the number of threads that must call the pthread_barrier_wait subroutine before any of them successfully return from the call. The value specified by the count parameter must be greater than zero.

If the pthread_barrier_init subroutine fails, the barrier is not initialized and the contents of barrier are undefined.
Only the object referenced by the barrier parameter can be used for performing synchronization. The result of referring to copies of that object in calls to the `pthread_barrier_destroy` or `pthread_barrier_wait` subroutine is undefined.

**Return Values**
Upon successful completion, these functions shall return zero; otherwise, an error number shall be returned to indicate the error.

**Error Codes**
The `pthread_barrier_destroy` subroutine can fail if:

<table>
<thead>
<tr>
<th>Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EBUSY</td>
<td>The implementation has detected an attempt to destroy a barrier while it is in use (for example, while being used in a <code>pthread_barrier_wait</code> call) by another thread.</td>
</tr>
<tr>
<td>EINVAL</td>
<td>The value specified by barrier is invalid.</td>
</tr>
</tbody>
</table>

The `pthread_barrier_init()` function will fail if:

<table>
<thead>
<tr>
<th>Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EAGAIN</td>
<td>The system lacks the necessary resources to initialize another barrier.</td>
</tr>
<tr>
<td>EINVAL</td>
<td>The value specified by the count parameter is equal to zero.</td>
</tr>
<tr>
<td>ENOMEM</td>
<td>Insufficient memory exists to initialize the barrier.</td>
</tr>
</tbody>
</table>

The `pthread_barrier_init` subroutine can fail if:

<table>
<thead>
<tr>
<th>Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EBUSY</td>
<td>The implementation has detected an attempt to reinitialize a barrier while it is in use (for example, while being used in a <code>pthread_barrier_wait</code> call) by another thread.</td>
</tr>
<tr>
<td>EINVAL</td>
<td>The value specified by the attr parameter is invalid.</td>
</tr>
</tbody>
</table>

**Related Information**
The `pthread.h` file.

---

**pthread_barrier_wait Subroutine**

**Purpose**
Synchronizes threads at a barrier.

**Syntax**
```c
#include <pthread.h>

int pthread_barrier_wait(pthread_barrier_t *barrier);
```

**Description**
The `pthread_barrier_wait` subroutine synchronizes participating threads at the barrier referenced by barrier. The calling thread blocks until the required number of threads have called `pthread_barrier_wait` specifying the barrier.
When the required number of threads have called `pthread_barrier_wait` specifying the barrier, the constant `PTHREAD_BARRIER_SERIAL_THREAD` is returned to one unspecified thread and 0 is returned to the remaining threads. At this point, the barrier resets to the state it had as a result of the most recent `pthread_barrier_init` function that referenced it.

The constant `PTHREAD_BARRIER_SERIAL_THREAD` is defined in `<pthread.h>`, and its value is distinct from any other value returned by `pthread_barrier_wait`.

The results are undefined if this function is called with an uninitialized barrier.

If a signal is delivered to a thread blocked on a barrier, upon return from the signal handler, the thread resumes waiting at the barrier if the barrier wait has not completed (that is, if the required number of threads have not arrived at the barrier during the execution of the signal handler); otherwise, the thread continues as normal from the completed barrier wait. Until the thread in the signal handler returns from it, other threads might proceed past the barrier after they have all reached it.

**Note:** In AIX 5.3, when the required number of threads has called `pthread_barrier_wait`, the `PTHREAD_BARRIER_SERIAL_THREAD` constant is returned by the last pthread that called `pthread_barrier_wait`. Furthermore, if a thread is in a signal handler while waiting and all the required threads have reached the barrier, the other threads can proceed past the barrier.

A thread that has blocked on a barrier does not prevent any unblocked thread that is eligible to use the same processing resources from eventually making forward progress in its execution. Eligibility for processing resources is determined by the scheduling policy.

**Parameters**

`barrier` Points to the barrier where participating threads wait.

**Return Values**

Upon successful completion, `pthread_barrier_wait` returns `PTHREAD_BARRIER_SERIAL_THREAD` for a single (arbitrary) thread synchronized at the barrier and 0 for the other threads. Otherwise, an error number is returned to indicate the error.

**Error Codes**

The `pthread_barrier_destroy` subroutine can fail if:

- `EINVAL` The value specified by `barrier` does not refer to an initialized barrier object.

This function does not return an error code of `EINTR`.

**Related Information**

The ["pthread_barrier_destroy or pthread_barrier_init Subroutine" on page 1205](#), ["pthread_barrierattr_destroy or pthread_barrierattr_init Subroutine;" #](#), ["pthread_barrierattr_getpshared or pthread_barrierattr_setpshared Subroutine" on page 1208](#).

The `pthread.h` file.

**pthread_barrierattr_destroy or pthread_barrierattr_init Subroutine**

**Purpose**

Destroys or initializes the barrier attributes object.
Syntax
#include <pthread.h>

int pthread_barrierattr_destroy(pthread_barrierattr_t *attr);
int pthread_barrierattr_init(pthread_barrierattr_t *attr);

Description
The `pthread_barrierattr_destroy` subroutine destroys a barrier attributes object. A destroyed `attr` attributes object can be reinitialized using the `pthread_barrierattr_init` subroutine; the results of otherwise referencing the object after it has been destroyed are undefined. An implementation can cause the `pthread_barrierattr_destroy` subroutine to set the object referenced by the `attr` parameter to an invalid value.

The `pthread_barrierattr_init` subroutine initializes a barrier attributes object `attr` with the default value for all of the attributes defined by the implementation.

Results are undefined if the `pthread_barrierattr_init` subroutine is called specifying an already initialized `attr` attributes object.

After a barrier attributes object has been used to initialize one or more barriers, any function affecting the attributes object (including destruction) do not affect any previously initialized barrier.

Return Values
If successful, the `pthread_barrierattr_destroy` and `pthread_barrierattr_init` subroutines return zero; otherwise, an error number shall be returned to indicate the error.

Error Codes
The `pthread_barrierattr_destroy` subroutine can fail if:

EINVAL
The value specified by the `attr` parameter is invalid.

The `pthread_barrierattr_init` subroutine will fail if:

ENOMEM
Insufficient memory exists to initialize the barrier attributes object.

Related Information
The "pthread_barrier_destroy or pthread_barrier_init Subroutine" on page 1205, "pthread_barrier_wait Subroutine" on page 1206, "pthread_barrierattr_getpshared or pthread_barrierattr_setpshared Subroutine."

`pthread_barrierattr_getpshared` or `pthread_barrierattr_setpshared` Subroutine

Purpose
Gets and sets the process-shared attribute of the barrier attributes object.

Syntax
#include <pthread.h>

int pthread_barrierattr_getpshared(const pthread_barrierattr_t *restrict attr, int *restrict pshared);
int pthread_barrierattr_setpshared(pthread_barrierattr_t *attr, int pshared);
Description
The `pthread_barrierattr_getpshared` subroutine obtains the value of the process-shared attribute from the attributes object referenced by the `attr` parameter. The `pthread_barrierattr_setpshared` subroutine sets the process-shared attribute in an initialized attributes object referenced by the `attr` parameter.

The process-shared attribute is set to `PTHREAD_PROCESS_SHARED` to permit a barrier to be operated upon by any thread that has access to the memory where the barrier is allocated. If the process-shared attribute is `PTHREAD_PROCESS_PRIVATE`, the barrier is only operated upon by threads created within the same process as the thread that initialized the barrier; if threads of different processes attempt to operate on such a barrier, the behavior is undefined. The default value of the attribute is `PTHREAD_PROCESS_PRIVATE`. Both constants `PTHREAD_PROCESS_SHARED` and `PTHREAD_PROCESS_PRIVATE` are defined in the `pthread.h` file.

Additional attributes, their default values, and the names of the associated functions to get and set those attribute values are implementation-defined.

Return Values
If successful, the `pthread_barrierattr_getpshared` subroutine will return zero and store the value of the process-shared attribute of `attr` into the object referenced by the `pshared` parameter. Otherwise, an error number shall be returned to indicate the error.

If successful, the `pthread_barrierattr_setpshared` subroutine will return zero; otherwise, an error number shall be returned to indicate the error.

Error Codes
These functions may fail if:

**EINVAL**
- The value specified by `attr` is invalid.

The `pthread_barrierattr_setpshared` subroutine will fail if:

**EINVAL**
The new value specified for the process-shared attribute is not one of the legal values `PTHREAD_PROCESS_SHARED` or `PTHREAD_PROCESS_PRIVATE`.

Related Information
The `pthread_barrier_destroy or pthread_barrier_init Subroutine` on page 1205, `pthread_barrier_wait Subroutine` on page 1206, `pthread_barrierattr_destroy or pthread_barrierattr_init Subroutine` on page 1207.

**pthread_cancel Subroutine**

Purpose
Requests the cancellation of a thread.

Library
Threads Library (`libpthreads.a`)
Syntax

```c
#include <pthread.h>

int pthread_cancel (thread)
    pthread_t thread;
```

Description

The `pthread_cancel` subroutine requests the cancellation of the thread `thread`. The action depends on the cancelability of the target thread:

- If its cancelability is disabled, the cancellation request is set pending.
- If its cancelability is deferred, the cancellation request is set pending till the thread reaches a cancellation point.
- If its cancelability is asynchronous, the cancellation request is acted upon immediately; in some cases, it may result in unexpected behavior.

The cancellation of a thread terminates it safely, using the same termination procedure as the `pthread_exit` subroutine.

Note: The `pthread.h` header file must be the first included file of each source file using the threads library. Otherwise, the `-D_THREAD_SAFE` compilation flag should be used, or the cc_r compiler used. In this case, the flag is automatically set.

Parameters

- `thread` Specifies the thread to be canceled.

Return Values

If successful, the `pthread_cancel` function returns zero. Otherwise, an error number is returned to indicate the error.

Error Codes

The `pthread_cancel` function may fail if:

- ESRCH No thread could be found corresponding to that specified by the given thread ID.

The `pthread_cancel` function will not return an error code of EINTR.

Related Information

The `pthread_kill` subroutine, `pthread_exit` subroutine, `pthread_join` subroutine, `pthread_cond_wait` subroutine, and `pthread_cond_timedwait` subroutine subroutines.

The `pthread.h` file.

Terminating Threads in AIX 5L Version 5.3 General Programming Concepts: Writing and Debugging Programs.
pthread_cleanup_pop or pthread_cleanup_push Subroutine

Purpose
Activates and deactivates thread cancellation handlers.

Library
Threads Library (libpthreads.a)

Syntax
```
#include <pthread.h>

void pthread_cleanup_pop (execute);

void pthread_cleanup_push (routine, arg);
void (*routine)(void *);
void *arg;
```

Description
The pthread_cleanup_push subroutine pushes the specified cancellation cleanup handler routine onto the calling thread’s cancellation cleanup stack. The cancellation cleanup handler is popped from the cancellation cleanup stack and invoked with the argument arg when: (a) the thread exits (that is, calls pthread_exit, (b) the thread acts upon a cancellation request, or (c) the thread calls pthread_cleanup_pop with a nonzero execute argument.

The pthread_cleanup_pop subroutine removes the subroutine at the top of the calling thread’s cancellation cleanup stack and optionally invokes it (if execute is nonzero).

These subroutines may be implemented as macros and will appear as statements and in pairs within the same lexical scope (that is, the pthread_cleanup_push macro may be thought to expand to a token list whose first token is '{' with pthread_cleanup_pop expanding to a token list whose last token is the corresponding '}').

The effect of calling longjmp or siglongjmp is undefined if there have been any calls to pthread_cleanup_push or pthread_cleanup_pop made without the matching call since the jump buffer was filled. The effect of calling longjmp or siglongjmp from inside a cancellation cleanup handler is also undefined unless the jump buffer was also filled in the cancellation cleanup handler.

Parameters
execute Specifies if the popped subroutine will be executed.
routine Specifies the address of the cancellation subroutine.
arg Specifies the argument passed to the cancellation subroutine.

Related Information
The pthread_cancel (“pthread_cancel Subroutine” on page 1209), pthread_setcancelstate (“pthread_setcancelstate, pthread_setcanceltype, or pthread_testcancel Subroutines” on page 1275) subroutines, the pthread.h file.

Terminating Threads in AIX 5L Version 5.3 General Programming Concepts: Writing and Debugging Programs.
**pthread_cond_destroy or pthread_cond_init Subroutine**

**Purpose**
Initialize and destroys condition variables.

**Library**
Threads Library (libpthreads.a)

**Syntax**
```c
#include <pthread.h>

int pthread_cond_init (cond, attr)
    pthread_cond_t *cond;
    const pthread_condattr_t *attr;

int pthread_cond_destroy (cond)
    pthread_cond_t *cond;

pthread_cond_t cond = PTHREAD_COND_INITIALIZER;
```

**Description**
The function `pthread_cond_init` initializes the condition variable referenced by `cond` with attributes referenced by `attr`. If `attr` is NULL, the default condition variable attributes are used; the effect is the same as passing the address of a default condition variable attributes object. Upon successful initialization, the state of the condition variable becomes initialized.

Attempting to initialize an already initialized condition variable results in undefined behavior.

The function `pthread_cond_destroy` destroys the given condition variable specified by `cond`; the object becomes, in effect, uninitialized. An implementation may cause `pthread_cond_destroy` to set the object referenced by `cond` to an invalid value. A destroyed condition variable object can be re-initialized using `pthread_cond_init`; the results of otherwise referencing the object after it has been destroyed are undefined.

It is safe to destroy an initialized condition variable upon which no threads are currently blocked. Attempting to destroy a condition variable upon which other threads are currently blocked results in undefined behavior.

In cases where default condition variable attributes are appropriate, the macro `PTHREAD_COND_INITIALIZER` can be used to initialize condition variables that are statically allocated. The effect is equivalent to dynamic initialization by a call to `pthread_cond_init` with parameter `attr` specified as NULL, except that no error checks are performed.

**Parameters**
- `cond` Pointer to the condition variable.
- `attr` Specifies the attributes of the condition.

**Return Values**
If successful, the `pthread_cond_init` and `pthread_cond_destroy` functions return zero. Otherwise, an error number is returned to indicate the error. The EBUSY and EINVAL error checks, if implemented, act as if they were performed immediately at the beginning of processing for the function and caused an error return prior to modifying the state of the condition variable specified by `cond`. 
Error Codes

The `pthread_cond_init` function will fail if:

- **EAGAIN**: The system lacked the necessary resources (other than memory) to initialize another condition variable.
- **ENOMEM**: Insufficient memory exists to initialize the condition variable.

The `pthread_cond_init` function may fail if:

- **EINVAL**: The value specified by `attr` is invalid.

The `pthread_cond_destroy` function may fail if:

- **EBUSY**: The implementation has detected an attempt to destroy the object referenced by `cond` while it is referenced (for example, while being used in a `pthread_cond_wait` or `pthread_cond_timedwait` by another thread).
- **EINVAL**: The value specified by `cond` is invalid.

These functions will not return an error code of EINTR.

Related Information

The `pthread_cond_signal` or `pthread_cond_broadcast` subroutine and the `pthread_cond_wait` or `pthread_cond_timedwait` subroutine.

The `pthread.h` file.

Using Condition Variables in AIX 5L Version 5.3 General Programming Concepts: Writing and Debugging Programs.

PTHREAD_COND_INITIALIZER Macro

**Purpose**

Initializes a static condition variable with default attributes.

**Library**

Threads Library (`libpthread.a`)

**Syntax**

```c
#include <pthread.h>
static pthread_cond_t cond = PTHREAD_COND_INITIALIZER;
```

**Description**

The `PTHREAD_COND_INITIALIZER` macro initializes the static condition variable `cond`, setting its attributes to default values. This macro should only be used for static condition variables, since no error checking is performed.

**Note:** The `pthread.h` header file must be the first included file of each source file using the threads library. Otherwise, the `-D_THREAD_SAFE` compilation flag should be used, or the `cc_r` compiler used. In this case, the flag is automatically set.
pthread_cond_signal or pthread_cond_broadcast Subroutine

Purpose
Unblocks one or more threads blocked on a condition.

Library
Threads Library (libpthreads.a)

Syntax
#include <pthread.h>

int pthread_cond_signal (condition)
pthread_cond_t *condition;

int pthread_cond_broadcast (condition)
pthread_cond_t *condition;

Description
These subroutines unblock one or more threads blocked on the condition specified by condition. The
pthread_cond_signal subroutine unblocks at least one blocked thread, while the
pthread_cond_broadcast subroutine unblocks all the blocked threads.

If more than one thread is blocked on a condition variable, the scheduling policy determines the order in
which threads are unblocked. When each thread unblocked as a result of a pthread_cond_signal or
pthread_cond_broadcast returns from its call to pthread_cond_wait or pthread_cond_timedwait, the
thread owns the mutex with which it called pthread_cond_wait or pthread_cond_timedwait. The
thread(s) that are unblocked contend for the mutex according to the scheduling policy (if applicable), and
as if each had called pthread_mutex_lock.

The pthread_cond_signal or pthread_cond_broadcast functions may be called by a thread whether or
not it currently owns the mutex that threads calling pthread_cond_wait or pthread_cond_timedwait have
associated with the condition variable during their waits; however, if predictable scheduling behavior is
required, then that mutex is locked by the thread calling pthread_cond_signal or
pthread_cond_broadcast.

If no thread is blocked on the condition, the subroutine succeeds, but the signalling of the condition is not
held. The next thread calling pthread_cond_wait will be blocked.

Note: The pthread.h header file must be the first included file of each source file using the threads
library. Otherwise, the -D_THREAD_SAFE compilation flag should be used, or the cc_r compiler
used. In this case, the flag is automatically set.

Parameters
condition Specifies the condition to signal.
Return Values
Upon successful completion, 0 is returned. Otherwise, an error code is returned.

Error Code
The `pthread_cond_signal` and `pthread_cond_broadcast` subroutines are unsuccessful if the following is true:

EINVAL       The `condition` parameter is not valid.

Related Information
The `pthread_cond_wait` or `pthread_cond_timedwait` subroutine.

Using Condition Variables in AIX 5L Version 5.3 General Programming Concepts: Writing and Debugging Programs.

pthread_cond_wait or pthread_cond_timedwait Subroutine

Purpose
Blocks the calling thread on a condition.

Library
Threads Library (libpthreads.a)

Syntax
```c
#include <pthread.h>

int pthread_cond_wait (cond, mutex);
pthread_cond_t *cond;
pthread_mutex_t *mutex;

int pthread_cond_timedwait (cond, mutex, timeout);
pthread_cond_t *cond;
pthread_mutex_t *mutex;
const struct timespec *timeout;
```

Description
The `pthread_cond_wait` and `pthread_cond_timedwait` functions are used to block on a condition variable. They are called with `mutex` locked by the calling thread or undefined behavior will result.

These functions atomically release `mutex` and cause the calling thread to block on the condition variable `cond`; atomically here means atomically with respect to access by another thread to the mutex and then the condition variable. That is, if another thread is able to acquire the mutex after the about-to-block thread has released it, then a subsequent call to `pthread_cond_signal` or `pthread_cond_broadcast` in that thread behaves as if it were issued after the about-to-block thread has blocked.

Upon successful return, the mutex is locked and owned by the calling thread.

When using condition variables there is always a boolean predicate involving shared variables associated with each condition wait that is true if the thread should proceed. Spurious wakeups from the `pthread_cond_wait` or `pthread_cond_timedwait` functions may occur. Since the return from `pthread_cond_wait` or `pthread_cond_timedwait` does not imply anything about the value of this predicate, the predicate should be reevaluated upon such return.
The effect of using more than one mutex for concurrent `pthread_cond_wait` or `pthread_cond_timedwait` operations on the same condition variable is undefined; that is, a condition variable becomes bound to a unique mutex when a thread waits on the condition variable, and this (dynamic) binding ends when the wait returns.

A condition wait (whether timed or not) is a cancellation point. When the cancelability enable state of a thread is set to PTHREAD_CANCEL_DEFERRED, a side effect of acting upon a cancellation request while in a condition wait is that the mutex is (in effect) reacquired before calling the first cancellation cleanup handler. The effect is as if the thread were unblocked, allowed to execute up to the point of returning from the call to `pthread_cond_wait` or ` pthread_cond_timedwait`, but at that point notices the cancellation request and instead of returning to the caller of `pthread_cond_wait` or `pthread_cond_timedwait`, starts the thread cancellation activities, which includes calling cancellation cleanup handlers.

A thread that has been unblocked because it has been canceled while blocked in a call to `pthread_cond_wait` or `pthread_cond_timedwait` does not consume any condition signal that may be directed concurrently at the condition variable if there are other threads blocked on the condition variable.

The ` pthread_cond_timedwait` function is the same as ` pthread_cond_wait` except that an error is returned if the absolute time specified by `timeout` passes (that is, system time equals or exceeds `timeout`) before the condition `cond` is signaled or broadcast, or if the absolute time specified by `timeout` has already been passed at the time of the call. When such time-outs occur, ` pthread_cond_timedwait` will nonetheless release and reacquire the mutex referenced by `mutex`. The function ` pthread_cond_timedwait` is also a cancellation point. The absolute time specified by `timeout` can be either based on the system realtime clock or the system monotonic clock. The reference clock for the condition variable is set by calling `pthread_condattr_setclock` before its initialization with the corresponding condition attributes object.

If a signal is delivered to a thread waiting for a condition variable, upon return from the signal handler the thread resumes waiting for the condition variable as if it was not interrupted, or it returns zero due to spurious wakeup.

**Parameters**

- `cond`: Specifies the condition variable to wait on.
- `mutex`: Specifies the mutex used to protect the condition variable. The mutex must be locked when the subroutine is called.
- `timeout`: Points to the absolute time structure specifying the blocked state timeout.

**Return Values**

Except in the case of ETIMEDOUT, all these error checks act as if they were performed immediately at the beginning of processing for the function and cause an error return, in effect, prior to modifying the state of the mutex specified by `mutex` or the condition variable specified by `cond`.

Upon successful completion, a value of zero is returned. Otherwise, an error number is returned to indicate the error.

**Error Codes**

The ` pthread_cond_timedwait` function will fail if:

- **ETIMEDOUT**: The time specified by `timeout` to ` pthread_cond_timedwait` has passed.
The `pthread_cond_wait` and `pthread_cond_timedwait` functions may fail if:

**EINVAL** The value specified by `cond`, `mutex`, or `timeout` is invalid.

**EINVAL** Different mutexes were supplied for concurrent `pthread_cond_wait` or `pthread_cond_timedwait` operations on the same condition variable.

**EINVAL** The mutex was not owned by the current thread at the time of the call.

**EPERM** The mutex was not owned by the current thread at the time of the call, XPG_SUS_ENV is set to ON, and XPG_UNIX98 is not set.

These functions will not return an error code of EINTR.

**Related Information**

The `pthread_cond_signal` or `pthread_cond_broadcast` subroutine, "pthread_cond_signal or pthread_cond_broadcast Subroutine" on page 1214, subroutine, "pthread_condattr_getclock, pthread_condattr_setclock Subroutine" on page 1218, the `pthread.h` file.

Using Condition Variables in AIX 5L Version 5.3 General Programming Concepts: Writing and Debugging Programs.

---

**pthread_condattr_destroy or pthread_condattr_init Subroutine**

**Purpose**

Initializes and destroys condition variable.

**Library**

Threads Library (`libpthreads.a`)

**Syntax**

```c
#include <pthread.h>

int pthread_condattr_destroy (attr)
    pthread_condattr_t *attr;

int pthread_condattr_init (attr)
    pthread_condattr_t *attr;
```

**Description**

The function `pthread_condattr_init` initializes a condition variable attributes object `attr` with the default value for all of the attributes defined by the implementation. Attempting to initialize an already initialized condition variable attributes object results in undefined behavior.

After a condition variable attributes object has been used to initialize one or more condition variables, any function affecting the attributes object (including destruction) does not affect any previously initialized condition variables.

The `pthread_condattr_destroy` function destroys a condition variable attributes object; the object becomes, in effect, uninitialized. The `pthread_condattr_destroy` subroutine may set the object referenced by `attr` to an invalid value. A destroyed condition variable attributes object can be re-initialized using `pthread_condattr_init`; the results of otherwise referencing the object after it has been destroyed are undefined.
Parameter

attr  Specifies the condition attributes object to delete.

Return Values

If successful, the `pthread_condattr_init` and `pthread_condattr_destroy` functions return zero. Otherwise, an error number is returned to indicate the error.

Error Code

The `pthread_condattr_init` function will fail if:

- `ENOMEM`  Insufficient memory exists to initialize the condition variable attributes object.

The `pthread_condattr_destroy` function may fail if:

- `EINVAL`  The value specified by `attr` is invalid.

These functions will not return an error code of EINTR.

Related Information

The `pthread_cond_init` subroutine, `pthread_condattr_getpshared` subroutine, `pthread_create` subroutine, `pthread_mutex_init` subroutine.

The `pthread.h` file.

Using Condition Variables in AIX 5L Version 5.3 General Programming Concepts: Writing and Debugging Programs.

`pthread_condattr_getclock`, `pthread_condattr_setclock` Subroutine

Purpose

Gets and sets the clock selection condition variable attribute.

Syntax

```c
int pthread_condattr_getclock(const pthread_condattr_t *restrict attr, 
clockid_t *restrict clock_id);
int pthread_condattr_setclock(pthread_condattr_t *attr, 
clockid_t clock_id);
```

Description

The `pthread_condattr_getclock` subroutine obtains the value of the clock attribute from the attributes object referenced by the `attr` argument. The `pthread_condattr_setclock` subroutine sets the clock attribute in an initialized attributes object referenced by the `attr` argument. If `pthread_condattr_setclock` is called with a `clock_id` argument that refers to a CPU-time clock, the call will fail.

The clock attribute is the clock ID of the clock that shall be used to measure the timeout service of the `pthread_cond_timedwait` subroutine. The default value of the clock attribute refers to the system clock.
Parameters

attr  Specifies the condition attributes object.
clock_id  For pthread_condattr_getclock(), points to where the clock attribute value will be stored.
          For pthread_condattr_setclock(), specifies the clock to set. Valid values are:

          CLOCK_REALTIME
          The system realtime clock.

          CLOCK_MONOTONIC
          The system monotonic clock. The value of this clock represents the amount of time since
          an unspecified point in the past. The value of this clock always grows: it cannot be set by
          clock_settime() and cannot have backward clock jumps.

Return Values

If successful, the pthread_condattr_getclock subroutine returns 0 and stores the value of the clock
attribute of attr in the object referenced by the clock_id argument. Otherwise, an error code is returned to
indicate the error.

If successful, the pthread_condattr_setclock subroutine returns 0; otherwise, an error code is returned to
indicate the error.

Error Codes

EINVAL  The value specified by attr is invalid.
EINVAL  The pthread_condattr_setclock subroutine returns this error if the value specified by the clock_id
does not refer to a known clock, or is a CPU-time clock.
ENOTSUP  The function is not supported with checkpoint-restart processes.

Related Information

"pthread_cond_destroy or pthread_cond_init Subroutine" on page 1212, "pthread_cond_wait or
pthread_cond_timedwait Subroutine" on page 1215, "pthread_condattr_getpshared Subroutine,"
"pthread_condattr_destroy or pthread_condattr_init Subroutine" on page 1217,
"pthread_condattr_setpshared Subroutine" on page 1221, "pthread_create Subroutine" on page 1222,
"pthread_mutex_init or pthread_mutex_destroy Subroutine" on page 1246.

The pthread.h file.


pthread_condattr_getpshared Subroutine

Purpose

Returns the value of the pshared attribute of a condition attributes object.

Library

Threads Library (libpthread.a)
Syntax

```c
#include <pthread.h>

int pthread_condattr_getpshared (attr, pshared);
const pthread_condattr_t *attr;
int *pshared;
```

Description

The `pthread_condattr_getpshared` subroutine returns the value of the pshared attribute of the condition attribute object `attr`. This attribute specifies the process sharing of the condition variable created with this attributes object. It may have one of the following values:

- **PTHREAD_PROCESS_SHARED**: Specifies that the condition variable can be used by any thread that has access to the memory where it is allocated, even if these threads belong to different processes.
- **PTHREAD_PROCESS_PRIVATE**: Specifies that the condition variable shall only be used by threads within the same process as the thread that created it. This is the default value.

**Note**: The `pthread.h` header file must be the first included file of each source file using the threads library. Otherwise, the `-D_THREAD_SAFE` compilation flag should be used, or the cc_r compiler used. In this case, the flag is automatically set.

Parameters

- **attr**: Specifies the condition attributes object.
- **pshared**: Points to where the pshared attribute value will be stored.

Return Values

Upon successful completion, the value of the pshared attribute is returned via the `pshared` parameter, and 0 is returned. Otherwise, an error code is returned.

Error Codes

The `pthread_condattr_getpshared` subroutine is unsuccessful if the following is true:

- **EINVAL**: The `attr` parameter is not valid.
- **ENOSYS**: The process sharing POSIX option is not implemented.

Related Information

The `pthread_condattr_setpshared` subroutine, `pthread_condattr_init` subroutine, `pthread_condattr_destroy` subroutine, `pthread_cond_destroy` subroutine, or `pthread_cond_init` subroutine.

**Advanced Attributes** in AIX 5L Version 5.3 General Programming Concepts: Writing and Debugging Programs.

**Threads Library Options** in AIX 5L Version 5.3 General Programming Concepts: Writing and Debugging Programs.
**pthread_condattr_setpshared Subroutine**

**Purpose**
Sets the value of the pshared attribute of a condition attributes object.

**Library**
Threads Library (libpthreads.a)

**Syntax**
```c
#include <pthread.h>

int pthread_condattr_setpshared (attr, pshared);
```

**Description**
The `pthread_condattr_setpshared` subroutine sets the value of the pshared attribute of the condition attributes object `attr`. This attribute specifies the process sharing of the condition variable created with this attributes object.

**Note:** The `pthread.h` header file must be the first included file of each source file using the threads library. Otherwise, the `-D_THREAD_SAFE` compilation flag should be used, or the cc_r compiler used. In this case, the flag is automatically set.

**Parameters**
- `attr` Specifies the condition attributes object.
- `pshared` Specifies the process sharing to set. It must have one of the following values:
  - `PTHREAD_PROCESS_SHARED` Specifies that the condition variable can be used by any thread that has access to the memory where it is allocated, even if these threads belong to different processes.
  - `PTHREAD_PROCESS_PRIVATE` Specifies that the condition variable shall only be used by threads within the same process as the thread that created it. This is the default value.

**Return Values**
Upon successful completion, 0 is returned. Otherwise, an error code is returned.

**Error Codes**
The `pthread_condattr_setpshared` subroutine is unsuccessful if the following is true:
- `EINVAL` The `attr` or `pshared` parameters are not valid.

**Related Information**
The `pthread_condattr_getpshared` subroutine, `pthread_condattr_init` or `pthread_cond_init` subroutine, `pthread_condattr_destroy or pthread_condattr_init` subroutine.

Advanced Attributes in AIX 5L Version 5.3 General Programming Concepts: Writing and Debugging Programs.
Threads Library Options in AIX 5L Version 5.3 General Programming Concepts: Writing and Debugging Programs.

pthread_create Subroutine

Purpose
Creates a new thread, initializes its attributes, and makes it runnable.

Library
Threads Library (libpthreads.a)

Syntax
```
#include <pthread.h>

int pthread_create (thread, attr, start_routine (void), arg)
pthread_t *thread;
const pthread_attr_t *attr;
void **start_routine (void);
void *arg;
```

Description
The pthread_create subroutine creates a new thread and initializes its attributes using the thread attributes object specified by the attr parameter. The new thread inherits its creating thread's signal mask; but any pending signal of the creating thread will be cleared for the new thread.

The new thread is made runnable, and will start executing the start_routine routine, with the parameter specified by the arg parameter. The arg parameter is a void pointer; it can reference any kind of data. It is not recommended to cast this pointer into a scalar data type (int for example), because the casts may not be portable.

After thread creation, the thread attributes object can be reused to create another thread, or deleted.

The thread terminates in the following cases:
• The thread returned from its starting routine (the main routine for the initial thread)
• The thread called the pthread_exit subroutine
• The thread was canceled
• The thread received a signal that terminated it
• The entire process is terminated due to a call to either the exec subroutine or exit subroutine.

Note: The pthread.h header file must be the first included file of each source file using the threads library. Otherwise, the -D_THREAD_SAFE compilation flag should be used, or the cc_r compiler used. In this case, the flag is automatically set.

When multiple threads are created in a process, the FULL_CORE flag is set for all signals. This means that if a core file is produced, it will be much bigger than a single_threaded application. This is necessary to debug multi-threaded processes.

When a process uses the pthread_create function, and thus becomes multi-threaded, the FULL_CORE flag is enabled for all signals. If a signal is received whose action is to terminate the process with a core dump, a full dump (usually much larger than a regular dump) will be produced. This is necessary so that multi-threaded programs can be debugged with the dbx command.
The following piece of pseudocode is an example of how to avoid getting a full core. Please note that in this case, debug will not be possible. It may be easier to limit the size of the core with the ulimit command.

```c
struct sigaction siga;
siga.sa_handler = SIG_DFL;
siga.sa_flags = SA_RESTART;
SIGINITSET(siga.as_mask);
sigaction(<SIGNAL_NUMBER>, &siga, NULL);
```

The alternate stack is not inherited.

**Parameters**

- `thread`: Points to where the thread ID will be stored.
- `attr`: Specifies the thread attributes object to use in creating the thread. If the value is `NULL`, the default attributes values will be used.
- `start_routine`: Points to the routine to be executed by the thread.
- `arg`: Points to the single argument to be passed to the `start_routine` routine.

**Return Values**

If successful, the `pthread_create` function returns zero. Otherwise, an error number is returned to indicate the error.

**Error Codes**

The `pthread_create` function will fail if:

- `EAGAIN`: If WLM is running, the limit on the number of threads in the class may have been met.
- `EINVAL`: The value specified by `attr` is invalid.
- `EPERM`: The caller does not have appropriate permission to set the required scheduling parameters or scheduling policy.

The `pthread_create` function will not return an error code of `EINTR`.

**Related Information**

The `core` file format.

The `dbx` and `ulimit` commands.

The `pthread_attr_init` subroutine, `pthread_attr_destroy` subroutine, `pthread_attr_setinheritsched` subroutine, `pthread_attr_setstacksize` subroutine, `pthread_create` subroutine, `pthread_exit` subroutine, `pthread_cancel` subroutine, `pthread_detach` subroutine, `pthread_self` subroutine, `pthread_once` subroutine, `pthread_kill` subroutine, `pthread_join` subroutine, `pthread_threadid` subroutine, `pthread_setschedparam` subroutine, `pthread_setschedpolicy` subroutine, and the `pthread.h` file.

Creating Threads in AIX 5L Version 5.3 General Programming Concepts: Writing and Debugging Programs.
**pthread_create_withcred_np Subroutine**

**Purpose**
Creates a new thread with a new set of credentials, initializes its attributes, and makes it runnable.

**Library**
Threads Library (libpthreads.a)

**Syntax**
```c
#include <pthread.h>
#include <sys/cred.h>

int pthread_create_withcred_np(pthread_t *thread, const pthread_attr_t *attr,
void *(*start_routine)(void),
void *arg, struct __pthrdscreds *credp)
```

**Description**
The `pthread_create_withcred_np` subroutine is exactly equivalent to the `pthread_create` routine except that it allows the new thread to be created and start running with the credentials specified by the `credp` parameter. Only a process that has the credentials capability or is running with an effective user ID as the root user is allowed to modify its credentials using this routine.

The following credentials can be modified:
- Effective, real and saved user IDs
- Effective, real and saved group IDs
- Supplementary group IDs

**Note:** The administrator can set the lowest user ID value to which a process with credentials capability is allowed to switch its user IDs. A value of 0 can be specified for any of the preceding credentials to indicate that the thread should inherit that specific credential from its caller. The administrator can also set the lowest group ID to which a process with credentials capability is allowed to switch its group IDs.

The `__pc_flags` flag field in the `credp` parameter provides options to inherit credentials from the parent thread.

The newly created thread runs with per-thread credentials, and system calls like `getuid` or `getgid` returns the thread’s credentials. Similarly, when a file is opened or a message is received, the thread’s credentials will be used to determine whether the thread has privilege to execute the operation.

**Parameters**
- **thread**: Points to where the thread ID will be stored.
- **attr**: Specifies the thread attributes object to use in creating the thread. If the value is NULL, the default attributes values will be used.
- **start_routine**: Points to the routine to be executed by the thread.
- **arg**: Points to the single argument to be passed to the `start_routine` routine.
Points to a structure of type __pthrdscreds, which contains the credentials structure and the inheritance flags. If set to NULL, the pthread_create_withcred_np subroutine is the same as the pthread_create routine. The __pc_cred field indicates the credentials to be assigned to the new pthread. The __pc_flags field indicates which credentials, if any, are to be inherited from the parent thread. This field is constructed by logically OR’ing one or more of the following values:

- PTHRDSCREDS_INHERIT_UIDS: Inherit user IDs from parent thread.
- PTHRDSCREDS_INHERIT_GIDS: Inherit group IDs from parent thread.
- PTHRDSCREDS_INHERIT_GSETS: Inherit the group sets from parent thread.
- PTHRDSCREDS_INHERIT_PRIVILEGES: Inherit privileges from the parent thread.

**Security**

Only a process that has the credentials capability or is running with an effective user ID (such as the root user) is allowed to modify its credentials using this routine.

**Return Values**

If successful, the pthread_create_withcred_np subroutine returns 0. Otherwise, an error number is returned to indicate the error.

**Error Codes**

- EAGAIN: If WLM is running, the limit on the number of threads in the class might have been met.
- EFAULT: The credp parameter points to a location outside of the allocated address space of the process.
- EINVAL: The credentials specified in the credp parameter are not valid.
- EPERM: The caller does not have appropriate permission to set the credentials.

The pthread_create_withcred_np subroutine will not return an error code of EINTR.

**Related Information**

- "pthread_create Subroutine" on page 1222

**pthread_delay_np Subroutine**

**Purpose**

Causes a thread to wait for a specified period.

**Library**

Threads Library (libpthreads.a)

**Syntax**

```c
#include <pthread.h>

int pthread_delay_np (struct timespec *interval);
```
**Description**

The `pthread_delay_np` subroutine causes the calling thread to delay execution for a specified period of elapsed wall clock time. The period of time the thread waits is at least as long as the number of seconds and nanoseconds specified in the `interval` parameter.

**Notes:**

1. The `pthread.h` header file must be the first included file of each source file using the threads library. Otherwise, the `-D_THREAD_SAFE` compilation flag should be used, or the cc_r compiler used. In this case, the flag is automatically set.
2. The `pthread_delay_np` subroutine is not portable.

This subroutine is not POSIX compliant and is provided only for compatibility with DCE threads. It should not be used when writing new applications.

**Parameters**

`interval` Points to the time structure specifying the wait period.

**Return Values**

Upon successful completion, 0 is returned. Otherwise, an error code is returned.

**Error Codes**

The `pthread_delay_np` subroutine is unsuccessful if the following is true:

- **EINVAL** The `interval` parameter is not valid.

**Related Information**

The `sleep`, `nsleep`, or `usleep` subroutine.

---

**pthread_equal Subroutine**

**Purpose**

Compares two thread IDs.

**Library**

Threads Library (`libpthreads.a`)

**Syntax**

```c
#include <pthread.h>

int pthread_equal (thread1, thread2)
    pthread_t thread1;
    pthread_t thread2;
```

**Description**

The `pthread_equal` subroutine compares the thread IDs `thread1` and `thread2`. Since the thread IDs are opaque objects, it should not be assumed that they can be compared using the equality operator (`==`).
Note: The `pthread.h` header file must be the first included file of each source file using the threads library. Otherwise, the `-D_THREAD_SAFE` compilation flag should be used, or the cc_r compiler used. In this case, the flag is automatically set.

Parameters

- `thread1`: Specifies the first ID to be compared.
- `thread2`: Specifies the second ID to be compared.

Return Values

The `pthread_equal` function returns a nonzero value if `thread1` and `thread2` are equal; otherwise, zero is returned.

If either `thread1` or `thread2` are not valid thread IDs, the behavior is undefined.

Related Information

The `pthread_self` subroutine, the `pthread_create` subroutine, the `pthread.h` file.

Creating Threads in AIX 5L Version 5.3 General Programming Concepts: Writing and Debugging Programs.

pthread_exit Subroutine

Purpose

Terminates the calling thread.

Library

Threads Library (`libpthreads.a`)

Syntax

```c
#include <pthread.h>

void pthread_exit (status);
void *status;
```

Description

The `pthread_exit` subroutine terminates the calling thread safely, and stores a termination status for any thread that may join the calling thread. The termination status is always a void pointer; it can reference any kind of data. It is not recommended to cast this pointer into a scalar data type (int for example), because the casts may not be portable. This subroutine never returns.

Unlike the `exit` subroutine, the `pthread_exit` subroutine does not close files. Thus any file opened and used only by the calling thread must be closed before calling this subroutine. It is also important to note that the `pthread_exit` subroutine frees any thread-specific data, including the thread’s stack. Any data allocated on the stack becomes invalid, since the stack is freed and the corresponding memory may be reused by another thread. Therefore, thread synchronization objects (mutexes and condition variables) allocated on a thread’s stack must be destroyed before the thread calls the `pthread_exit` subroutine.

Returning from the initial routine of a thread implicitly calls the `pthread_exit` subroutine, using the return value as parameter.
If the thread is not detached, its resources, including the thread ID, the termination status, the thread-specific data, and its storage, are all maintained until the thread is detached or the process terminates.

If another thread joins the calling thread, that thread wakes up immediately, and the calling thread is automatically detached.

If the thread is detached, the cleanup routines are popped from their stack and executed. Then the destructor routines from the thread-specific data are executed. Finally, the storage of the thread is reclaimed and its ID is freed for reuse.

Terminating the initial thread by calling this subroutine does not terminate the process, it just terminates the initial thread. However, if all the threads in the process are terminated, the process is terminated by implicitly calling the exit subroutine with a return code of 0 if the last thread is detached, or 1 otherwise.

**Note:** The pthread.h header file must be the first included file of each source file using the threads library. Otherwise, the -D_THREAD_SAFE compilation flag should be used, or the cc_r compiler used. In this case, the flag is automatically set.

**Parameters**

status Points to an optional termination status, used by joining threads. If no termination status is desired, its value should be NULL.

**Return Values**
The pthread_exit function cannot return to its caller.

**Errors**
No errors are defined.

The pthread_exit function will not return an error code of EINTR.

**Related Information**
The pthread_cleanup_push subroutine, pthread_cleanup_pop subroutine, pthread_key_create subroutine, pthread_create subroutine, pthread_join subroutine, pthread_cancel subroutine, exit subroutine, the pthread.h file.

Terminating Threads in AIX 5L Version 5.3 General Programming Concepts: Writing and Debugging Programs.

**pthread_get_expiration_np Subroutine**

**Purpose**
Obtains a value representing a desired expiration time.

**Library**
Threads Library (lib pthread.a)
Syntax
#include <pthread.h>

int pthread_get_expiration_np ( struct timespec *delta, struct timespec *abstime )

Description
The pthread_get_expiration_np subroutine adds the interval delta to the current absolute system time and returns a new absolute time. This new absolute time can be used as the expiration time in a call to the pthread_cond_timedwait subroutine.

Notes:
1. The pthread.h header file must be the first included file of each source file using the threads library. Otherwise, the -D_THREAD_SAFE compilation flag should be used, or the cc_r compiler used. In this case, the flag is automatically set.
2. The pthread_get_expiration_np subroutine is not portable.

This subroutine is not POSIX compliant and is provided only for compatibility with DCE threads. It should not be used when writing new applications.

Parameters
delta Points to the time structure specifying the interval.
abstime Points to where the new absolute time will be stored.

Return Values
Upon successful completion, the new absolute time is returned via the abstime parameter, and 0 is returned. Otherwise, an error code is returned.

Error Codes
The pthread_get_expiration_np subroutine is unsuccessful if the following is true:
EINVAL The delta or abstime parameters are not valid.

Related Information
The pthread_cond_timedwait subroutine.

pthread_getconcurrency or pthread_setconcurrency Subroutine

Purpose
Gets or sets level of concurrency.

Library
Threads Library (libthreads.a)
Syntax

```c
#include <pthread.h>

int pthread_getconcurrency (void);

int pthread_setconcurrency (new_level);
int new_level;
```

Description

The `pthread_setconcurrency` subroutine allows an application to inform the threads implementation of its desired concurrency level, `new_level`. The actual level of concurrency provided by the implementation as a result of this function call is unspecified.

If `new_level` is zero, it causes the implementation to maintain the concurrency level at its discretion as if `pthread_setconcurrency` was never called.

The `pthread_getconcurrency` subroutine returns the value set by a previous call to the `pthread_setconcurrency` subroutine. If the `pthread_setconcurrency` subroutine was not previously called, this function returns zero to indicate that the implementation is maintaining the concurrency level.

When an application calls `pthread_setconcurrency`, it is informing the implementation of its desired concurrency level. The implementation uses this as a hint, not a requirement.

Use of these subroutines changes the state of the underlying concurrency upon which the application depends. Library developers are advised to not use the `pthread_getconcurrency` and `pthread_setconcurrency` subroutines since their use may conflict with an applications use of these functions.

Parameters

- `new_level` Specifies the value of the concurrency level.

Return Value

If successful, the `pthread_setconcurrency` subroutine returns zero. Otherwise, an error number is returned to indicate the error.

The `pthread_getconcurrency` subroutine always returns the concurrency level set by a previous call to `pthread_setconcurrency`. If the `pthread_setconcurrency` subroutine has never been called, `pthread_getconcurrency` returns zero.

Error Codes

The `pthread_setconcurrency` subroutine will fail if:

- EINVAL The value specified by `new_level` is negative.
- EAGAIN The value specific by `new_level` would cause a system resource to be exceeded.

Related Information

The `pthread.h` file.
pthread_getcpuclockid Subroutine

Purpose
Accesses a thread CPU-time clock.

Syntax
#include <pthread.h>
#include <time.h>

int pthread_getcpuclockid(pthread_t thread_id, clockid_t *clock_id);

Description
The pthread_getcpuclockid subroutine returns in the clock_id parameter the clock ID of the CPU-time clock of the thread specified by thread_id, if the thread specified by thread_id exists.

Parameters

thread_id Specifies the ID of the pthread whose clock ID is requested.
clock_id Points to the clockid_t structure used to return the thread CPU-time clock ID of thread_id.

Return Values
Upon successful completion, the pthread_getcpuclockid subroutine returns 0; otherwise, an error number is returned to indicate the error.

Error Codes

ENOTSUP The subroutine is not supported with checkpoint-restart'ed processes.
ESRCH The value specified by thread_id does not refer to an existing thread.

Related Information
"clock_getcpuclockid Subroutine" on page 169, "clock_getres, clock_gettime, and clock_settime Subroutine" on page 170, timer_create Subroutine, timer_gettime Subroutine

pthread_getrusage_np Subroutine

Purpose
Enable or disable pthread library resource collection, and retrieve resource information for any pthread in the current process.

Library
Threads Library (libpthreads.a)

Syntax
#include <pthread.h>

int pthread_getrusage_np(pthread_t Ptid, struct rusage *Rusage, int Mode);

Base Operating System (BOS) Runtime Services (A-P) 1231
Description

The **pthread_getrusage_np** subroutine enables and disables resource collection in the pthread library and collects resource information for any pthread in the current process. When compiled in 64-bit mode, resource usage (rusage) counters are 64-bits for the calling thread. When compiled in 32-bit mode, rusage counters are 32-bits for the calling pthread.

This functionality is enabled by default. The previous **AIXTHREAD_ENRUSG** used with **pthread_getrusage_np** is no longer supported.

Parameters

\[Ptid\]

Specifies the target thread. Must be within the current process.
Points to a buffer described in the `/usr/include/sys/resource.h` file. The fields are defined as follows:

- **ru_utime**
  The total amount of time running in user mode.

- **ru_stime**
  The total amount of time spent in the system executing on behalf of the processes.

- **ru_maxrss**
  The maximum size, in kilobytes, of the used resident set size.

- **ru_ixrss**
  An integral value indicating the amount of memory used by the text segment that was also shared among other processes. This value is expressed in *units of kilobytes X seconds-of-execution* and is calculated by adding the number of shared memory pages in use each time the internal system clock ticks, and then averaging over one-second intervals.

- **ru_idrss**
  An integral value of the amount of unshared memory in the data segment of a process, which is expressed in *units of kilobytes X seconds-of-execution*.

- **ru_minflt**
  The number of page faults serviced without any I/O activity. In this case, I/O activity is avoided by reclaiming a page frame from the list of pages awaiting reallocation.

- **ru_majflt**
  The number of page faults serviced that required I/O activity.

- **ru_nswap**
  The number of times that a process was swapped out of main memory.

- **ru_inblock**
  The number of times that the file system performed input.

- **ru_oublock**
  The number of times that the file system performed output.

  **Note:** The numbers that the `ru_inblock` and `ru_oublock` fields display account for real I/O only; data supplied by the caching mechanism is charged only to the first process that reads or writes the data.

- **ru_msgsnd**
  The number of IPC messages sent.

- **ru_msgrcv**
  The number of IPC messages received.

- **ru_nsignals**
  The number of signals delivered.

- **ru_nvcsw**
  The number of times a context switch resulted because a process voluntarily gave up the processor before its time slice was completed. This usually occurs while the process waits for a resource to become available.

- **ru_nivcsw**
  The number of times a context switch resulted because a higher priority process ran or because the current process exceeded its time slice.
Mode Indicates which task the subroutine should perform. Acceptable values are as follows:

PTHRDSINFO_RUSAGE_START
Returns the current resource utilization, which will be the start measurement.

PTHRDSINFO_RUSAGE_STOP
Returns total current resource utilization since the last time a
PTHRDSINFO_RUSAGE_START was performed. If the task
PTHRDSINFO_RUSAGE_START was not performed, then the resource information
returned is the accumulated value since the start of the pthread.

PTHRDSINFO_RUSAGE_COLLECT
Collects resource information for the target thread. If the task
PTHRDSINFO_RUSAGE_START was not performed, then the resource information
returned is the accumulated value since the start of the pthread.

Return Values
Upon successful completion, the pthread_getrusage_np subroutine returns a value of 0. Otherwise, an
error number is returned to indicate the error.

Error Codes
The pthread_getrusage_np subroutine fails if:
EINVAL The address specified for RUsage is NULL, not valid, or a null value for Ptid was given.
ESRCH Either no thread could be found corresponding to the ID thread of the Ptid thread or the thread
corresponding to the Ptid thread ID was not in the current process.

Related Information
The pthread.h subroutine.

pthread_getschedparam Subroutine

Purpose
Returns the current schedpolicy and schedparam attributes of a thread.

Library
Threads Library (libpthread.a)

Syntax
#include <pthread.h>
#include <sys/sched.h>

int pthread_getschedparam (thread, schedpolicy, schedparam)
pthread_t thread;
int *schedpolicy;
struct sched_param *schedparam;

Description
The pthread_getschedparam subroutine returns the current schedpolicy and schedparam attributes of the
thread thread. The schedpolicy attribute specifies the scheduling policy of a thread. It may have one of the
following values:

SCHED_FIFO Denotes first-in first-out scheduling.
SCHED_RR  Denotes round-robin scheduling.
SCHED_OTHER Denotes the default operating system scheduling policy. It is the default value.

The schedparam attribute specifies the scheduling parameters of a thread created with this attributes object. The sched_priority field of the sched_param structure contains the priority of the thread. It is an integer value.

Note: The pthread.h header file must be the first included file of each source file using the threads library. Otherwise, the -D_THREAD_SAFE compilation flag should be used, or the cc_r compiler used. In this case, the flag is automatically set.

The implementation of this subroutine is dependent on the priority scheduling POSIX option. The priority scheduling POSIX option is implemented in the operating system.

Parameters

- **thread**  Specifies the target thread.
- **schedpolicy**  Points to where the schedpolicy attribute value will be stored.
- **schedparam**  Points to where the schedparam attribute value will be stored.

Return Values

Upon successful completion, the current value of the schedpolicy and schedparam attributes are returned via the schedpolicy and schedparam parameters, and 0 is returned. Otherwise, an error code is returned.

Error Codes

The pthread_getschedparam subroutine is unsuccessful if the following is true:

- **ESRCH**  The thread thread does not exist.

Related Information

The pthread_attr_getschedparam subroutine.

Threads Scheduling in AIX 5L Version 5.3 General Programming Concepts: Writing and Debugging Programs.

Threads Library Options in AIX 5L Version 5.3 General Programming Concepts: Writing and Debugging Programs.

pthread_getspecific or pthread_setspecific Subroutine

Purpose

Returns and sets the thread-specific data associated with the specified key.

Library

Threads Library (libpthreads.a)

Syntax

```
#include <pthread.h>
void *pthread_getspecific (key)
pthread_key_t key;
```
int pthread_setspecific (key, value)
pthread_key_t key;
const void *value;

Description
The **pthread_setspecific** function associates a thread-specific value with a key obtained via a previous call to **pthread_key_create**. Different threads may bind different values to the same key. These values are typically pointers to blocks of dynamically allocated memory that have been reserved for use by the calling thread.

The **pthread_getspecific** function returns the value currently bound to the specified key on behalf of the calling thread.

The effect of calling **pthread_setspecific** or **pthread_getspecific** with a key value not obtained from **pthread_key_create** or after key has been deleted with **pthread_key_delete** is undefined.

Both **pthread_setspecific** and **pthread_getspecific** may be called from a thread-specific data destructor function. However, calling **pthread_setspecific** from a destructor may result in lost storage or infinite loops.

Parameters

- **key** Specifies the key to which the value is bound.
- **value** Specifies the new thread-specific value.

Return Values
The function **pthread_getspecific** returns the thread-specific data value associated with the given key. If no thread-specific data value is associated with key, then the value NULL is returned. If successful, the **pthread_setspecific** function returns zero. Otherwise, an error number is returned to indicate the error.

Error Codes
The **pthread_setspecific** function will fail if:

- **ENOMEM** Insufficient memory exists to associate the value with the key.

The **pthread_setspecific** function may fail if:

- **EINVAL** The key value is invalid.

No errors are returned from **pthread_getspecific**.

These functions will not return an error code of **EINTR**.

Related Information
The **pthread_key_create** (**pthread_key_create Subroutine** on page 1242) subroutine, the **pthread.h** file. [Thread-Specific Data](#) in AIX 5L Version 5.3 General Programming Concepts: Writing and Debugging Programs.
pthread_getthrds_np Subroutine

Purpose
Retrieves register and stack information for threads.

Library
Threads Library (libpthreads.a)

Syntax
#include <pthread.h>

int pthread_getthrds_np (thread, mode, buf, bufsize, regbuf, regbufsize)

pthread_t *ptid;
int mode;
struct __pthrdsinfo *buf;
int bufsize;
void *regbuf;
int *regbufsize;

Description
The pthread_getthrds_np subroutine retrieves information on the state of the thread thread and its underlying kernel thread, including register and stack information.

Parameters

thread The pointer to the thread. On input it identifies the target thread of the operation, or 0 to operate on the first entry in the list of threads. On output it identifies the next entry in the list of threads, or 0 if the end of the list has been reached. pthread_getthrds_np can be used to traverse the whole list of threads by starting with thread pointing to 0 and calling pthread_getthrds_np repeatedly until it returns with thread pointing to 0.
Specifies the type of query. These values can be bitwise or'ed together to specify more than one type of query.

PTHRDSINFO_QUERY_GPRS  
get general purpose registers

PTHRDSINFO_QUERY_SPRS  
get special purpose registers

PTHRDSINFO_QUERY_FPRS  
get floating point registers

PTHRDSINFO_QUERY_REGS  
get all of the above registers

PTHRDSINFO_QUERY_TID  
get the kernel thread id

PTHRDSINFO_QUERY_TLS  
get the thread-local storage information.

This value can be or'ed with any value of the mode parameter. The thread-local storage information is returned to the caller in a caller-provided buffer, regbuf. If the buffer is too small for the data, the buffer is filled up to the end of the buffer and ERANGE is returned. The caller also provides the size of the buffer, regbufsize, which on return is changed to the size of the thread local storage information even if it does not fit into a buffer.

The thread-local storage information is returned in form of an array of touplets: memory address and TLS region (unique number assigned by the loader). The TLS region is also included in the loader info structure returned by loadquery. If you need any additional information such as TLS size, you can find it in that structure.

```c
typedef struct __pthrdstlsinfo{
    void *pti_vaddr;
    int pti_region;
} PTHREADS_TLS_INFO;
```

PTHRDSINFO_QUERY_EXTCTX  
get the extended machine context

PTHRDSINFO_QUERY_ALL  
get everything (except for the extended context, which must be explicitly requested)
buf

Specifies the address of the struct __pthrdsinfo structure that will be filled in by
pthread_getthrds_np. On return, this structure holds the following data (depending on the type
of query requested):

__pi_ptid
The thread’s thread identifier

__pi_tid
The thread’s kernel thread id, or 0 if the thread does not have a kernel thread

__pi_state
The state of the thread, equal to one of the following:

PTHRDSINFO_STATE_RUN
The thread is running

PTHRDSINFO_STATE_READY
The thread is ready to run

PTHRDSINFO_STATE_IDLE
The thread is being initialized

PTHRDSINFO_STATE_SLEEP
The thread is sleeping

PTHRDSINFO_STATE_TERM
The thread is terminated

PTHRDSINFO_STATE_NOTSUP
Error condition

__pi_suspended
1 if the thread is suspended, 0 if it is not

__pi_returned
The return status of the thread

__pi_ustk
The thread’s user stack pointer

__pi_context
The thread’s context (register information)

If the PTHRDSINFO_QUERY_EXTCTX mode is requested, then the buf specifies the address of
a __pthrdsinfox structure, which, in addition to all of the preceding information, also contains the
following:

__pi_ec
The thread’s extended context (extended register state)

bufsize
The size of the __pthrdsinfo or __pthrdsinfox structure in bytes.

regbuf
The location of the buffer to hold the register save data and to pass the TLS information from the
kernel if the thread is in a system call.

regbufsize
The pointer to the size of the regbuf buffer. On input, it identifies the maximum size of the buffer
in bytes. On output, it identifies the number of bytes of register save data and pass the TLS
information. If the thread is not in a system call, there is no register save data returned from the
kernel, and regbufsize is 0. If the size of the register save data is larger than the input value of
regbufsize, the number of bytes specified by the input value of regbufsize is copied to regbuf,

pthread_getthrds_n() returns ERANGE, and the output value of regbufsize specifies the
number of bytes required to hold all of the register save data.

Return Values

If successful, the pthread_getthrds_np function returns zero. Otherwise, an error number is returned to
indicate the error.
Error Codes
The `pthread_getthrds_np` function will fail if:

- **EINVAL**: Either thread or buf is NULL, or bufsize is not equal to the size of the __pthrdsinfo structure in the library.
- **ESRCH**: No thread could be found corresponding to that specified by the thread ID thread.
- **ERANGE**: regbuf was not large enough to handle all of the register save data.
- **ENOMEM**: Insufficient memory exists to perform this operation.

Related Information
The `pthread.h` file.

pthread_getunique_np Subroutine

Purpose
Returns the sequence number of a thread.

Library
Threads Library (libpthreads.a)

Syntax
```c
#include <pthread.h>

#include <pthread.h>

int pthread_getunique_np (thread, sequence)
pthread_t *thread;
int *sequence;
```

Description
The `pthread_getunique_np` subroutine returns the sequence number of the thread thread. The sequence number is a number, unique to each thread, associated with the thread at creation time.

Notes:
1. The `pthread.h` header file must be the first included file of each source file using the threads library. Otherwise, the `-D_THREAD_SAFE` compilation flag should be used, or the cc_r compiler used. In this case, the flag is automatically set.
2. The `pthread_getunique_np` subroutine is not portable.

This subroutine is not POSIX compliant and is provided only for compatibility with DCE threads. It should not be used when writing new applications.

Parameters
- `thread` Specifies the thread.
- `sequence` Points to where the sequence number will be stored.

Return Values
Upon successful completion, the sequence number is returned via the `sequence` parameter, and 0 is returned. Otherwise, an error code is returned.
Error Codes
The `pthread_getunique_np` subroutine is unsuccessful if the following is true:

<table>
<thead>
<tr>
<th>Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EINVAL</td>
<td>The <code>thread</code> or <code>sequence</code> parameters are not valid.</td>
</tr>
<tr>
<td>ESRCH</td>
<td>The thread <code>thread</code> does not exist.</td>
</tr>
</tbody>
</table>

Related Information
The `pthread_self` ([`pthread_self` Subroutine” on page 1274]) subroutine.

**pthread_join or pthread_detach Subroutine**

**Purpose**
Blocks or detaches the calling thread until the specified thread terminates.

**Library**
Threads Library (`libpthreads.a`)

**Syntax**
```
#include <pthread.h>

int pthread_join (int thread, void **status);

int pthread_detach (int thread);
```

**Description**
The `pthread_join` subroutine blocks the calling thread until the thread `thread` terminates. The target thread’s termination status is returned in the `status` parameter.

If the target thread is already terminated, but not yet detached, the subroutine returns immediately. It is impossible to join a detached thread, even if it is not yet terminated. The target thread is automatically detached after all joined threads have been woken up.

This subroutine does not itself cause a thread to be terminated. It acts like the `pthread_cond_wait` subroutine to wait for a special condition.

**Note:** The `pthread.h` header file must be the first included file of each source file using the threads library. Otherwise, the `-D_THREAD_SAFE` compilation flag should be used, or the cc_r compiler used. In this case, the flag is automatically set.

The `pthread_detach` subroutine is used to indicate to the implementation that storage for the thread whose thread ID is in the location `thread` can be reclaimed when that thread terminates. This storage shall be reclaimed on process exit, regardless of whether the thread has been detached or not, and may include storage for `thread` return value. If `thread` has not yet terminated, `pthread_detach` shall not cause it to terminate. Multiple `pthread_detach` calls on the same target thread causes an error.

**Parameters**

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>thread</code></td>
<td>Specifies the target thread.</td>
</tr>
<tr>
<td><code>status</code></td>
<td>Points to where the termination status of the target thread will be stored. If the value is NULL, the termination status is not returned.</td>
</tr>
</tbody>
</table>
Return Values
If successful, the `pthread_join` function returns zero. Otherwise, an error number is returned to indicate the error.

Error Codes
The `pthread_join` and `pthread_detach` functions will fail if:

- **EINVAL** The implementation has detected that the value specified by `thread` does not refer to a joinable thread.
- **ESRCH** No thread could be found corresponding to that specified by the given thread ID.

The `pthread_join` function will fail if:

- **EDEADLK** The value of `thread` specifies the calling thread.

The `pthread_join` function will not return an error code of **EINVAL**.

Related Information
The `pthread_exit` ([`pthread_exit Subroutine` on page 1227]) subroutine, `pthread_create` ([`pthread_create Subroutine` on page 1222]) subroutine, `wait` subroutine, `pthread_cond_wait` or `pthread_cond_timedwait` ([`pthread_cond_wait or pthread_cond_timedwait Subroutine` on page 1215]) subroutines, the `pthread.h` file.

`Joining Threads` in *AIX 5L Version 5.3 General Programming Concepts: Writing and Debugging Programs.*

### pthread_key_create Subroutine

**Purpose**
Creates a thread-specific data key.

**Library**
Threads Library (`libpthreads.a`)

**Syntax**
```
#include <pthread.h>

int pthread_key_create ( key_t *key, void (*destructor)(void *) );
```

**Description**
The `pthread_key_create` subroutine creates a thread-specific data key. The key is shared among all threads within the process, but each thread has specific data associated with the key. The thread-specific data is a void pointer, initially set to **NULL**.

The application is responsible for ensuring that this subroutine is called only once for each requested key. This can be done, for example, by calling the subroutine before creating other threads, or by using the one-time initialization facility.
Typically, thread-specific data are pointers to dynamically allocated storage. When freeing the storage, the value should be set to `NULL`. It is not recommended to cast this pointer into scalar data type (int for example), because the casts may not be portable, and because the value of `NULL` is implementation dependent.

An optional destructor routine can be specified. It will be called for each thread when it is terminated and detached, after the call to the cleanup routines, if the specific value is not `NULL`. Typically, the destructor routine will release the storage thread-specific data. It will receive the thread-specific data as a parameter.

**Note:** The `pthread.h` header file must be the first included file of each source file using the threads library. Otherwise, the `-D_THREAD_SAFE` compilation flag should be used, or the cc_r compiler used. In this case, the flag is automatically set.

**Parameters**

- **key**  
  Points to where the key will be stored.

- **destructor**  
  Points to an optional destructor routine, used to cleanup data on thread termination. If no cleanup is desired, this pointer should be `NULL`.

**Return Values**

If successful, the `pthread_key_create` function stores the newly created key value at *key* and returns zero. Otherwise, an error number is returned to indicate the error.

**Error Codes**

The `pthread_key_create` function will fail if:

- **EAGAIN**  
  The system lacked the necessary resources to create another thread-specific data key, or the system-imposed limit on the total number of keys per process `PTHREAD_KEYS_MAX` has been exceeded.

- **ENOMEM**  
  Insufficient memory exists to create the key.

The `pthread_key_create` function will not return an error code of `EINVAL`.

**Related Information**

The `pthread_exit` ("pthread_exit Subroutine" on page 1227) subroutine, `pthread_key_delete` ("pthread_key_delete Subroutine") subroutine, `pthread_getspecific` ("pthread_getspecific or pthread_setspecific Subroutine" on page 1235) subroutine, `pthread_once` ("pthread_once Subroutine" on page 1262) subroutine, `pthread.h` file.

**pthread_key_delete Subroutine**

**Purpose**

Deletes a thread-specific data key.

**Library**

Threads Library (libpthread.a)
Syntax

```c
#include <pthread.h>

int pthread_key_delete (key);
pthread_key_t key;
```

Description

The `pthread_key_delete` subroutine deletes the thread-specific data key `key`, previously created with the `pthread_key_create` subroutine. The application must ensure that no thread-specific data is associated with the key. No destructor routine is called.

Note: The `pthread.h` header file must be the first included file of each source file using the threads library. Otherwise, the `-D_THREAD_SAFE` compilation flag should be used, or the cc_r compiler used. In this case, the flag is automatically set.

Parameters

`key` Specifies the key to delete.

Return Values

If successful, the `pthread_key_delete` function returns zero. Otherwise, an error number is returned to indicate the error.

Error Codes

The `pthread_key_delete` function will fail if:

- `EINVAL` The key value is invalid.

The `pthread_key_delete` function will not return an error code of `EINTR`.

Related Information

The `pthread_key_create` subroutine, `pthread.h` file.

Thread-Specific Data in AIX 5L Version 5.3 General Programming Concepts: Writing and Debugging Programs.

pthread_kill Subroutine

Purpose

Sends a signal to the specified thread.

Library

Threads Library (`libpthreads.a`)

Syntax

```c
#include <signal.h>

int pthread_kill (thread, signal);
pthread_t thread;
int signal;
```
Description
The `pthread_kill` subroutine sends the signal `signal` to the thread `thread`. It acts with threads like the `kill` subroutine with single-threaded processes.

If the receiving thread has blocked delivery of the signal, the signal remains pending on the thread until the thread unblocks delivery of the signal or the action associated with the signal is set to ignore the signal.

**Note:** The `pthread.h` header file must be the first included file of each source file using the threads library. Otherwise, the `-D_THREAD_SAFE` compilation flag should be used, or the cc_r compiler used. In this case, the flag is automatically set.

Parameters
- `thread` Specifies the target thread for the signal.
- `signal` Specifies the signal to be delivered. If the signal value is 0, error checking is performed, but no signal is delivered.

Return Values
Upon successful completion, the function returns a value of zero. Otherwise the function returns an error number. If the `pthread_kill` function fails, no signal is sent.

Error Codes
The `pthread_kill` function will fail if:
- `ESRCH` No thread could be found corresponding to that specified by the given thread ID.
- `EINVAL` The value of the `signal` parameter is an invalid or unsupported signal number.

The `pthread_kill` function will not return an error code of `EINTR`.

Related Information
The `kill` subroutine, `pthread_cancel` subroutine, `pthread_create` subroutine, `sigaction` subroutine, `pthread_self` subroutine, `raise` subroutine, `pthread.h` file.

Signal Management in AIX 5L Version 5.3 General Programming Concepts: Writing and Debugging Programs.

**pthread_lock_global_np Subroutine**

**Purpose**
Locks the global mutex.

**Library**
Threads Library (libpthreads.a)

**Syntax**
```c
#include <pthread.h>
void pthread_lock_global_np ()
```
**Description**

The `pthread_lock_global_np` subroutine locks the global mutex. If the global mutex is currently held by another thread, the calling thread waits until the global mutex is unlocked. The subroutine returns with the global mutex locked by the calling thread.

Use the global mutex when calling a library package that is not designed to run in a multithreaded environment. (Unless the documentation for a library function specifically states that it is compatible with multithreading, assume that it is not compatible; in other words, assume it is nonreentrant.)

The global mutex is one lock. Any code that calls any function that is not known to be reentrant uses the same lock. This prevents dependencies among threads calling library functions and those functions calling other functions, and so on.

The global mutex is a recursive mutex. A thread that has locked the global mutex can relock it without deadlocking. The thread must then call the `pthread_unlock_global_np` subroutine as many times as it called this routine to allow another thread to lock the global mutex.

**Notes:**

1. The `pthread.h` header file must be the first included file of each source file using the threads library. Otherwise, the `-D_THREAD_SAFE` compilation flag should be used, or the cc_r compiler used. In this case, the flag is automatically set.
2. The `pthread_lock_global_np` subroutine is not portable.

This subroutine is not POSIX compliant and is provided only for compatibility with DCE threads. It should not be used when writing new applications.

**Related Information**

The `pthread_mutex_lock` subroutine, `pthread_mutex_trylock`, or `pthread_mutex_unlock` subroutine, `pthread_unlock_global_np` subroutine.

Using Mutexes in AIX 5L Version 5.3 General Programming Concepts: Writing and Debugging Programs.

**pthread_mutex_init or pthread_mutex_destroy Subroutine**

**Purpose**

Initializes or destroys a mutex.

**Library**

Threads Library (*libpthreads.a*)

**Syntax**

```c
#include <pthread.h>

int pthread_mutex_init (mutex, attr)
    pthread_mutex_t *mutex;
    const pthread_mutexattr_t *attr;

int pthread_mutex_destroy (mutex)
    pthread_mutex_t *mutex;
```
Description
The **pthread_mutex_init** function initializes the mutex referenced by *mutex* with attributes specified by *attr*. If *attr* is NULL, the default mutex attributes are used; the effect is the same as passing the address of a default mutex attributes object. Upon successful initialization, the state of the mutex becomes initialized and unlocked.

Attempting to initialize an already initialized mutex results in undefined behavior.

The **pthread_mutex_destroy** function destroys the mutex object referenced by *mutex*; the mutex object becomes, in effect, uninitialized. An implementation may cause **pthread_mutex_destroy** to set the object referenced by *mutex* to an invalid value. A destroyed mutex object can be re-initialized using **pthread_mutex_init**; the results of otherwise referencing the object after it has been destroyed are undefined.

It is safe to destroy an initialized mutex that is unlocked. Attempting to destroy a locked mutex results in undefined behavior.

In cases where default mutex attributes are appropriate, the macro **PTHREAD_MUTEX_INITIALIZER** can be used to initialize mutexes that are statically allocated. The effect is equivalent to dynamic initialization by a call to **pthread_mutex_init** with parameter *attr* specified as NULL, except that no error checks are performed.

Parameters

*mutex* Specifies the mutex to initialize or delete.

*attr* Specifies the mutex attributes object.

Return Values
If successful, the **pthread_mutex_init** and **pthread_mutex_destroy** functions return zero. Otherwise, an error number is returned to indicate the error. The EBUSY and EINVAL error checks act as if they were performed immediately at the beginning of processing for the function and cause an error return prior to modifying the state of the mutex specified by *mutex*.

Error Codes
The **pthread_mutex_init** function will fail if:

- **ENOMEM** Insufficient memory exists to initialize the mutex.
- **EINVAL** The value specified by *attr* is invalid.
- **EPERM** The caller does not have the privilege to perform the operation in a strictly standards conforming environment where environment variable **XPG_SUS_ENV=ON**.

The **pthread_mutex_destroy** function may fail if:

- **EBUSY** The implementation has detected an attempt to destroy the object referenced by *mutex* while it is locked or referenced (for example, while being used in a **pthread_cond_wait** or **pthread_cond_timedwait** by another thread.
- **EINVAL** The value specified by *mutex* is invalid.

These functions will not return an error code of EINTR.
pthread_mutex_getprioceiling or pthread_mutex_setprioceiling Subroutine

Purpose

 Gets and sets the priority ceiling of a mutex.

Syntax

```c
#include <pthread.h>

int pthread_mutex_getprioceiling(const pthread_mutex_t *restrict mutex,
                                 int *restrict prioceiling);

int pthread_mutex_setprioceiling(pthread_mutex_t *restrict mutex,
                                 int prioceiling,
                                 int *restrict old_ceiling);
```

Description

The `pthread_mutex_getprioceiling` subroutine returns the current priority ceiling of the mutex.

The `pthread_mutex_setprioceiling` subroutine either locks the mutex if it is unlocked, or blocks until it can successfully lock the mutex, then it changes the mutex's priority ceiling and releases the mutex. When the change is successful, the previous value of the priority ceiling shall be returned in `old_ceiling`. The process of locking the mutex need not adhere to the priority protect protocol.

If the `pthread_mutex_setprioceiling` subroutine fails, the mutex priority ceiling is not changed.

Return Values

If successful, the `pthread_mutex_getprioceiling` and `pthread_mutex_setprioceiling` subroutines return zero; otherwise, an error number is returned to indicate the error.

Error Codes

The `pthread_mutex_getprioceiling` and `pthread_mutex_setprioceiling` subroutines can fail if:

- `EINVAL`: The priority requested by the `prioceiling` parameter is out of range.
- `EINVAL`: The value specified by the `mutex` parameter does not refer to a currently existing mutex.
- `ENOSYS`: This function is not supported (draft 7).
- `ENOTSUP`: This function is not supported together with checkpoint/restart.
- `EPERM`: The caller does not have the privilege to perform the operation in a strictly standards conforming environment where environment variable `XPG_SUS_ENV=ON`.

Related Information

The "pthread_mutex_init or pthread_mutex_destroy Subroutine" on page 1246, "pthread_mutex_lock, pthread_mutex_trylock, or pthread_mutex_unlock Subroutine" on page 1249, "pthread_mutexattr_getpshared or pthread_mutexattr_setpshared Subroutine" on page 1258.
The pthread.h file.

**PTHREAD_MUTEX_INITIALIZER Macro**

**Purpose**
Initializes a static mutex with default attributes.

**Library**
Threads Library (**libpthreads.a**)

**Syntax**
```
#include <pthread.h>
static pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;
```

**Description**
The **PTHREAD_MUTEX_INITIALIZER** macro initializes the static mutex `mutex`, setting its attributes to default values. This macro should only be used for static mutexes, as no error checking is performed.

**Note:** The pthread.h header file must be the first included file of each source file using the threads library. Otherwise, the -D_THREAD_SAFE compilation flag should be used, or the cc_r compiler used. In this case, the flag is automatically set.

**Related Information**
The pthread_mutex_init subroutine.

Using Mutexes in AIX 5L Version 5.3 General Programming Concepts: Writing and Debugging Programs.

**pthread_mutex_lock, pthread_mutex_trylock, or pthread_mutex_unlock Subroutine**

**Purpose**
Locks and unlocks a mutex.

**Library**
Threads Library (**libpthreads.a**)

**Syntax**
```
#include <pthread.h>

int pthread_mutex_lock ( pthread_mutex_t *mutex);
int pthread_mutex_trylock ( pthread_mutex_t *mutex);
int pthread_mutex_unlock ( pthread_mutex_t *mutex);
```
Description
The mutex object referenced by the mutex parameter is locked by calling pthread_mutex_lock. If the mutex is already locked, the calling thread blocks until the mutex becomes available. This operation returns with the mutex object referenced by the mutex parameter in the locked state with the calling thread as its owner.

If the mutex type is PTHREAD_MUTEX_NORMAL, deadlock detection is not provided. Attempting to relock the mutex causes deadlock. If a thread attempts to unlock a mutex that it has not locked or a mutex which is unlocked, undefined behavior results.

If the mutex type is PTHREAD_MUTEX_ERRORCHECK, then error checking is provided. If a thread attempts to relock a mutex that it has already locked, an error will be returned. If a thread attempts to unlock a mutex that it has not locked or a mutex which is unlocked, an error will be returned.

If the mutex type is PTHREAD_MUTEX_RECURSIVE, then the mutex maintains the concept of a lock count. When a thread successfully acquires a mutex for the first time, the lock count is set to one. Each time the thread relocks this mutex, the lock count is incremented by one. Each time the thread unlocks the mutex, the lock count is decremented by one. When the lock count reaches zero, the mutex becomes available for other threads to acquire. If a thread attempts to unlock a mutex that has not locked or a mutex which is unlocked, an error will be returned.

If the mutex type is PTHREAD_MUTEX_DEFAULT, attempting to recursively lock the mutex results in undefined behavior. Attempting to unlock the mutex if it was not locked by the calling thread results in undefined behavior. Attempting to unlock the mutex if it is not locked results in undefined behavior.

The function pthread_mutex_trylock is identical to pthread_mutex_lock except that if the mutex object referenced by the mutex parameter is currently locked (by any thread, including the current thread), the call returns immediately.

The pthread_mutex_unlock function releases the mutex object referenced by mutex. The manner in which a mutex is released is dependent upon the mutex's type attribute. If there are threads blocked on the mutex object referenced by the mutex parameter when pthread_mutex_unlock is called, resulting in the mutex becoming available, the scheduling policy is used to determine which thread will acquire the mutex. (In the case of PTHREAD_MUTEX_RECURSIVE mutexes, the mutex becomes available when the count reaches zero and the calling thread no longer has any locks on this mutex).

If a signal is delivered to a thread waiting for a mutex, upon return from the signal handler the thread resumes waiting for the mutex as if it was not interrupted.

Parameter
mutex Specifies the mutex to lock.

Return Values
If successful, the pthread_mutex_lock and pthread_mutex_unlock functions return zero. Otherwise, an error number is returned to indicate the error.

The function pthread_mutex_trylock returns zero if a lock on the mutex object referenced by the mutex parameter is acquired. Otherwise, an error number is returned to indicate the error.
Error Codes
The **pthread_mutex_trylock** function will fail if:

**EBUSY**  The mutex could not be acquired because it was already locked.

The **pthread_mutex_lock**, **pthread_mutex_trylock** and **pthread_mutex_unlock** functions will fail if:

**EINVAL**  The value specified by the **mutex** parameter does not refer to an initialized mutex object.

The **pthread_mutex_lock** function will fail if:

**EDEADLK**  The current thread already owns the mutex and the mutex type is PTHREAD_MUTEX_ERRORCHECK.

The **pthread_mutex_unlock** function will fail if:

**EPERM**  The current thread does not own the mutex and the mutex type is not PTHREAD_MUTEX_NORMAL.

These functions will not return an error code of EINTR.

Related Information
The **pthread_mutex_init** or **pthread_mutex_destroy** subroutine, pthread.h file.

Using Mutexes in AIX 5L Version 5.3 General Programming Concepts: Writing and Debugging Programs.

**pthread_mutex_timedlock** Subroutine

**Purpose**
Locks a mutex (ADVANCED REALTIME).

**Syntax**

```c
#include <pthread.h>
#include <time.h>

int pthread_mutex_timedlock(pthread_mutex_t *restrict mutex,
 const struct timespec *restrict abs_timeout);
```

**Description**
The **pthread_mutex_timedlock()** function locks the mutex object referenced by **mutex**. If the mutex is already locked, the calling thread blocks until the mutex becomes available, as in the **pthread_mutex_lock()** function. If the mutex cannot be locked without waiting for another thread to unlock the mutex, this wait terminates when the specified timeout expires.

The timeout expires when the absolute time specified by **abs_timeout** passes—as measured by the clock on which timeouts are based (that is, when the value of that clock equals or exceeds **abs_timeout**)—or when the absolute time specified by **abs_timeout** has already been passed at the time of the call.

If the **Timers** option is supported, the timeout is based on the CLOCK_REALTIME clock; if the **Timers** option is not supported, the timeout is based on the system clock as returned by the **time()** function.
The resolution of the timeout matches the resolution of the clock on which it is based. The `timespec` data type is defined in the `<time.h>` header.

The function never fails with a timeout if the mutex can be locked immediately. The validity of the `abs_timeout` parameter does not need to be checked if the mutex can be locked immediately.

As a consequence of the priority inheritance rules (for mutexes initialized with the PRIO_INHERIT protocol), if a timed mutex wait is terminated because its timeout expires, the priority of the owner of the mutex adjusts as necessary to reflect the fact that this thread is no longer among the threads waiting for the mutex.

**Application Usage**

The `pthread_mutex_timedlock()` function is part of the Threads and Timeouts options and do not need to be provided on all implementations.

**Return Values**

If successful, the `pthread_mutex_timedlock()` function returns 0; otherwise, an error number is returned to indicate the error.

**Error Codes**

The `pthread_mutex_timedlock()` function fails if:

- `[EDEADLK]` The current thread already owns the mutex.
- `[EINVAL]` The mutex was created with the protocol attribute having the value `PTHREAD_PRIO_PROTECT`, and the calling thread's priority is higher than the mutex's current priority ceiling.
- `[EINVAL]` The process or thread would have blocked, and the `abs_timeout` parameter specified a nanoseconds field value less than 0 or greater than or equal to 1000 million.
- `[EINVAL]` `abs_timeout` is a NULL pointer.
- `[EINVAL]` The value specified by `mutex` does not refer to an initialized mutex object.
- `[ETIMEDOUT]` The mutex could not be locked before the specified timeout expired.

This function does not return an error code of [EINVAL].

**Related Information**

- "mq_receive, mq_timedreceive Subroutine" on page 861.
- "posix_trace_getnext_event, posix_trace_timedgetnext_event, posix_trace_trygetnext_event Subroutine" on page 1143.
- "pthread_mutexattr_destroy or pthread_mutexattr_init Subroutine, pthread_mutex_lock, pthread_mutex_trylock, or pthread_mutex_unlock Subroutine" on page 1249.
- "pthread_rwlock_timedrdlock Subroutine" on page 1266.


The `pthread.h` and `time.h` files in AIX 5L Version 5.3 Files Reference.

**pthread_mutexattr_destroy or pthread_mutexattr_init Subroutine**

**Purpose**

Initializes and destroys mutex attributes.
Library
Threads Library (libpthreads.a)

Syntax
#include <pthread.h>

int pthread_mutexattr_init (attr)
    pthread_mutexattr_t *attr;

int pthread_mutexattr_destroy (attr)
    pthread_mutexattr_t *attr;

Description
The function pthread_mutexattr_init initializes a mutex attributes object attr with the default value for all of the attributes defined by the implementation.

The effect of initializing an already initialized mutex attributes object is undefined.

After a mutex attributes object has been used to initialize one or more mutexes, any function affecting the attributes object (including destruction) does not affect any previously initialized mutexes.

The pthread_mutexattr_destroy function destroys a mutex attributes object; the object becomes, in effect, uninitialized. An implementation may cause pthread_mutexattr_destroy to set the object referenced by attr to an invalid value. A destroyed mutex attributes object can be re-initialized using pthread_mutexattr_init; the results of otherwise referencing the object after it has been destroyed are undefined.

Parameters
attr Specifies the mutex attributes object to initialize or delete.

Return Values
Upon successful completion, pthread_mutexattr_init and pthread_mutexattr_destroy return zero. Otherwise, an error number is returned to indicate the error.

Error Codes
The pthread_mutexattr_init function will fail if:

ENOMEM Insufficient memory exists to initialize the mutex attributes object.

The pthread_mutexattr_destroy function will fail if:

EINVAL The value specified by attr is invalid.

These functions will not return EINTR.

Related Information
The pthread_create subroutine, pthread_mutex_init or pthread_mutex_destroy subroutine, pthread_cond_destroy or pthread_cond_init subroutine, pthread.h file.
**pthread_mutexattr_getkind_np Subroutine**

**Purpose**
Returns the value of the kind attribute of a mutex attributes object.

**Library**
Threads Library (libpthread.a)

**Syntax**
```
#include <pthread.h>

int pthread_mutexattr_getkind_np (attr, kind)
pthread_mutexattr_t *attr;
int *kind;
```

**Description**
The `pthread_mutexattr_getkind_np` subroutine returns the value of the kind attribute of the mutex attributes object `attr`. This attribute specifies the kind of the mutex created with this attributes object. It may have one of the following values:

- **MUTEX_FAST_NP**
  Denotes a fast mutex. A fast mutex can be locked only once. If the same thread unlocks twice the same fast mutex, the thread will deadlock. Any thread can unlock a fast mutex. A fast mutex is not compatible with the priority inheritance protocol.

- **MUTEX_RECURSIVE_NP**
  Denotes a recursive mutex. A recursive mutex can be locked more than once by the same thread without causing that thread to deadlock. The thread must then unlock the mutex as many times as it locked it. Only the thread that locked a recursive mutex can unlock it. A recursive mutex must not be used with condition variables.

- **MUTEX_NONRECURSIVE_NP**
  Denotes the default non-recursive POSIX compliant mutex.

**Notes:**
1. The `pthread.h` header file must be the first included file of each source file using the threads library. Otherwise, the `-D_THREAD_SAFE` compilation flag should be used, or the cc_r compiler used. In this case, the flag is automatically set.
2. The `pthread_mutexattr_getkind_np` subroutine is not portable.

This subroutine is not POSIX compliant and is provided only for compatibility with DCE threads. It should not be used when writing new applications.

**Parameters**
- `attr` Specifies the mutex attributes object.
- `kind` Points to where the kind attribute value will be stored.
Return Values
Upon successful completion, the value of the kind attribute is returned via the kind parameter, and 0 is returned. Otherwise, an error code is returned.

Error Codes
The pthread_mutexattr_getkind_np subroutine is unsuccessful if the following is true:
EINVAL The attr parameter is not valid.

Related Information
The pthread_mutexattr_setkind_np subroutine.

pthreads in AIX 5L Version 5.3 General Programming Concepts: Writing and Debugging Programs.

pthread_mutexattr_getprioceiling or pthread_mutexattr_setprioceiling Subroutine

Purpose
Gets and sets the priority ceiling attribute of the mutex attributes object.

Syntax
#include <pthread.h>

int pthread_mutexattr_getprioceiling(const pthread_mutexattr_t * restrict attr, int *restrict prioceiling);
int pthread_mutexattr_setprioceiling(pthread_mutexattr_t *attr, int prioceiling);

Description
The pthread_mutexattr_getprioceiling and pthread_mutexattr_setprioceiling subroutines, respectively, get and set the priority ceiling attribute of a mutex attributes object pointed to by the attr parameter, which was previously created by the pthread_mutexattr_init subroutine.

The prioceiling attribute contains the priority ceiling of initialized mutexes. The values of the prioceiling parameter are within the maximum range of priorities defined by SCHED_FIFO.

The prioceiling parameter defines the priority ceiling of initialized mutexes, which is the minimum priority level at which the critical section guarded by the mutex is executed. In order to avoid priority inversion, the priority ceiling of the mutex is set to a priority higher than or equal to the highest priority of all the threads that may lock that mutex. The values of the prioceiling parameter are within the maximum range of priorities defined under the SCHED_FIFO scheduling policy.

Return Values
Upon successful completion, the pthread_mutexattr_getprioceiling and pthread_mutexattr_setprioceiling subroutines return zero; otherwise, an error number shall be returned to indicate the error.
Error Codes
The `pthread_mutexattr_getprioceiling` and `pthread_mutexattr_setprioceiling` subroutines can fail if:

- **EINVAL** The value specified by the `attr` or `prioceiling` parameter is invalid.
- **ENOSYS** This function is not supported (draft 7).
- **ENOTSUP** This function is not supported together with checkpoint/restart.
- **EPERM** The caller does not have the privilege to perform the operation in a strictly standards conforming environment where environment variable `XPG_SUS_ENV=ON`.

Related Information
The "pthread_mutex_init or pthread_mutex_destroy Subroutine" on page 1246, "pthread_mutex_lock, pthread_mutex_trylock, or pthread_mutex_unlock Subroutine" on page 1249, "pthread_mutex_timedlock Subroutine" on page 1251.

The `pthread.h` file.

**pthread_mutexattr_getprotocol or pthread_mutexattr_setprotocol**

Subroutine

**Purpose**
Gets and sets the protocol attribute of the mutex attributes object.

**Syntax**
```c
#include <pthread.h>

int pthread_mutexattr_getprotocol(const pthread_mutexattr_t * restrict attr, int *restrict protocol);
int pthread_mutexattr_setprotocol(pthread_mutexattr_t *attr, int protocol);
```

**Description**
The `pthread_mutexattr_getprotocol` subroutine and `pthread_mutexattr_setprotocol` subroutine get and set the `protocol` parameter of a mutex attributes object pointed to by the `attr` parameter, which was previously created by the `pthread_mutexattr_init` subroutine.

The protocol attribute defines the protocol to be followed in utilizing mutexes. The value of the `protocol` parameter can be one of the following, which are defined in the `pthread.h` header file:

- **PTHREAD_PRIO_NONE**
- **PTHREAD_PRIO_INHERIT**
- **PTHREAD_PRIO_PROTECT**

When a thread owns a mutex with the `PTHREAD_PRIO_NONE` protocol attribute, its priority and scheduling are not affected by its mutex ownership.

When a thread is blocking higher priority threads because of owning one or more mutexes with the `PTHREAD_PRIO_INHERIT` protocol attribute, it executes at the higher of its priority or the priority of the highest priority thread waiting on any of the mutexes owned by this thread and initialized with this protocol.

When a thread owns one or more mutexes initialized with the `PTHREAD_PRIO_PROTECT` protocol, it executes at the higher of its priority or the highest of the priority ceilings of all the mutexes owned by this thread and initialized with this attribute, regardless of whether other threads are blocked on any of these mutexes. Privilege checking is necessary when the mutex priority ceiling is more favored than current...
thread priority and the thread priority must be changed. The `pthread_mutex_lock` subroutine does not fail because of inappropriate privileges. Locking succeeds in this case, but no boosting is performed.

While a thread is holding a mutex which has been initialized with the `PTHREAD_PRIO_INHERIT` or `PTHREAD_PRIO_PROTECT` protocol attributes, it is not subject to being moved to the tail of the scheduling queue at its priority in the event that its original priority is changed, such as by a call to the `sched_setparam` subroutine. Likewise, when a thread unlocks a mutex that has been initialized with the `PTHREAD_PRIO_INHERIT` or `PTHREAD_PRIO_PROTECT` protocol attributes, it is not subject to being moved to the tail of the scheduling queue at its priority in the event that its original priority is changed.

If a thread simultaneously owns several mutexes initialized with different protocols, it executes at the highest of the priorities that it would have obtained by each of these protocols.

When a thread makes a call to the `pthread_mutex_lock` subroutine, the mutex was initialized with the protocol attribute having the value `PTHREAD_PRIO_INHERIT`, when the calling thread is blocked because the mutex is owned by another thread, that owner thread inherits the priority level of the calling thread as long as it continues to own the mutex. The implementation updates its execution priority to the maximum of its assigned priority and all its inherited priorities. Furthermore, if this owner thread itself becomes blocked on another mutex, the same priority inheritance effect shall be propagated to this other owner thread, in a recursive manner.

Behavior prior to AIX 5.3 is maintained under the non-POSIX protocol `PTHREAD_PRIO_DEFAULT`.

**Return Values**

Upon successful completion, the `pthread_mutexattr_getprotocol` subroutine and the `pthread_mutexattr_setprotocol` subroutine return zero; otherwise, an error number shall be returned to indicate the error.

**Error Codes**

The `pthread_mutexattr_setprotocol` subroutine fails if:

- **ENOTSUP**: The value specified by the `protocol` parameter is an unsupported value.

The `pthread_mutexattr_getprotocol` subroutine and `pthread_mutexattr_setprotocol` subroutine can fail if:

- **EINVAL**: The value specified by the `attr` parameter or the `protocol` parameter is invalid.
- **ENOSYS**: This function is not supported (draft 7).
- **ENOTSUP**: This function is not supported together with checkpoint/restart.
- **EPERM**: The caller does not have the privilege to perform the operation in a strictly standards conforming environment where environment variable `XPG_SUS_ENV=ON`.

**Related Information**

The "pthread_mutex_init or pthread_mutex_destroy Subroutine" on page 1246, "pthread_mutex_lock, pthread_mutex_trylock, or pthread_mutex_unlock Subroutine" on page 1249, "pthread_mutex_timedlock Subroutine" on page 1251.

The `pthread.h` file.
**pthread_mutexattr_getpshared or pthread_mutexattr_setpshared**

**Subroutine**

**Purpose**
Sets and gets process-shared attribute.

**Library**
Threads Library (libpthreads.a)

**Syntax**

```c
#include <pthread.h>

int pthread_mutexattr_getpshared (attr, pshared)
const pthread_mutexattr_t *attr;
int *pshared;

int pthread_mutexattr_setpshared (attr, pshared)
pthread_mutexattr_t *attr;
int pshared;
```

**Description**

The `pthread_mutexattr_getpshared` subroutine obtains the value of the process-shared attribute from the attributes object referenced by `attr`. The `pthread_mutexattr_setpshared` subroutine is used to set the process-shared attribute in an initialized attributes object referenced by `attr`.

The process-shared attribute is set to PTHREAD_PROCESS_SHARED to permit a mutex to be operated upon by any thread that has access to the memory where the mutex is allocated, even if the mutex is allocated in memory that is shared by multiple processes. If the process-shared attribute is PTHREAD_PROCESS_PRIVATE, the mutex will only be operated upon by threads created within the same process as the thread that initialized the mutex; if threads of differing processes attempt to operate on such a mutex, the behavior is undefined. The default value of the attribute is PTHREAD_PROCESS_PRIVATE.

**Parameters**

- `attr` Specifies the mutex attributes object.
- `pshared` Points to where the pshared attribute value will be stored.

**Return Values**

Upon successful completion, the `pthread_mutexattr_setpshared` subroutine returns zero. Otherwise, an error number is returned to indicate the error.

Upon successful completion, the `pthread_mutexattr_getpshared` subroutine returns zero and stores the value of the process-shared attribute of `attr` into the object referenced by the `pshared` parameter. Otherwise, an error number is returned to indicate the error.

**Error Codes**

The `pthread_mutexattr_getpshared` and `pthread_mutexattr_setpshared` subroutines will fail if:

- `EINVAL` The value specified by `attr` is invalid.
The `pthread_mutexattr_setpshared` function will fail if:

**EINVAL** The new value specified for the attribute is outside the range of legal values for that attribute.

These subroutines will not return an error code of EINTR.

**Related Information**
The **pthread_mutexattr_init** ("pthread_mutexattr_destroy or pthread_mutexattr_init Subroutine" on page 1252) subroutine.

**Advanced Attributes** in AIX 5L Version 5.3 General Programming Concepts: Writing and Debugging Programs.

**Threads Library Options** in AIX 5L Version 5.3 General Programming Concepts: Writing and Debugging Programs.

---

**pthread_mutexattr_gettype or pthread_mutexattr_settype Subroutine**

**Purpose**
 Gets or sets a mutex type.

**Library**
 Threads Library (libthreads.a)

**Syntax**

```c
#include <pthread.h>

int pthread_mutexattr_gettype (attr, type)
const pthread_mutexattr_t *attr;
int *type;

int pthread_mutexattr_settype (attr, type)
pthread_mutexattr_t *attr;
int type;
```

**Description**
The **pthread_mutexattr_gettype** and **pthread_mutexattr_settype** subroutines respectively get and set the mutex type attribute. This attribute is set in the *type* parameter to these subroutines. The default value of the type attribute is PTHREAD_MUTEX_DEFAULT. The type of mutex is contained in the type attribute of the mutex attributes. Valid mutex types include:

**PTHREAD_MUTEX_NORMAL**

This type of mutex does not detect deadlock. A thread attempting to relock this mutex without first unlocking it will deadlock. Attempting to unlock a mutex locked by a different thread results in undefined behavior. Attempting to unlock an unlocked mutex results in undefined behavior.

**PTHREAD_MUTEX_ERRORCHECK**

This type of mutex provides error checking. A thread attempting to relock this mutex without first unlocking it will return with an error. A thread attempting to unlock a mutex which another thread has locked will return with an error. A thread attempting to unlock an unlocked mutex will return with an error.
PTHREAD_MUTEX_RECURSIVE

A thread attempting to relock this mutex without first unlocking it will succeed in locking the mutex. The relocking deadlock which can occur with mutexes of type PTHREAD_MUTEX_NORMAL cannot occur with this type of mutex. Multiple locks of this mutex require the same number of unlocks to release the mutex before another thread can acquire the mutex. A thread attempting to unlock a mutex which another thread has locked will return with an error. A thread attempting to unlock an unlocked mutex will return with an error.

PTHREAD_MUTEX_DEFAULT

Attempting to recursively lock a mutex of this type results in undefined behavior. Attempting to unlock a mutex of this type which was not locked by the calling thread results in undefined behavior. Attempting to unlock a mutex of this type which is not locked results in undefined behavior. An implementation is allowed to map this mutex to one of the other mutex types.

It is advised that an application should not use a PTHREAD_MUTEX_RECURSIVE mutex with condition variables because the implicit unlock performed for a pthread_cond_wait or pthread_cond_timedwait may not actually release the mutex (if it had been locked multiple times). If this happens, no other thread can satisfy the condition of the predicate.

Parameters

attr Specifies the mutex object to get or set.

type Specifies the type to get or set.

Return Values

If successful, the pthread_mutexattr_settype subroutine returns zero. Otherwise, an error number is returned to indicate the error. Upon successful completion, the pthread_mutexattr_gettype subroutine returns zero and stores the value of the type attribute of attr into the object referenced by the type parameter. Otherwise an error is returned to indicate the error.

Error Codes

The pthread_mutexattr_gettype and pthread_mutexattr_settype subroutines will fail if:

EINVAL The value of the type parameter is invalid.
EINVAL The value specified by the attr parameter is invalid.

Related Information

The pthread_cond_wait and pthread_cond_timedwait subroutines.

The pthread.h file.

pthread_mutexattr_setkind_np Subroutine

Purpose

Sets the value of the kind attribute of a mutex attributes object.
Library
Threads Library (libpthreads.a)

Syntax
#include <pthread.h>

int pthread_mutexattr_setkind_np (attr, kind)
pthread_mutexattr_t *attr;
int kind;

Description
The `pthread_mutexattr_setkind_np` subroutine sets the value of the kind attribute of the mutex attributes object `attr`. This attribute specifies the kind of the mutex created with this attributes object.

Notes:
1. The `pthread.h` header file must be the first included file of each source file using the threads library. Otherwise, the `-D_THREAD_SAFE` compilation flag should be used, or the cc_r compiler used. In this case, the flag is automatically set.
2. The `pthread_mutexattr_setkind_np` subroutine is not portable.

This subroutine is provided only for compatibility with the DCE threads. It should not be used when writing new applications.

Parameters
attr Specifies the mutex attributes object.
kind Specifies the kind to set. It must have one of the following values:

- **MUTEX_FAST_NP**
  Denotes a fast mutex. A fast mutex can be locked only once. If the same thread unlocks twice the same fast mutex, the thread will deadlock. Any thread can unlock a fast mutex. A fast mutex is not compatible with the priority inheritance protocol.

- **MUTEX_RECURSIVE_NP**
  Denotes a recursive mutex. A recursive mutex can be locked more than once by the same thread without causing that thread to deadlock. The thread must then unlock the mutex as many times as it locked it. Only the thread that locked a recursive mutex can unlock it. A recursive mutex must not be used with condition variables.

- **MUTEX_NONRECURSIVE_NP**
  Denotes the default non-recursive POSIX compliant mutex.

Return Values
Upon successful completion, 0 is returned. Otherwise, an error code is returned.

Error Codes
The `pthread_mutexattr_setkind_np` subroutine is unsuccessful if the following is true:

- **EINVAL** The `attr` parameter is not valid.
- **ENOTSUP** The value of the `kind` parameter is not supported.

Related Information
The `pthread_mutexattr_getkind_np` subroutine.
**pthread_once Subroutine**

**Purpose**
Executes a routine exactly once in a process.

**Library**
Threads Library (libpthread.a)

**Syntax**
```c
#include <pthread.h>

int pthread_once (once_control, init_routine)

pthread_once_t *once_control;
void (*init_routine)(void);

, pthread_once_t once_control = PTHREAD_ONCE_INIT;
```

**Description**
The `pthread_once` subroutine executes the routine `init_routine` exactly once in a process. The first call to this subroutine by any thread in the process executes the given routine, without parameters. Any subsequent call will have no effect.

The `init_routine` routine is typically an initialization routine. Multiple initializations can be handled by multiple instances of `pthread_once_t` structures. This subroutine is useful when a unique initialization has to be done by one thread among many. It reduces synchronization requirements.

**Note:** The `pthread.h` header file must be the first included file of each source file using the threads library. Otherwise, the `-D_THREAD_SAFE` compilation flag should be used, or the cc_r compiler used. In this case, the flag is automatically set.

**Parameters**
- `once_control` Points to a synchronization control structure. This structure has to be initialized by the static initializer macro `PTHREAD_ONCE_INIT`.
- `init_routine` Points to the routine to be executed.

**Return Values**
Upon successful completion, `pthread_once` returns zero. Otherwise, an error number is returned to indicate the error.

**Error Codes**
No errors are defined. The `pthread_once` function will not return an error code of EINTR.

**Related Information**
The `pthread_create` subroutine, `pthread.h` file, `PTHREAD_ONCE_INIT` macro.
PTHREAD_ONCE_INIT Macro

Purpose
Initializes a once synchronization control structure.

Library
Threads Library (libpthreads.a)

Syntax
#include <pthread.h>
static pthread_once_t once_block = PTHREAD_ONCE_INIT;

Description
The PTHREAD_ONCE_INIT macro initializes the static once synchronization control structure once_block, used for one-time initializations with the pthread_once ("pthread_once Subroutine" on page 1262) subroutine. The once synchronization control structure must be static to ensure the unicity of the initialization.

Note: The pthread.h file header file must be the first included file of each source file using the threads library. Otherwise, the -D_THREAD_SAFE compilation flag should be used, or the cc_r compiler used. In this case, the flag is automatically set.

Related Information
The pthread_once ("pthread_once Subroutine" on page 1262) subroutine.

One Time Initializations in AIX 5L Version 5.3 General Programming Concepts: Writing and Debugging Programs.

pthread_rwlock_init or pthread_rwlock_destroy Subroutine

Purpose
Initializes or destroys a read-write lock object.

Library
Threads Library (libthreads.a)

Syntax
#include <pthread.h>
int pthread_rwlock_init (rwlock, attr)
    pthread_rwlock_t *rwlock;
    const pthread_rwlockattr_t *attr;

int pthread_rwlock_destroy (rwlock)
    pthread_rwlock_t *rwlock;
    pthread_rwlock_t rwlock=PTHREAD_RWLOCK_INITIALIZER;

Description
The pthread_rwlock_init subroutine initializes the read-write lock referenced by rwlock with the attributes referenced by attr. If attr is NULL, the default read-write lock attributes are used; the effect is the same as passing the address of a default read-write lock attributes object. Once initialized, the lock can be used
any number of times without being re-initialized. Upon successful initialization, the state of the read-write lock becomes initialized and unlocked. Results are undefined if `pthread_rwlock_init` is called specifying an already initialized read-write lock. Results are undefined if a read-write lock is used without first being initialized.

If the `pthread_rwlock_init` function fails, `rwlock` is not initialized and the contents of `rwlock` are undefined.

The `pthread_rwlock_destroy` function destroys the read-write lock object referenced by `rwlock` and releases any resources used by the lock. The effect of subsequent use of the lock is undefined until the lock is re-initialized by another call to `pthread_rwlock_init`. An implementation may cause `pthread_rwlock_destroy` to set the object referenced by `rwlock` to an invalid value. Results are undefined if `pthread_rwlock_destroy` is called when any thread holds `rwlock`. Attempting to destroy an uninitialized read-write lock results in undefined behavior. A destroyed read-write lock object can be re-initialized using `pthread_rwlock_init`; the results of otherwise referencing the read-write lock object after it has been destroyed are undefined.

In cases where default read-write lock attributes are appropriate, the macro `PTHREAD_RWLOCK_INITIALIZER` can be used to initialize read-write locks that are statically allocated. The effect is equivalent to dynamic initialization by a call to `pthread_rwlock_init` with the parameter `attr` specified as NULL, except that no error checks are performed.

**Parameters**

- **rwlock** Specifies the read-write lock to be initialized or destroyed.
- **attr** Specifies the attributes of the read-write lock to be initialized.

**Return Values**

If successful, the `pthread_rwlock_init` and `pthread_rwlock_destroy` functions return zero. Otherwise, an error number is returned to indicate the error. The EBUSY and EINVAL error checks, if implemented, will act as if they were performed immediately at the beginning of processing for the function and caused an error return prior to modifying the state of the read-write lock specified by `rwlock`.

**Error Codes**

The `pthread_rwlock_init` subroutine will fail if:

- **ENOMEM** Insufficient memory exists to initialize the read-write lock.
- **EINVAL** The value specified by `attr` is invalid.

The `pthread_rwlock_destroy` subroutine will fail if:

- **EBUSY** The implementation has detected an attempt to destroy the object referenced by `rwlock` while it is locked.
- **EINVAL** The value specified by `attr` is invalid.

**Related Information**

The `pthread.h` file.

The `pthread_rwlock_rdlock` subroutine on page 1265, `pthread_rwlock_wrlock` subroutine on page 1270, `pthread_rwlockattr_init` subroutine on page 1272, and `pthread_rwlock_unlock` subroutine on page 1269.
pthread_rwlock_rdlock or pthread_rwlock_tryrdlock Subroutines

Purpose
Locks a read-write lock object for reading.

Library
Threads Library (libpthreads.a)

Syntax
#include <pthread.h>

int pthread_rwlock_rdlock (rwlock)
    pthread_rwlock_t *rwlock;

int pthread_rwlock_tryrdlock (rwlock)
    pthread_rwlock_t *rwlock;

Description
The pthread_rwlock_rdlock function applies a read lock to the read-write lock referenced by rwlock. The calling thread acquires the read lock if a writer does not hold the lock and there are no writers blocked on the lock. It is unspecified whether the calling thread acquires the lock when a writer does not hold the lock and there are writers waiting for the lock. If a writer holds the lock, the calling thread will not acquire the read lock. If the read lock is not acquired, the calling thread blocks (that is, it does not return from the pthread_rwlock_rdlock call) until it can acquire the lock. Results are undefined if the calling thread holds a write lock on rwlock at the time the call is made.

Implementations are allowed to favor writers over readers to avoid writer starvation.

A thread may hold multiple concurrent read locks on rwlock (that is, successfully call the pthread_rwlock_rdlock function n times). If so, the thread must perform matching unlocks (that is, it must call the pthread_rwlock_unlock function n times).

The function pthread_rwlock_tryrdlock applies a read lock as in the pthread_rwlock_rdlock function with the exception that the function fails if any thread holds a write lock on rwlock or there are writers blocked on rwlock.

Results are undefined if any of these functions are called with an uninitialized read-write lock.

If a signal is delivered to a thread waiting for a read-write lock for reading, upon return from the signal handler the thread resumes waiting for the read-write lock for reading as if it was not interrupted.

Parameters
rwlock Specifies the read-write lock to be locked for reading.

Return Values
If successful, the pthread_rwlock_rdlock function returns zero. Otherwise, an error number is returned to indicate the error.

The function pthread_rwlock_tryrdlock returns zero if the lock for reading on the read-write lock object referenced by rwlock is acquired. Otherwise an error number is returned to indicate the error.


Error Codes

The `pthread_rwlock_tryrdlock` function will fail if:

- **EBUSY** The read-write lock could not be acquired for reading because a writer holds the lock or was blocked on it.

The `pthread_rwlock_rdlock` and `pthread_rwlock_tryrdlock` functions will fail if:

- **EINVAL** The value specified by `rwlock` does not refer to an initialized read-write lock object.
- **EDEADLK** The current thread already owns the read-write lock for writing.
- **EAGAIN** The read lock could not be acquired because the maximum number of read locks for `rwlock` has been exceeded.

Implementation Specifics

Realtime applications may encounter priority inversion when using read-write locks. The problem occurs when a high priority thread 'locks' a read-write lock that is about to be 'unlocked' by a low priority thread, but the low priority thread is preempted by a medium priority thread. This scenario leads to priority inversion; a high priority thread is blocked by lower priority threads for an unlimited period of time. During system design, realtime programmers must take into account the possibility of this kind of priority inversion. They can deal with it in a number of ways, such as by having critical sections that are guarded by read-write locks execute at a high priority, so that a thread cannot be preempted while executing in its critical section.

Related Information

The `pthread.h` file.

The `pthread_rwlock_init` function ("pthread_rwlock_init or pthread_rwlock_destroy Subroutine" on page 1263), `pthread_rwlock_rwlock` function ("pthread_rwlock_rwlock or pthread_rwlock_tryrwlock Subroutines" on page 1270), `pthread_rwlockattr_init` function ("pthread_rwlockattr_init or pthread_rwlockattr_destroy Subroutines" on page 1272), and `pthread_rwlock_unlock` function ("pthread_rwlock_unlock Subroutine" on page 1269) subroutines.

pthread_rwlock_timedrdlock Subroutine

Purpose

Locks a read-write lock for reading.

Syntax

```c
#include <pthread.h>
#include <time.h>

int pthread_rwlock_timedrdlock(pthread_rwlock_t *restrict rwlock,
                                 const struct timespec *restrict abs_timeout);
```

Description

The `pthread_rwlock_timedrdlock()` function applies a read lock to the read-write lock referenced by `rwlock` as in the `pthread_rwlock_rwlock()` function. However, if the lock cannot be acquired without waiting for other threads to unlock the lock, this wait terminates when the specified timeout expires. The timeout expires when the absolute time specified by `abs_timeout` passes—as measured by the clock on which timeouts are based (that is, when the value of that clock equals or exceeds `abs_timeout`)—or when the absolute time specified by `abs_timeout` has already been passed at the time of the call.
If the **Timers** option is supported, the timeout is based on the CLOCK_REALTIME clock; if the **Timers** option is not supported, the timeout is based on the system clock as returned by the **time()** function.

The resolution of the timeout matches the resolution of the clock on which it is based. The **timestpec** data type is defined in the `<time.h>` header.

The function never fails with a timeout if the lock can be acquired immediately. The validity of the **abs_timeout** parameter does not need to be checked if the lock can be immediately acquired.

If a signal that causes a signal handler to be executed is delivered to a thread that is blocked on a read-write lock through a call to **pthread_rwlock_timedrdlock()**, the thread resumes waiting for the lock (as if it were not interrupted) after the signal handler returns.

The calling thread can deadlock if it holds a write lock on **rwlock** at the time the call is made. The results are undefined if this function is called with an uninitialized read-write lock.

**Application Usage**

The **pthread_rwlock_timedrdlock()** function is part of the **Threads** and **Timeouts** options and do not need to be provided on all implementations.

**Return Values**

The **pthread_rwlock_timedrdlock()** function returns 0 if the lock for reading on the read-write lock object referenced by **rwlock** is acquired. Otherwise, an error number is returned to indicate the error.

**Error Codes**

The **pthread_rwlock_timedrdlock()** function fails if:

- **[ETIMEDOUT]** The lock could not be acquired before the specified timeout expired.

The **pthread_rwlock_timedrdlock()** function might fail if:

- **[EAGAIN]** The read lock could not be acquired because the maximum number of read locks for lock would be exceeded.
- **[EDEADLK]** The calling thread already holds a write lock on **rwlock**.
- **[EINVAL]** The value specified by **rwlock** does not refer to an initialized read-write lock object, or the **abs_timeout** nanosecond value is less than 0 or greater than or equal to 1000 million.

This function does not return an error code of **[EINTR]**.

**Related Information**

- "mq_receive, mq_timedreceive Subroutine" on page 861
- "posix_trace_getnext_event, posix_trace_timedgetnext_event, posix_trace_trygetnext_event Subroutine" on page 1143
- "pthread_mutex_timedlock Subroutine" on page 1251
- "pthread_rwlock_init or pthread_rwlock_destroy Subroutine" on page 1263
- "pthread_rwlock_rdlock or pthread_rwlock_tryrdlock Subroutines" on page 1265
- "pthread_rwlock_timedwrlock Subroutine" on page 1268
- "pthread_rwlock_wrlock or pthread_rwlock_trywrlock Subroutines" on page 1270
- "pthread_rwlock_unlock Subroutine" on page 1269

The **sem_timedwait** subroutine in **AIX 5L Version 5.3 Technical Reference: Base Operating System and Extensions Volume 2**.

The **pthread.h** and **time.h** files in **AIX 5L Version 5.3 Files Reference**.
pthread_rwlock_timedwrlock Subroutine

Purpose
Locks a read-write lock for writing.

Syntax
#include <pthread.h>
#include <time.h>

int pthread_rwlock_timedwrlock(pthread_rwlock_t *restrict rwlock,
                const struct timespec *restrict abs_timeout);

Description
The pthread_rwlock_timedwrlock() function applies a write lock to the read-write lock referenced by rwlock as in the pthread_rwlock_wrlock() function. However, if the lock cannot be acquired without waiting for other threads to unlock the lock, this wait terminates when the specified timeout expires. The timeout expires when the absolute time specified by abs_timeout passes—as measured by the clock on which timeouts are based (that is, when the value of that clock equals or exceeds abs_timeout)—or when the absolute time specified by abs_timeout has already been passed at the time of the call.

If the Timers option is supported, the timeout is based on the CLOCK_REALTIME clock; if the Timers option is not supported, the timeout is based on the system clock as returned by the time() function.

The resolution of the timeout matches the resolution of the clock on which it is based. The timespec data type is defined in the <time.h> header.

The function never fails with a timeout if the lock can be acquired immediately. The validity of the abs_timeout parameter does not need to be checked if the lock can be immediately acquired.

If a signal that causes a signal handler to be executed is delivered to a thread that is blocked on a read-write lock through a call to pthread_rwlock_timedwrlock(), the thread resumes waiting for the lock (as if it were not interrupted) after the signal handler returns.

The calling thread can deadlock if it holds the read-write lock at the time the call is made. The results are undefined if this function is called with an uninitialized read-write lock.

Application Usage
The pthread_rwlock_timedwrlock() function is part of the Threads and Timeouts options and do not need to be provided on all implementations.

Return Values
The pthread_rwlock_timedwrlock() function returns 0 if the lock for writing on the read-write lock object referenced by rwlock is acquired. Otherwise, an error number is returned to indicate the error.

Error Codes
The pthread_rwlock_timedwrlock() function fails if:

ETIMEDOUT The lock could not be acquired before the specified timeout expired.

The pthread_rwlock_timedwrlock() function might fail if:

EDEADLK The calling thread already holds the rwlock.
EINVAL

The value specified by `rwlock` does not refer to an initialized read-write lock object, or
the `abs_timeout` nanosecond value is less than 0 or greater than or equal to 1000
million.

This function does not return an error code of EINTR.

### Related Information

- "mq_receive, mq_timedreceive Subroutine" on page 861
- "posix_trace_getnext_event, posix_trace_timedgetnext_event Subroutine" on page 1143
- "pthread_mutex_timedlock Subroutine" on page 1251
- "pthread_rwlock_init or pthread_rwlock_destroy Subroutine" on page 1263
- "pthread_rwlock_rdlock or pthread_rwlock_tryrdlock Subroutines" on page 1265
- "pthread_rwlock_wrlock or pthread_rwlock_trywrlock Subroutines" on page 1270
- "pthread_rwlock_unlock Subroutine."

The `sem_timedwait` subroutine in AIX 5L Version 5.3 Technical Reference: Base Operating System and
Extensions Volume 2.

The `pthread.h` and `time.h` files in AIX 5L Version 5.3 Files Reference.

### pthread_rwlock_unlock Subroutine

#### Purpose

Unlocks a read-write lock object.

#### Library

Threads Library (libthreads.a)

#### Syntax

```c
#include <pthread.h>

int pthread_rwlock_unlock (pthread_rwlock_t *rwlock);
```

#### Description

The `pthread_rwlock_unlock` subroutine is called to release a lock held on the read-write lock object
referenced by `rwlock`. Results are undefined if the read-write lock `rwlock` is not held by the calling thread.

If this subroutine is called to release a read lock from the read-write lock object and there are other read
locks currently held on this read-write lock object, the read-write lock object remains in the read locked
state. If this subroutine releases the calling thread’s last read lock on this read-write lock object, then the
calling thread is no longer one of the owners of the object. If this subroutine releases the last read lock for
this read-write lock object, the read-write lock object will be put in the unlocked state with no owners.

If this subroutine is called to release a write lock for this read-write lock object, the read-write lock object
will be put in the unlocked state with no owners.

If the call to the `pthread_rwlock_unlock` subroutine results in the read-write lock object becoming
unlocked and there are multiple threads waiting to acquire the read-write lock object for writing, the
scheduling policy is used to determine which thread acquires the read-write lock object for writing. If there
are multiple threads waiting to acquire the read-write lock object for reading, the scheduling policy is used
to determine the order in which the waiting threads acquire the read-write lock object for reading. If there
are multiple threads blocked on rwlock for both read locks and write locks, it is unspecified whether the readers acquire the lock first or whether a writer acquires the lock first.

Results are undefined if any of these subroutines are called with an uninitialized read-write lock.

**Parameters**

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>rwlock</td>
<td>Specifies the read-write lock to be unlocked.</td>
</tr>
</tbody>
</table>

**Return Values**

If successful, the `pthread_rwlock_unlock` subroutine returns zero. Otherwise, an error number is returned to indicate the error.

**Error Codes**

The `pthread_rwlock_unlock` subroutine may fail if:

<table>
<thead>
<tr>
<th>Error Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EINVAL</td>
<td>The value specified by <code>rwlock</code> does not refer to an initialized read-write lock object.</td>
</tr>
<tr>
<td>EPERM</td>
<td>The current thread does not own the read-write lock.</td>
</tr>
</tbody>
</table>

**Related Information**

The `pthread.h` file.

The `pthread_rwlock_init`, `pthread_rwlock_wrlock`, `pthread_rwlockattr_init`, `pthread_rwlock_rdlock` subroutines.

---

**pthread_rwlock_wrlock or pthread_rwlock_trywrlock Subroutines**

**Purpose**

Locks a read-write lock object for writing.

**Library**

Threads Library (libpthreads.a)

**Syntax**

```c
#include <pthread.h>

int pthread_rwlock_wrlock (rwlock)
    pthread_rwlock_t *rwlock;

int pthread_rwlock_trywrlock (rwlock)
    pthread_rwlock_t *rwlock;
```

**Description**

The `pthread_rwlock_wrlock` subroutine applies a write lock to the read-write lock referenced by `rwlock`. The calling thread acquires the write lock if no other thread (reader or writer) holds the read-write lock `rwlock`. Otherwise, the thread blocks (that is, does not return from the `pthread_rwlock_wrlock` call) until it can acquire the lock. Results are undefined if the calling thread holds the read-write lock (whether a read or write lock) at the time the call is made.
Implementations are allowed to favor writers over readers to avoid writer starvation.

The `pthread_rwlock_trywrlock` subroutine applies a write lock like the `pthread_rwlock_wrlock` subroutine, with the exception that the function fails if any thread currently holds `rwlock` (for reading or writing).

Results are undefined if any of these functions are called with an uninitialized read-write lock.

If a signal is delivered to a thread waiting for a read-write lock for writing, upon return from the signal handler the thread resumes waiting for the read-write lock for writing as if it was not interrupted.

Realtime applications may encounter priority inversion when using read-write locks. The problem occurs when a high priority thread 'locks' a read-write lock that is about to be 'unlocked' by a low priority thread, but the low priority thread is preempted by a medium priority thread. This scenario leads to priority inversion; a high priority thread is blocked by lower priority threads for an unlimited period of time. During system design, realtime programmers must take into account the possibility of this kind of priority inversion. They can deal with it in a number of ways, such as by having critical sections that are guarded by read-write locks execute at a high priority, so that a thread cannot be preempted while executing in its critical section.

**Parameters**

`rwlock` Specifies the read-write lock to be locked for writing.

**Return Values**

If successful, the `pthread_rwlock_wrlock` subroutine returns zero. Otherwise, an error number is returned to indicate the error.

The `pthread_rwlock_trywrlock` subroutine returns zero if the lock for writing on the read-write lock object referenced by `rwlock` is acquired. Otherwise an error number is returned to indicate the error.

**Error Codes**

The `pthread_rwlock_trywrlock` subroutine will fail if:

- **EBUSY** The read-write lock could not be acquired for writing because it was already locked for reading or writing.

The `pthread_rwlock_wrlock` and `pthread_rwlock_trywrlock` subroutines may fail if:

- **EINVAL** The value specified by `rwlock` does not refer to an initialized read-write lock object.
- **EDEADLK** The current thread already owns the read-write lock for writing or reading.

**Related Information**

The `pthread.h` file.

The `pthread_rwlock_init`, `pthread_rwlock_unlock`, `pthread_rwlockattr_init` subroutines.

The `pthread_rwlock_rdlock` or `pthread_rwlock_tryrdlock` Subroutines.
**pthread_rwlockattr_init or pthread_rwlockattr_destroy Subroutines**

**Purpose**
Initializes and destroys read-write lock attributes object.

**Library**
Threads Library (`libpthreads.a`)

**Syntax**
```c
#include <pthread.h>

int pthread_rwlockattr_init (attr)
    pthread_rwlockattr_t *attr;

int pthread_rwlockattr_destroy (attr)
    pthread_rwlockattr_t *attr;
```

**Description**
The `pthread_rwlockattr_init` subroutine initializes a read-write lock attributes object `attr` with the default value for all of the attributes defined by the implementation. Results are undefined if `pthread_rwlockattr_init` is called specifying an already initialized read-write lock attributes object.

After a read-write lock attributes object has been used to initialize one or more read-write locks, any function affecting the attributes object (including destruction) does not affect any previously initialized read-write locks.

The `pthread_rwlockattr_destroy` subroutine destroys a read-write lock attributes object. The effect of subsequent use of the object is undefined until the object is re-initialized by another call to `pthread_rwlockattr_init`. An implementation may cause `pthread_rwlockattr_destroy` to set the object referenced by `attr` to an invalid value.

**Parameters**

- `attr` Specifies a read-write lock attributes object to be initialized or destroyed.

**Return Value**
If successful, the `pthread_rwlockattr_init` and `pthread_rwlockattr_destroy` subroutines return zero. Otherwise, an error number is returned to indicate the error.

**Error Codes**
The `pthread_rwlockattr_init` subroutine will fail if:

- `ENOMEM` Insufficient memory exists to initialize the read-write lock attributes object.

The `pthread_rwlockattr_destroy` subroutine will fail if:

- `EINVAL` The value specified by `attr` is invalid.

**Related Information**
The `pthread.h` file.
The `pthread_rwlock_init` subroutine initializes a read-write lock attributes object. The `pthread_rwlock_unlock` subroutine unlocks a read-write lock. The `pthread_rwlock_wrlock` subroutine acquires a write lock, while the `pthread_rwlock_rdlock` subroutine acquires a read lock. The `pthread_rwlockattr_getpshared` subroutine obtains the process-shared attribute of a read-write lock attributes object, and the `pthread_rwlockattr_setpshared` subroutine sets the process-shared attribute.

### pthread_rwlockattr_getpshared or pthread_rwlockattr_setpshared Subroutines

#### Purpose
Gets and sets process-shared attribute of read-write lock attributes object.

#### Library
Threads Library (`libpthreads.a`) library

#### Syntax
```c
#include <pthread.h>

int pthread_rwlockattr_getpshared (attr, pshared)
const pthread_rwlockattr_t *attr;
int *pshared;

int pthread_rwlockattr_setpshared (attr, pshared)
pthread_rwlockattr_t *attr;
int pshared;
```

#### Description
The process-shared attribute is set to `PTHREAD_PROCESS_SHARED` to permit a read-write lock to be operated upon by any thread that has access to the memory where the read-write lock is allocated, even if the read-write lock is allocated in memory that is shared by multiple processes. If the process-shared attribute is `PTHREAD_PROCESS_PRIVATE`, the read-write lock will only be operated upon by threads created within the same process as the thread that initialized the read-write lock; if threads of differing processes attempt to operate on such a read-write lock, the behavior is undefined. The default value of the process-shared attribute is `PTHREAD_PROCESS_PRIVATE`.

The `pthread_rwlockattr_getpshared` subroutine obtains the value of the process-shared attribute from the initialized attributes object referenced by `attr`. The `pthread_rwlockattr_setpshared` subroutine is used to set the process-shared attribute in an initialized attributes object referenced by `attr`.

#### Parameters
- `attr`: Specifies the initialized attributes object.
- `pshared`: Specifies the process-shared attribute of read-write lock attributes object to be gotten and set.

#### Return Values
If successful, the `pthread_rwlockattr_setpshared` subroutine returns zero. Otherwise, an error number is returned to indicate the error.

Upon successful completion, the `pthread_rwlockattr_getpshared` subroutine returns zero and stores the value of the process-shared attribute of `attr` into the object referenced by the `pshared` parameter. Otherwise an error number is returned to indicate the error.
Error Codes
The `pthread_rwlockattr_getpshared` and `pthread_rwlockattr_setpshared` subroutines will fail if:

EINVAL    The value specified by `attr` is invalid.

The `pthread_rwlockattr_setpshared` subroutine will fail if:
EINVAL    The new value specified for the attribute is outside the range of legal values for that attribute.

Related Information
The `pthread.h` file.

The `pthread_rwlock_init` subroutine, `pthread_rwlock_unlock` subroutine, `pthread_rwlock_wrlock` subroutine, `pthread_rwlock_rdlock` subroutine, and `pthread_rwlockattr_init` subroutine.

pthread_self Subroutine

Purpose
Returns the calling thread's ID.

Library
Threads Library (libpthreads.a)

Syntax
```
#include <pthread.h>

pthread_t pthread_self (void);
```

Description
The `pthread_self` subroutine returns the calling thread's ID.

Note: The `pthread.h` header file must be the first included file of each source file using the threads library. Otherwise, the -D_THREAD_SAFE compilation flag should be used, or the cc_r compiler used. In this case, the flag is automatically set.

Return Values
The calling thread's ID is returned.

Errors
No errors are defined.

The `pthread_self` function will not return an error code of EINTR.

Related Information
The `pthread_create` subroutine, `pthread_equal` subroutine.
The `pthread.h` file.

Creating Threads in *AIX 5L Version 5.3 General Programming Concepts: Writing and Debugging Programs.*

**pthread_setcancelstate, pthread_setcanceltype, or pthread_testcancel Subroutines**

**Purpose**
Sets the calling thread's cancelability state.

**Library**
Threads Library (`libpthreads.a`)  

**Syntax**
```c
#include <pthread.h>

int pthread_setcancelstate (state, oldstate);
int state;
int *oldstate;

int pthread_setcanceltype (type, oldtype);
int type;
int *oldtype;

int pthread_testcancel (void)
```

**Description**
The `pthread_setcancelstate` subroutine atomically both sets the calling thread's cancelability state to the indicated state and returns the previous cancelability state at the location referenced by `oldstate`. Legal values for state are `PTHREAD_CANCEL_DISABLE` and `PTHREAD_CANCEL_ENABLE`.  

The `pthread_setcanceltype` subroutine atomically both sets the calling thread's cancelability type to the indicated type and returns the previous cancelability type at the location referenced by `oldtype`. Legal values for type are `PTHREAD_CANCEL_DEFERRED` and `PTHREAD_CANCEL_ASYNCHRONOUS`.  

The cancelability state and type of any newly created threads, including the thread in which `main` was first invoked, are `PTHREAD_CANCEL_ENABLE` and `PTHREAD_CANCEL_DEFERRED` respectively.

The `pthread_testcancel` subroutine creates a cancellation point in the calling thread. The `pthread_testcancel` subroutine has no effect if cancelability is disabled.

**Parameters**
- `state` Specifies the new cancelability state to set. It must have one of the following values:
  - `PTHREAD_CANCEL_DISABLE`
    - Disables cancelability; the thread is not cancelable. Cancellation requests are held pending.
  - `PTHREAD_CANCEL_ENABLE`
    - Enables cancelability; the thread is cancelable, according to its cancelability type. This is the default value.
- `oldstate` Points to where the previous cancelability state value will be stored.
- `type` Specifies the new cancelability type to set.
- `oldtype` Points to where the previous cancelability type value will be stored.
Return Values
If successful, the `pthread_setcancelstate` and `pthread_setcanceltype` subroutines return zero. Otherwise, an error number is returned to indicate the error.

Error Codes
The `pthread_setcancelstate` subroutine will fail if:

**EINVAL**  The specified state is not `PTHREAD_CANCEL_ENABLE` or `PTHREAD_CANCEL_DISABLE`.

The `pthread_setcanceltype` subroutine will fail if:

**EINVAL**  The specified type is not `PTHREAD_CANCEL_DEFERRED` or `PTHREAD_CANCELASYNCHRONOUS`.

These subroutines will not return an error code of EINTR.

Related Information
The `pthread_cancel` ("pthread_cancel Subroutine" on page 1209) subroutine.

The `pthread.h` file.

Terminating Threads in AIX 5L Version 5.3 General Programming Concepts: Writing and Debugging Programs.

pthread_setschedparam Subroutine

Purpose
Sets schedpolicy and schedparam attributes of a thread.

Library
Threads Library (libpthreads.a)

Syntax
```c
#include <pthread.h>
#include <sys/sched.h>

int pthread_setschedparam (thread, schedpolicy, schedparam)

pthread_t thread;
int schedpolicy;
const struct sched_param *schedparam;
```

Description
The `pthread_setschedparam` subroutine dynamically sets the schedpolicy and schedparam attributes of the thread `thread`. The schedpolicy attribute specifies the scheduling policy of the thread. The schedparam attribute specifies the scheduling parameters of a thread created with this attributes object. The sched_priority field of the `sched_param` structure contains the priority of the thread. It is an integer value.

If the target thread has system contention scope, the process must have root authority to set the scheduling policy to either `SCHED_FIFO` or `SCHED_RR`. 
Note: The pthread.h header file must be the first included file of each source file using the threads library. Otherwise, the -D_THREAD_SAFE compilation flag should be used, or the cc_r compiler used. In this case, the flag is automatically set.

This subroutine is part of the Base Operating System (BOS) Runtime. The implementation of this subroutine is dependent on the priority scheduling POSIX option. The priority scheduling POSIX option is implemented in the operating system.

Parameters

- **thread**: Specifies the target thread.
- **schedpolicy**: Points to the schedpolicy attribute to set. It must have one of the following values:
  - **SCHED_FIFO**: Denotes first-in first-out scheduling.
  - **SCHED_RR**: Denotes round-robin scheduling.
  - **SCHED_OTHER**: Denotes the default operating system scheduling policy. It is the default value. If schedpolicy is SCHED_OTHER, then sched_priority must be in the range from 40 to 80, where 40 is the least favored priority and 80 is the most favored.

  **Note**: Priority of threads with a process contention scope and a SCHED_OTHER policy is controlled by the kernel; thus, setting the priority of such a thread has no effect. However, priority of threads with a system contention scope and a SCHED_OTHER policy can be modified. The modification directly affects the underlying kernel thread nice value.

  **schedparam**: Points to where the scheduling parameters to set are stored. The sched_priority field must be in the range from 1 to 127, where 1 is the least favored priority, and 127 the most favored. If schedpolicy is SCHED_OTHER, then sched_priority must be in the range from 40 to 80, where 40 is the least favored priority and 80 is the most favored.

  **Note**: Prior to AIX 5.3, users are not permitted to change the priority of a thread when setting its scheduling policy to SCHED_OTHER. In this case, the priority is managed directly by the kernel, and the only legal value that can be passed to pthread_setschedparam is DEFAULT_PRIO, which is defined in pthread.h as 1. All other passed values are ignored.

  Beginning with AIX 5.3, users can change the priority of a thread when setting its scheduling policy to SCHED_OTHER. The legal values that can be passed to pthread_setschedparam range from 40 to 80. Only privileged users can set a priority higher than 60. A value ranging from 1 to 39 provides the same priority as 40, and a value ranging from 81 to 127 provides the same priority as 80.

Return Values

Upon successful completion, 0 is returned. Otherwise, an error code is returned.

Error Codes

The pthread_setschedparam subroutine is unsuccessful if the following is true:

- **EINVAL**: The thread or schedparam parameters are not valid.
- **ENOSYS**: The priority scheduling POSIX option is not implemented.
- **ENOTSUP**: The value of the schedpolicy or schedparam attributes are not supported.
- **EPERM**: The target thread has insufficient permission to perform the operation or is already engaged in a mutex protocol.
- **ESRCH**: The thread thread does not exist.
Related Information

The `pthread_getschedparam` subroutine, `pthread_attr_setschedparam` subroutine.

Threads Scheduling in AIX 5L Version 5.3 General Programming Concepts: Writing and Debugging Programs.

Threads Library Options in AIX 5L Version 5.3 General Programming Concepts: Writing and Debugging Programs.

---

pthread_setschedprio Subroutine

Purpose

Dynamic thread scheduling parameters access (REALTIME THREADS).

Syntax

```c
#include <pthread.h>

int pthread_setschedprio(pthread_t thread, int prio);
```

Description

The `pthread_setschedprio()` function sets the scheduling priority for the thread whose thread ID is given by `thread` to the value given by `prio`. If a thread whose policy or priority has been modified by `pthread_setschedprio()` is a running thread or is runnable, the effect on its position in the thread list depends on the direction of the modification as follows:

- If the priority is raised, the thread becomes the tail of the thread list.
- If the priority is unchanged, the thread does not change position in the thread list.
- If the priority is lowered, the thread becomes the head of the thread list.

Valid priorities are within the range returned by the `sched_get_priority_max()` and `sched_get_priority_min()`.

If the `pthread_setschedprio()` function fails, the scheduling priority of the target thread remains unchanged.

Rationale

The `pthread_setschedprio()` function provides a way for an application to temporarily raise its priority and then lower it again, without having the undesired side-effect of yielding to other threads of the same priority. This is necessary if the application is to implement its own strategies for bounding priority inversion, such as priority inheritance or priority ceilings. This capability is especially important if the implementation does not support the Thread Priority Protection or Thread Priority Inheritance options; but even if those options are supported, this capability is needed if the application is to bound priority inheritance for other resources, such as semaphores.

The standard developers considered that, while it might be preferable conceptually to solve this problem by modifying the specification of `pthread_setschedparam()`, it was too late to make such a change, because there might be implementations that would need to be changed. Therefore, this new function was introduced.

Return Values

If successful, the `pthread_setschedprio()` function returns 0; otherwise, an error number is returned to indicate the error.
Error Codes

The `pthread_setschedprio()` function might fail if:

- **EINVAL** The value of `prio` is invalid for the scheduling policy of the specified thread.
- **ENOTSUP** An attempt was made to set the priority to an unsupported value.
- **EPERM** The caller does not have the appropriate permission to set the scheduling policy of the specified thread.
- **EPERM** The implementation does not allow the application to modify the priority to the value specified.
- **ESRCH** The value specified by `thread` does not refer to an existing thread.

The `pthread_setschedprio` function does not return an error code of [EINTR].

Related Information

- “`pthread_getschedparam Subroutine`” on page 1234, “`pthread_setschedparam Subroutine`” on page 1276.

The `pthread.h` file in AIX 5L Version 5.3 Files Reference.

pthread_sigmask Subroutine

**Purpose**
Examines and changes blocked signals.

**Library**
Threads Library (libpthreads.a)

**Syntax**

```c
#include <signal.h>

int pthread_sigmask (how, set, oset)
int how;
const sigset_t *set;
sigset_t *oset;
```

**Description**

pthread_signal_to_cancel_np Subroutine

**Purpose**
Cancels the specified thread.

**Library**
Threads Library (libpthreads.a)

**Syntax**

```c
#include <pthread.h>
```
int pthread_signal_to_cancel_np (sigset_t *sigset, thread_t *thread);

Description
The `pthread_signal_to_cancel_np` subroutine cancels the target thread `thread` by creating a handler thread. The handler thread calls the `sigwait` subroutine with the `sigset` parameter, and cancels the target thread when the `sigwait` subroutine returns. Successive calls to this subroutine override the previous ones.

Notes:
1. The `pthread.h` header file must be the first included file of each source file using the threads library. Otherwise, the `-D_THREAD_SAFE` compilation flag should be used, or the cc_r compiler used. In this case, the flag is automatically set.
2. The `pthread_signal_to_cancel_np` subroutine is not portable.

This subroutine is not POSIX compliant and is provided only for compatibility with DCE threads. It should not be used when writing new applications.

Parameters
- **sigset**: Specifies the set of signals to wait on.
- **thread**: Specifies the thread to cancel.

Return Values
Upon successful completion, 0 is returned. Otherwise, an error code is returned.

Error Codes
The `pthread_signal_to_cancel_np` subroutine is unsuccessful if the following is true:
- **EAGAIN**: The handler thread cannot be created.
- **EINVAL**: The `sigset` or `thread` parameters are not valid.

Related Information
The `pthread_cancel` subroutine, `sigwait` subroutine.

---

**pthread_spin_destroy or pthread_spin_init Subroutine**

Purpose
Destroys or initializes a spin lock object.

Syntax
```c
#include <pthread.h>

int pthread_spin_destroy(pthread_spinlock_t *lock);
int pthread_spin_init(pthread_spinlock_t *lock, int pshared);
```

Description
The `pthread_spin_destroy` subroutine destroys the spin lock referenced by `lock` and releases any resources used by the lock. The effect of subsequent use of the lock is undefined until the lock is
reinitialized by another call to the `pthread_spin_init` subroutine. The results are undefined if the `pthread_spin_destroy` subroutine is called when a thread holds the lock, or if this function is called with an uninitialized thread spin lock.

The `pthread_spin_init` subroutine allocates any resources required to use the spin lock referenced by lock and initializes the lock to an unlocked state.

If the Thread Process-Shared Synchronization option is supported and the value of `pshared` is PTHREAD_PROCESS_SHARED, the implementation shall permit the spin lock to be operated upon by any thread that has access to the memory where the spin lock is allocated, even if it is allocated in memory that is shared by multiple processes.

If the Thread Process-Shared Synchronization option is supported and the value of `pshared` is PTHREAD_PROCESS_PRIVATE, or if the option is not supported, the spin lock shall only be operated upon by threads created within the same process as the thread that initialized the spin lock. If threads of differing processes attempt to operate on such a spin lock, the behavior is undefined.

The results are undefined if the `pthread_spin_init` subroutine is called specifying an already initialized spin lock. The results are undefined if a spin lock is used without first being initialized.

If the `pthread_spin_init` subroutine function fails, the lock is not initialized and the contents of lock are undefined.

Only the object referenced by lock may be used for performing synchronization.

The result of referring to copies of that object in calls to the `pthread_spin_destroy` subroutine, `pthread_spin_lock` subroutine, `pthread_spin_trylock` subroutine, or the `pthread_spin_unlock` subroutine is undefined.

**Return Values**

Upon successful completion, these functions shall return zero; otherwise, an error number shall be returned to indicate the error.

**Error Codes**

- **EBUSY** The implementation has detected an attempt to initialize or destroy a spin lock while it is in use (for example, while being used in a `pthread_spin_lock` call) by another thread.
- **EINVAL** The value specified by the `lock` parameter is invalid.

The `pthread_spin_init` subroutine will fail if:

- **EAGAIN** The system lacks the necessary resources to initialize another spin lock.
- **ENOMEM** Insufficient memory exists to initialize the lock.

**Related Information**

The "pthread_spin_lock or pthread_spin_trylock Subroutine," "pthread_spin_unlock Subroutine" on page 1282.

**pthread_spin_lock or pthread_spin_trylock Subroutine**

**Purpose**

Locks a spin lock object.
Syntax
#include <pthread.h>

int pthread_spin_lock(pthread_spinlock_t *lock);
int pthread_spin_trylock(pthread_spinlock_t *lock);

Description
The **pthread_spin_lock** subroutine locks the spin lock referenced by the *lock* parameter. The calling thread shall acquire the lock if it is not held by another thread. Otherwise, the thread spins (that is, does not return from the **pthread_spin_lock** call) until the lock becomes available. The results are undefined if the calling thread holds the lock at the time the call is made. The **pthread_spin_trylock** subroutine locks the spin lock referenced by the *lock* parameter if it is not held by any thread. Otherwise, the function fails.

The results are undefined if any of these subroutines is called with an uninitialized spin lock.

Return Values
Upon successful completion, these functions return zero; otherwise, an error number is returned to indicate the error.

Error Codes
EINVAL The value specified by the *lock* parameter does not refer to an initialized spin lock object.

The **pthread_spin_lock** subroutine fails if:
EDEADLK The calling thread already holds the lock.

The **pthread_spin_trylock** subroutine fails if:
EBUSY A thread currently holds the lock.

Related Information
"pthread_spin_destroy or pthread_spin_init Subroutine" on page 1280
"pthread_spin_unlock Subroutine."

**pthread_spin_unlock Subroutine**

Purpose
Unlocks a spin lock object.

Syntax
#include <pthread.h>

int pthread_spin_unlock(pthread_spinlock_t *lock);

Description
The **pthread_spin_unlock** subroutine releases the spin lock referenced by the *lock* parameter which was locked using the **pthread_spin_lock** subroutine or the **pthread_spin_trylock** subroutine. The results are undefined if the lock is not held by the calling thread. If there are threads spinning on the lock when the **pthread_spin_unlock** subroutine is called, the lock becomes available and an unspecified spinning thread shall acquire the lock.
The results are undefined if this subroutine is called with an uninitialized thread spin lock.

**Return Values**

Upon successful completion, the `pthread_spin_unlock` subroutine returns zero; otherwise, an error number is returned to indicate the error.

**Error Codes**

- **EINVAL** An invalid argument was specified.
- **EPERM** The calling thread does not hold the lock.

**Related Information**

- "pthread_spin_destroy or pthread_spin_init Subroutine" on page 1280
- "pthread_spin_lock or pthread_spin_trylock Subroutine" on page 1281

**pthread_suspend_np, pthread_unsuspend_np and pthread_continue_np Subroutine**

**Purpose**

Suspends and resume execution of the pthread specified by `thread`.

**Library**

Threads Library (libpthreads.a)

**Syntax**

```c
#include <pthread.h>

pthread_t thread;
int pthread_suspend_np(thread);
int pthread_unsuspend_np(thread);
int pthread_continue_np(thread);
```

**Description**

The `pthread_suspend_np` subroutine immediately suspends the execution of the pthread specified by `thread`. On successful return from `pthread_suspend_np`, the suspended pthread is no longer executing. If `pthread_suspend_np` is called for a pthread that is already suspended, the pthread is unchanged and `pthread_suspend_np` returns successful.

Deadlock can occur if `pthread_suspend_np` is used with the following pthread functions.

- `pthread_getusage_np`
- `pthread_cancel`
- `pthread_detach`
- `pthread_join`
- `pthread_getunique_np`
- `pthread_join_np`
- `pthread_setschedparam`
- `pthread_setschedparam`
- `pthread_kill`

To prevent deadlock, PTHREAD_SUSPENDIBLE=ON should be set.
The `pthread_unsuspend_np` routine decrements the suspend count and once the count is zero, the routine resumes the execution of a suspended pthread. If `pthread_unsuspend_np` is called for a pthread that is not suspended, the pthread is unchanged and `pthread_unsuspend_np` returns successful.

The `pthread_continue_np` routine clears the suspend count and resumes the execution of a suspended pthread. If `pthread_continue_np` is called for a pthread that is not suspended, the pthread is unchanged and `pthread_continue_np` returns successful.

A suspended pthread will not be awakened by a signal. The signal stays pending until the execution of pthread is resumed by `pthread_continue_np`.

**Note:** Using `pthread_suspend_np` should only be used by advanced users because improper use of this subcommand can lead to application deadlock or the target thread may be suspended holding application locks.

**Parameters**

`thread` Specifies the target thread.

**Return Values**

Zero is returned when successful. A nonzero value indicates an error.

**Error Codes**

If any of the following conditions occur, `pthread_suspend_np`, `pthread_unsuspend_np` and `pthread_continue_np` fail and return the corresponding value:

- **ESRCH** The target thread specified by `thread` attribute cannot be found in the current process.

---

**pthread_unlock_global_np Subroutine**

**Purpose**

Unlocks the global mutex.

**Library**

Threads Library (libpthreads.a)

**Syntax**

```c
#include <pthread.h>
void pthread_unlock_global_np ()
```

**Description**

The `pthread_unlock_global_np` subroutine unlocks the global mutex when each call to the `pthread_lock_global_np` subroutine is matched by a call to this routine. For example, if a thread called the `pthread_lock_global_np` three times, the global mutex is unlocked after the third call to the `pthread_unlock_global_np` subroutine.

If no threads are waiting for the global mutex, it becomes unlocked with no current owner. If one or more threads are waiting to lock the global mutex, exactly one thread returns from its call to the `pthread_lock_global_np` subroutine.
**Notes:**

1. The `pthread.h` header file must be the first included file of each source file using the threads library. Otherwise, the `-D_THREAD_SAFE` compilation flag should be used, or the cc_r compiler used. In this case, the flag is automatically set.

2. The `pthread_unlock_global_np` subroutine is not portable.

This subroutine is not POSIX compliant and is provided only for compatibility with DCE threads. It should not be used when writing new applications.

**Related Information**

The `pthread_lock_global_np` subroutine.

**Using Mutexes** in *AIX 5L Version 5.3 General Programming Concepts: Writing and Debugging Programs.*

---

**pthread_yield Subroutine**

**Purpose**

Forces the calling thread to relinquish use of its processor.

**Library**

Threads Library (`libpthreads.a`)

**Syntax**

```c
#include <pthread.h>

void pthread_yield ()
```

**Description**

The `pthread_yield` subroutine forces the calling thread to relinquish use of its processor, and to wait in the run queue before it is scheduled again. If the run queue is empty when the `pthread_yield` subroutine is called, the calling thread is immediately rescheduled.

If the thread has global contention scope (`PTHREAD_SCOPE_SYSTEM`), calling this subroutine acts like calling the `yield` subroutine. Otherwise, another local contention scope thread is scheduled.

The `pthread.h` header file must be the first included file of each source file using the threads library. Otherwise, the `-D_THREAD_SAFE` compilation flag should be used, or the cc_r compiler used. In this case, the flag is automatically set.

**Related Information**

The `yield` subroutine and the `sched_yield` subroutine.

**Threads Scheduling** in *AIX 5L Version 5.3 General Programming Concepts: Writing and Debugging Programs.*

**Threads Library Options** in *AIX 5L Version 5.3 General Programming Concepts: Writing and Debugging Programs.*
ptrace, ptracex, ptrace64 Subroutine

Purpose
Traces the execution of another process.

Library
Standard C Library (libc.a)

Syntax
```
#include <sys/reg.h>
#include <sys/ptrace.h>
#include <sys/ldr.h>
int ptrace (Request, Identifier, Address, Data, Buffer)
int Request;
int Identifier;
int *Address;
int Data;
int *Buffer;

int ptracex (request, identifier, addr, data, buff)
int request;
int identifier;
long long addr;
int data;
int *buff;

int ptrace64 (request, identifier, addr, data, buff)
int request;
long long identifier;
long long addr;
int data;
int *buff;
```

Description
The ptrace subroutine allows a 32-bit process to trace the execution of another process. The ptrace subroutine is used to implement breakpoint debugging.

A debugged process runs normally until it encounters a signal. Then it enters a stopped state and its debugging process is notified with the wait subroutine.

Exception: If the process encounters the SIGTRAP signal, a signal handler for SIGTRAP exists, and fast traps (“Fast Trap Instructions” on page 1287) have been enabled for the process, then the signal handler is called and the debugger is not notified. This exception only applies to AIX 4.3.3 and later releases.

While the process is in the stopped state, the debugger examines and modifies the memory image of the process being debugged by using the ptrace subroutine. For multi-threaded processes, the geth th r st s subroutine identifies each kernel thread in the debugged process. Also, the debugging process can cause the debugged process to terminate or continue, with the possibility of ignoring the signal that caused it to stop.

As a security measure, the ptrace subroutine inhibits the set-user-ID facility on subsequent exec subroutines.
(This paragraph only applies to AIX 4.3.2 and later releases.) When a process running under ptrace control calls load or unload, the debugger is notified and the W_SLWTEd flag is set in the status returned by wait. (A 32-bit process calling loadbind is stopped as well.) If the process being debugged has added modules in the shared library to its address space, the modules are added to the process’s private copy of the shared library segments. If shared library modules are removed from a process’s address space, the modules are deleted from the process’s private copy of the library text segment by freeing the pages that contain the module. No other changes to the segment are made, and existing breakpoints do not have to be reinserted.

To allow a debugger to generate code more easily (in order to handle fast trap instructions, for example), memory from the end of the main program up to the next segment boundary can be modified. That memory is read-only to the process but can be modified by the debugger.

When a process being traced forks, the child process is initialized with the unmodified main program and shared library segment, effectively removing breakpoints in these segments in the child process. If multiprocess debugging is enabled, new copies of the main program and shared library segments are made. Modifications to privately loaded modules, however, are not affected by a fork. These breakpoints will remain in the child process, and if these breakpoints are run, a SIGTRAP signal is generated and delivered to the process.

If a traced process initiates an exec subroutine, the process stops before executing the first instruction of the new image and returns the SIGTRAP signal.

Note: The ptrace and ptracex subroutines are not supported in 64-bit mode.

Fast Trap Instructions
Sometimes, allowing the process being debugged to handle certain trap instructions is useful, instead of causing the process to stop and notify the debugger. You can use this capability to patch running programs or programs whose source codes are not available. For a process to use this capability, you must enable fast traps, which requires you to make a ptrace call from a debugger on behalf of the process.

To let a process handle fast traps, a debugger uses the ptrace (PT_SET, pid, 0, PTFLAG_FAST_TRAP, 0) subroutine call. Cancel this capability with the ptrace (PT_CLEAR, pid, 0, PTFLAG_FAST_TRAP, 0) subroutine call. If a process is able to handle fast traps when the debugger detaches, the fast trap capability remains in effect. Consequently, when another debugger attaches to that process, fast trap processing is still enabled. When no debugger is attached to a process, SIGTRAP signals are handled in the same manner, regardless of whether fast traps are enabled.

A fast trap instruction is an unconditional trap immediate instruction in the form twi 14,r13,0xNXXX. This instruction has the binary form 0x0ddfNXXX, where N is a hex digit >=8 and XXX are any three hex digits. By using different values of 0xNXXX, a debugger can generate different fast trap instructions, allowing a signal handler to quickly determine how to handle the signal. (The fast trap instruction is defined by the macro _PTRACE_FASTTRAP. The _PTRACE_FASTTRAP_MASK macro can be used to check whether a trap is a fast trap.)

Usually, a fast trap instruction is treated like any other trap instruction. However, if a process has a signal handler for SIGTRAP, the signal is not blocked, and the fast trap capability is enabled, then the signal handler is called and the debugger is not notified.

A signal handler can logically AND the trap instruction with _PTRACE_FASTTRAP_NUM (0x7FFF) to obtain an integer identifying which trap instruction was run.

Fast data watchpoint
In AIX 5.3 ML5 and later, ptrace supports the ability to enable fast watchpoint trap handling. This is similar to fast trap instruction handling in that when it is enabled. Processes that have a signal handler for
SIGTRAP will have the handler called when a watchpoint trap is encountered. In the SIGTRAP signal handler, the traced process can detect a fast watchpoint trap by checking the SI_FAST_WATCH in the _si_flags of the siginfo_t that is passed to the handler. The fast watchpoint handling employs trap-after semantics, which means that the store to the watched location is completed before calling the trap handler, so the instruction address pointer in the signal context that is passed to the handler will point to the instruction following the instruction that caused the trap.

Thread-level tracing
In AIX 5.3 ML5 and later, ptrace supports setting breakpoints and watchpoints per-thread for system scope (1:1) threads. With these, the tracing process (debugger) is only notified when the specific thread of interest has encountered a trap. This provides an efficient means for debuggers to trace individual threads of interest since it doesn't have to filter “false hit” notifications. See the PTT_WATCH, PTT_SET_TRAP, and PTT_CLEAR_TRAP request types below for the usage description.

The ptrace programming model remains unchanged with thread-level breakpoints and watchpoints in that the attachment is still done at the process level, and the target process stops and notifies the tracing process upon encountering a trap. The tracing process can detect that the traced process has stopped for a thread-level trap by checking the TTHRDCALLS flag (in ti_flag2) of the stopping thread (the thread with TTRCSIG set in ti_flag). These flags can be checked by calling getthrds64 on the target process.

Other behaviors that are specific to thread-level tracing:

- Thread-level breakpoints
  - Clear automatically when all threads for which the breakpoint is active have terminated.
  - Not supported for multiprocess debugging (PT_MULTI). They are cleared upon fork and exec.

- Thread-level watchpoints
  - Newly created threads inherit the process-level watch location.
  - Not inherited across fork and exec.

For the 64-bit Process
Use ptracex where the debuggee is a 64-bit process and the operation requested uses the third (Address) parameter to reference the debuggee’s address space or is sensitive to register size. Note that ptracex and ptrace64 will also support 32-bit debugees.

If returning or passing an int doesn’t work for a 64-bit debuggee (for example, PT_READ_GPR), the buffer parameter takes the address for the result. Thus, with the ptracex subroutine, PT_READ_GPR and PT_WRITE_GPR take a pointer to an 8 byte area representing the register value.

In general, ptracex supports all the calls that ptrace does when they are modified for any that are extended for 64-bit addresses (for example, GPRs, LR, CTR, IAR, and MSR). Anything whose size increases for 64-bit processes must be allowed for in the obvious way (for example, PT_REGSET must be an array of long longs for a 64-bit debuggee).

Parameters

Request
Determines the action to be taken by the ptrace subroutine and has one of the following values:

PT_ATTACH
This request allows a debugging process to attach a current process and place it into trace mode for debugging. This request cannot be used if the target process is already being traced. The Identifier parameter is interpreted as the process ID of the traced process. The Address, Data, and Buffer parameters are ignored.
If this request is unsuccessful, a value of -1 is returned and the \texttt{errno} global variable is set to one of the following codes:

\textbf{ESRCH}

\textit{Process} ID is not valid; the traced process is a kernel process; the process is currently being traced; or, the debugger or traced process already exists.

\textbf{EPERM}

Real or effective user ID of the debugger does not match that of the traced process, or the debugger does not have root authority.

\textbf{EINVAL}

The debugger and the traced process are the same.

\textbf{PT\_CLEAR}

This request clears an internal flag or capability. The \textit{Data} parameter specifies which flags to clear. The following flag can be cleared:

\textbf{PTFLAG\_FAST\_TRAP}

Disables the special handling of a fast trap instruction \cite{Fast Trap Instructions} on page 1287. This allows all fast trap instructions causing an interrupt to generate a \texttt{SIGTRAP} signal.

The \textit{Identifier} parameter specifies the process ID of the traced process. The \textit{Address} parameter, \textit{Buffer} parameter, and the unused bits in the \textit{Data} parameter are reserved for future use and should be set to 0.

\textbf{PTFLAG\_FAST\_WATCH}

Enables fast watchpoint trap handling. When a watchpoint trap occurs in a process that has a signal handler for \texttt{SIGTRAP}, and the process has fast watchpoints enabled, the signal handler will be called instead of notifying the tracing process.

\textbf{PTT\_CLEAR\_TRAP}

This request type clears thread-level breakpoints.

The \textit{Identifier} parameter is a valid kernel thread ID in the target process (-1 for all). The \textit{Address} parameter is the address of the breakpoint. The \textit{Data} parameter must be 0. The \textit{Buffer} parameter must be NULL.

If the request is unsuccessful, -1 is returned and the \texttt{errno} global variable is set to one of the following:

\textbf{ESRCH}

The \textit{Identifier} parameter does not refer to a valid kernel thread in the target process, or no breakpoint was found for the target thread at the given \textit{Address}.

\textbf{EINVAL}

The \textit{Data} parameter was non-zero or \textit{Buffer} was non-NULL.

\textbf{PT\_CONTINUE}

This request allows the process to resume execution. If the \textit{Data} parameter is 0, all pending signals, including the one that caused the process to stop, are concealed before the process resumes execution. If the \textit{Data} parameter is a valid signal number, the process resumes execution as if it had received that signal. If the \textit{Address} parameter equals 1, the execution continues from where it stopped. If the \textit{Address} parameter is not 1, it is assumed to be the address at which the process should resume execution. Upon
successful completion, the value of the *Data* parameter is returned to the debugging process. The *Identifier* parameter is interpreted as the process ID of the traced process. The *Buffer* parameter is ignored.

If this request is unsuccessful, a value of -1 is returned and the *errno* global variable is set to the following code:

**EIO**  The signal to be sent to the traced process is not a valid signal number.

**Note:** For the PT_CONTINUE request, use `ptracex` or `ptrace64` with a 64-bit debuggee because the resume address needs 64 bits.

**PTT_CONTINUE**

This request asks the scheduler to resume execution of the kernel thread specified by *Identifier*. This kernel thread must be the one that caused the exception. The *Data* parameter specifies how to handle signals:

- If the *Data* parameter is 0, the kernel thread which caused the exception will be resumed as if the signal never occurred.
- If the *Data* parameter is a valid signal number, the kernel thread which caused the exception will be resumed as if it had received that signal.

The *Address* parameter specifies where to resume execution:

- If the *Address* parameter is 1, execution resumes from the address where it stopped.
- If the *Address* parameter contains an address value other than 1, execution resumes from that address.

The *Buffer* parameter should point to a PTTHREADS structure, which contains a list of kernel thread identifiers to be started. This list should be NULL terminated if it is smaller than the maximum allowed.

On successful completion, the value of the *Data* parameter is returned to the debugging process. On unsuccessful completion, the value -1 is returned, and the *errno* global variable is set as follows:

**EINVAL**  The *Identifier* parameter names the wrong kernel thread.

**EIO**  The signal to be sent to the traced kernel thread is not a valid signal number.

**ESRCH**  The *Buffer* parameter names an invalid kernel thread. Each kernel thread in the list must be stopped and belong to the same process as the kernel thread named by the *Identifier* parameter.

**Note:** For the PTT_CONTINUE request, use `ptracex` or `ptrace64` with a 64-bit debuggee because the resume address needs 64 bits.

**PT_DETACH**

This request allows a debugged process, specified by the *Identifier* parameter, to exit trace mode. The process then continues running, as if it had received the signal whose number is contained in the *Data* parameter. The process is no longer traced and does not process any further `ptrace` calls. The *Address* and *Buffer* parameters are ignored.

If this request is unsuccessful, a value of -1 is returned and the *errno* global variable is set to the following code:

**EIO**  Signal to be sent to the traced process is not a valid signal number.
PT_KILL
This request allows the process to terminate the same way it would with an exit subroutine.

PT_LDINFO
This request retrieves a description of the object modules that were loaded by the debugged process. The Identifier parameter is interpreted as the process ID of the traced process. The Buffer parameter is ignored. The Address parameter specifies the location where the loader information is copied. The Data parameter specifies the size of this area. The loader information is retrieved as a linked list of ld_info structures. The first element of the list corresponds to the main executable module. The ld_info structures are defined in the /usr/include/sys/ldr.h file. The linked list is implemented so that the ldinfo_next field of each element gives the offset of the next element from this element. The ldinfo_next field of the last element has the value 0.

Each object module reported is opened on behalf of the debugger process. The file descriptor for an object module is saved in the ldinfo_fd field of the corresponding ld_info structure. The debugger process is responsible for managing the files opened by the ptrace subroutine.

If this request is unsuccessful, a value of -1 is returned and the errno global variable is set to the following code:

ENOMEM
Either the area is not large enough to accommodate the loader information, or there is not enough memory to allocate an equivalent buffer in the kernel.

Note: For the PT_LDINFO request, use ptracex or ptrace64 with a 64-bit debuggee because the source address needs 64 bits.

PT_LDXINFO
This request is similar to the PT_LDINFO request. A linked list of ld_xinfo structures is returned instead of a list of ld_info structures. The first element of the list corresponds to the main executable module. The ld_xinfo structures are defined in the /usr/include/sys/ldr.h file. The linked list is implemented so that the ldinfo_next field of each element gives the offset of the next element from this element. The ldinfo_next field of the last element has the value 0.

Each object module reported is opened on behalf of the debugger process. The file descriptor for an object module is saved in the ldinfo_fd field of the corresponding ld_xinfo structure. The debugger process is responsible for managing the files opened by the ptrace subroutine.

If this request is unsuccessful, a value of -1 is returned and the errno global variable is set to the following code:

ENOMEM
Either the area is not large enough to accommodate the loader information, or there is not enough memory to allocate an equivalent buffer in the kernel.

Note: For the PT_LDXINFO request, use ptracex or ptrace64 with a 64-bit debuggee because the source address needs 64 bits.

PT_MULTI
This request turns multiprocess debugging mode on and off, to allow debugging to continue across fork and exec subroutines. A 0 value for the Data parameter turns multiprocess debugging mode off, while all other values turn it on. When multiprocess debugging mode is in effect, any fork subroutine allows both the traced process and its
newly created process to trap on the next instruction. If a traced process initiated an `exec` subroutine, the process stops before executing the first instruction of the new image and returns the `SIGTRAP` signal. The `Identifier` parameter is interpreted as the process ID of the traced process. The `Address` and `Buffer` parameters are ignored.

Also, when multiprocess debugging mode is enabled, the following values are returned from the `wait` subroutine:

- `W_SEWTED`  
  Process stopped during execution of the `exec` subroutine.

- `W_SFWTED`  
  Process stopped during execution of the `fork` subroutine.

**PT_READ_BLOCK**

This request reads a block of data from the debugged process address space. The `Address` parameter points to the block of data in the process address space, and the `Data` parameter gives its length in bytes. The value of the `Data` parameter must not be greater than 1024. The `Identifier` parameter is interpreted as the process ID of the traced process. The `Buffer` parameter points to the location in the debugging process address space where the data is copied. Upon successful completion, the `ptrace` subroutine returns the value of the `Data` parameter.

If this request is unsuccessful, a value of -1 is returned and the `errno` global variable is set to one of the following codes:

- `EIO`  
  The `Data` parameter is less than 1 or greater than 1024.

- `EIO`  
  The `Address` parameter is not a valid pointer into the debugged process address space.

- `EFAULT`  
  The `Buffer` parameter does not point to a writable location in the debugging process address space.

**Note:** For the `PT_READ_BLOCK` request, use `ptracex` or `ptrace64` with a 64-bit debuggee because the source address needs 64 bits.

**PT_READ_FPR**

This request stores the value of a floating-point register into the location pointed to by the `Address` parameter. The `Data` parameter specifies the floating-point register, defined in the `sys/reg.h` file for the machine type on which the process is run. The `Identifier` parameter is interpreted as the process ID of the traced process. The `Buffer` parameter is ignored.

If this request is unsuccessful, a value of -1 is returned and the `errno` global variable is set to the following code:

- `EIO`  
  The `Data` parameter is not a valid floating-point register. The `Data` parameter must be in the range 256-287.

**PTT_READ_FPRS**

This request writes the contents of the 32 floating point registers to the area specified by the `Address` parameter. This area must be at least 256 bytes long. The `Identifier` parameter specifies the traced kernel thread. The `Data` and `Buffer` parameters are ignored.

**PT_READ_GPR**

This request returns the contents of one of the general-purpose or special-purpose registers of the debugged process. The `Address` parameter specifies the register whose
value is returned. The value of the Address parameter is defined in the sys/reg.h file for the machine type on which the process is run. The Identifier parameter is interpreted as the process ID of the traced process. The Data and Buffer parameters are ignored. The buffer points to long long target area.

**Note:** If ptracex or ptrace64 with a 64-bit debuggee is used for this request, the register value is instead returned to the 8-byte area pointed to by the buffer pointer.

If this request is unsuccessful, a value of -1 is returned and the errno global variable is set to the following code:

- **EIO** The Address is not a valid general-purpose or special-purpose register. The Address parameter must be in the range 0-31 or 128-136.

**PTT_READ_GPRS**

This request writes the contents of the 32 general purpose registers to the area specified by the Address parameter. This area must be at least 128 bytes long.

**Note:** If ptracex or ptrace64 are used with a 64-bit debuggee for the PTT_READ_GPRS request, there must be at least a 256 byte target area. The Identifier parameter specifies the traced kernel thread. The Data and Buffer parameters are ignored.

**PT_READ_I** or **PT_READ_D**

These requests return the word-aligned address in the debugged process address space specified by the Address parameter. On all machines currently supported by AIX Version 4, the PT_READ_I and PT_READ_D instruction and data requests can be used with equal results. The Identifier parameter is interpreted as the process ID of the traced process. The Data parameter is ignored.

If this request is unsuccessful, a value of -1 is returned and the errno global variable is set to the following code:

- **EIO** The Address is not word-aligned, or the Address is not valid. User blocks, kernel segments, and kernel extension segments are not considered as valid addresses.

**Note:** For the PT_READ_I or the PT_READ_D request, use ptracex or ptrace64 with a 64-bit debuggee because the source address needs 64 bits.

**PTT_READ_SPRS**

This request writes the contents of the special purpose registers to the area specified by the Address parameter, which points to a ptsprs structure. The Identifier parameter specifies the traced kernel thread. The Data and Buffer parameters are ignored.

**Note:** For the PTT_READ_SPRS request, use ptracex or ptrace64 with the 64-bit debuggee because the new ptxsprs structure must be used.

**PTT_READ_VEC**

This request reads the vector register state of the specified thread. The data format is a __vmx_context_t structure that contains the 32 vector registers, in addition to the VSCR and VRSAVE registers.

**PT_REATT**

This request allows a new debugger, with the proper permissions, to trace a process that was already traced by another debugger. The Identifier parameter is interpreted as the process ID of the traced process. The Address, Data, and Buffer parameters are ignored.

If this request is unsuccessful, a value of -1 is returned and the errno global variable is set to one of the following codes:
ESRCH
The Identifier is not valid; or the traced process is a kernel process.

EPERM
Real or effective user ID of the debugger does not match that of the traced process, or the debugger does not have root authority.

EINVAL
The debugger and the traced process are the same.

PT_REGSET
This request writes the contents of all 32 general purpose registers to the area specified by the Address parameter. This area must be at least 128 bytes for the 32-bit debuggee or 256 bytes for the 64-bit debuggee. The Identifier parameter is interpreted as the process ID of the traced process. The Data and Buffer parameters are ignored.

If this request is unsuccessful, a value of -1 is returned and the errno global variable is set to the following code:

EIO The Address parameter points to a location outside of the allocated address space of the process.

Note: For the PT_REGSET request, use ptracex or trace64 with the 64-bit debuggee because 64-bit registers requiring 256 bytes are returned.

PT_SET
This request sets an internal flag or capability. The Data parameter indicates which flags are set. The following flag can be set:

PTFLAG_FAST_TRAP Enables the special handling of a fast trap instruction (“Fast Trap Instructions” on page 1287). When a fast trap instruction is run in a process that has a signal handler for SIGTRAP, the signal handler will be called even if the process is being traced.

The Identifier parameter specifies the process ID of the traced process. The Address parameter, Buffer parameter, and the unused bits in the Data parameter are reserved for future use and should be set to 0.

PTT_SET_TRAP
This request type sets thread-level breakpoints.

The Identifier parameter is a valid kernel ID in the target process. The Address parameter is the address in the target process for the breakpoint. The Data parameter is the length of data in Buffer, it must be 4. The Buffer parameter is a pointer to trap instruction to be written.

The system call will not evaluate the contents of the buffer for this request, but by convention, it should contain a single trap instruction.

If the request is unsuccessful, a value of -1 is returned and the errno global variable is set to one of the following:

ENOMEM Could not allocate kernel memory.
The Identifier parameter does not refer to a valid kernel thread in the target process.

EIO The Address parameter does not point to a writable location in the address space of the target process.

EINVAL Data parameter was not 4, or the target thread already has a breakpoint set at Address.

EFAULT The Buffer parameter does not point to a readable location in the caller’s address space.

PT_TRACE_ME
This request must be issued by the debugged process to be traced. Upon receipt of a signal, this request sets the process trace flag, placing the process in a stopped state, rather than the action specified by the sigaction subroutine. The Identifier, Address, Data, and Buffer parameters are ignored. Do not issue this request if the parent process does not expect to trace the debugged process.

As a security measure, the ptrace subroutine inhibits the set-user-ID facility on subsequent exec subroutines, as shown in the following example:

if((childpid = fork()) == 0)
{ /* child process */
    ptrace(PT_TRACE_ME,0,0,0,0);
    execlp( ) /* your favorite exec*/
} else
{ /* parent */
    /* wait for child to stop */
    rc = wait(status)
}

Note: This is the only request that should be performed by the child. The parent should perform all other requests when the child is in a stopped state.

If this request is unsuccessful, a value of -1 is returned and the errno global variable is set to the following code:

ESRCH Process is debugged by a process that is not its parent.

PT_WATCH
This request allows to have a watchpoint on the memory region specified when the debugged process changes the content at the specified memory region.

The Identifier parameter is interpreted as the process ID of the traced process. The Buffer parameter is ignored. The Address parameter specifies beginning of the memory region to be watched. To clear the watchpoint the Address parameter must be NULL. The Data parameter specifies the size of the memory region.

Watchpoints are supported only on the hardware POWER630, POWER5 and POWER6™. Currently the size of the memory region, that is, the parameter Data must be 8 because only 8 byte watchpoint is supported at the hardware level.

If this request is unsuccessful, a value of -1 is returned and the errno global variable is set to the following code:
PTT_WATCH
This request sets and clears thread-level watchpoints.

The Identifier parameter is a valid kernel thread ID in the target process (-1 for all). The Address parameter is the double-worded aligned address to watch. A value of 0 clears the watchpoint. The Data parameter must be 0 (clear) or 8 (set). The Buffer parameter must be NULL.

If the request is unsuccessful, a value of -1 is returned and the errno global variable is set to one of the following:

ESRCH
The Identifier parameter does not refer to a valid kernel thread in the target process.

EPERM
The hardware watchpoint facility is not supported on the platform.

EIO
The requested Address is not a valid, double-worded aligned address in target process address space, or the Address is non-zero and Data is not 8.

PT_WRITE_BLOCK
This request writes a block of data into the debugged process address space. The Address parameter points to the location in the process address space to be written into. The Data parameter gives the length of the block in bytes, and must not be greater than 1024. The Identifier parameter is interpreted as the process ID of the traced process. The Buffer parameter points to the location in the debugging process address space where the data is copied. Upon successful completion, the value of the Data parameter is returned to the debugging process.

If this request is unsuccessful, a value of -1 is returned and the errno global variable is set to one of the following codes:

EIO
The Data parameter is less than 1 or greater than 1024.

EIO
The Address parameter is not a valid pointer into the debugged process address space.

EFAULT
The Buffer parameter does not point to a readable location in the debugging process address space.

Note: For the PT_WRITE_BLOCK request, use ptracex or ptrace64 with the 64-bit debuggee because 64-bit registers requiring 256 bytes are returned.

PT_WRITE_FPR
This request sets the floating-point register specified by the Data parameter to the value specified by the Address parameter. The Identifier parameter is interpreted as the process ID of the traced process. The Buffer parameter is ignored.
If this request is unsuccessful, a value of -1 is returned and the "errno" global variable is set to the following code:

**EIO** The *Data* parameter is not a valid floating-point register. The *Data* parameter must be in the range 256-287.

**PTT_WRITE_FPRS**
This request updates the contents of the 32 floating point registers with the values specified in the area designated by the *Address* parameter. This area must be at least 256 bytes long. The *Identifier* parameter specifies the traced kernel thread. The *Data* and *Buffer* parameters are ignored.

**Note:** If *ptracex* or *ptrace64* are used with a 64-bit debuggee for the **PT_WRITE_GPR** request, the new register value is NOT passed via the *Data* parameter, but is instead passed via the 8-byte area pointed to by the buffer parameter.

If this request is unsuccessful, a value of -1 is returned and the "errno" global variable is set to the following code:

**EIO** The *Address* parameter is not a valid general-purpose or special-purpose register. The *Address* parameter must be in the range 0-31 or 128-136.

**PTT_WRITE_GPRS**
This request updates the contents of the 32 general purpose registers with the values specified in the area designated by the *Address* parameter. This area must be at least 128 bytes long. The *Identifier* parameter specifies the traced kernel thread. The *Data* and *Buffer* parameters are ignored.

**Note:** For the **PTT_WRITE_GPRS** request, use *ptracex* or *ptrace64* with the 64-bit debuggee because 64-bit registers requiring 256 bytes are returned. The buffer points to long long source area.

**PT_WRITE_I or PT_WRITE_D**
These requests write the value of the *Data* parameter into the address space of the debugged process at the word-aligned address specified by the *Address* parameter. On all machines currently supported by AIX Version 4, instruction and data address spaces are not separated. The **PT_WRITE_I** and **PT_WRITE_D** instruction and data requests can be used with equal results. Upon successful completion, the value written into the address space of the debugged process is returned to the debugging process. The *Identifier* parameter is interpreted as the process ID of the traced process. The *Buffer* parameter is ignored.

If this request is unsuccessful, a value of -1 is returned and the "errno" global variable is set to the following code:

**EIO** The *Address* parameter points to a location in a pure procedure space and a copy cannot be made; the *Address* is not word-aligned; or, the *Address* is not valid. User blocks, kernel segments, and kernel extension segments are not considered valid addresses.
Note: For the or PT_WRITE_I or PT_WRITE_D request, use ptracex or ptrace64 with a 64-bit debuggee because the target address needs 64 bits.

PTT_WRITE_SPRS
This request updates the special purpose registers with the values in the area specified by the Address parameter, which points to a ptsprs structure. The Identifier parameter specifies the traced kernel thread. The Data and Buffer parameters are ignored.

Identifier
Determined by the value of the Request parameter.

Address
Determined by the value of the Request parameter.

Data
Determined by the value of the Request parameter.

Buffer
Determined by the value of the Request parameter.

Note: For the PTT_READ_SPRS request, use ptracex or ptrace64 with the 64-bit debuggee because the new ptxsprs structure must be used.

PTT_WRITE_VEC
This request writes the vector register state of the specified thread. The data format is a __vmx_context_t structure that contains the 32 vector registers, in addition to the VSCR and VRSAVE registers.

Error Codes
The ptrace subroutine is unsuccessful when one of the following is true:

EFAULT  The Buffer parameter points to a location outside the debugging process address space.
EINVAL  The debugger and the traced process are the same; or the Identifier parameter does not identify the thread that caused the exception.
EIO  The Request parameter is not one of the values listed, or the Request parameter is not valid for the machine type on which the process is run.
ENOMEM  Either the area is not large enough to accommodate the loader information, or there is not enough memory to allocate an equivalent buffer in the kernel.
ENXIO  The target thread has not referenced the VMX unit and is not currently a VMX thread.
EPERM  The Identifier parameter corresponds to a kernel thread which is stopped in kernel mode and whose computational state cannot be read or written.
ESRCH  The Identifier parameter identifies a process or thread that does not exist, that has not run a ptrace call with the PT_TRACE_ME request, or that is not stopped.

For ptrace: If the debuggee is a 64-bit process, the options that refer to GPRs or SPRs fail with errno = EIO, and the options that specify addresses are limited to 32-bits.

For ptracex or ptrace64: If the debuggee is a 32-bit process, the options that refer to GPRs or SPRs fail with errno = EIO, and the options that specify addresses in the debuggee’s address space that are larger than 2**32 - 1 fail with errno set to EIO.

Also, the options PT_READ_U and PT_WRITE_U are not supported if the debuggee is a 64-bit program (errno = ENOTSUP).

Related Information
The "exec: execl, execle, execcl, execvp, or exec Subroutine” on page 235, "getprocs Subroutine” on page 410, "getthreads Subroutine” on page 438, and "load Subroutine” on page 721.
The `sigaction` subroutine, `unload` subroutine, and `wait`, `waitpid`, or `wait3` subroutine in *AIX 5L Version 5.3 Technical Reference: Base Operating System and Extensions Volume 2*.

The `dbx` command in *AIX 5L Version 5.3 Commands Reference, Volume 2*.

The `sys/ldr.h` file.

### ptsname Subroutine

#### Purpose
Returns the name of a pseudo-terminal device.

#### Library
Standard C Library (*libc.a*)

#### Syntax
```
#include <stdlib.h>

char *ptsname(int FileDescriptor)
```

#### Description
The `ptsname` subroutine gets the path name of the slave pseudo-terminal associated with the master pseudo-terminal device defined by the `FileDescriptor` parameter.

#### Parameters
- `FileDescriptor` Specifies the file descriptor of the master pseudo-terminal device

#### Return Values
The `ptsname` subroutine returns a pointer to a string containing the null-terminated path name of the pseudo-terminal device associated with the file descriptor specified by the `FileDescriptor` parameter. A null pointer is returned and the `errno` global variable is set to indicate the error if the file descriptor does not describe a pseudo-terminal device in the `/dev` directory.

#### Files
- `/dev/*` Terminal device special files.

#### Related Information
- The `ttyname` subroutine.
- The Input and Output Handling Programmer's Overview in *AIX 5L Version 5.3 General Programming Concepts: Writing and Debugging Programs*.

### putc, putchar, fputc, or putw Subroutine

#### Purpose
Writes a character or a word to a stream.
Library
Standard I/O Package (libc.a)

Syntax
#include <stdio.h>

int putc (Character, Stream)
int Character;
FILE *Stream;

int putchar (Character)
int Character;

int fputc (Character, Stream)
int Character;
FILE *Stream;

int putw (Word, Stream)
int Word;
FILE *Stream;

Description
The putc and putchar macros write a character or word to a stream. The fputc and putw subroutines serve similar purposes but are true subroutines.

The putc macro writes the character Character (converted to an unsigned char data type) to the output specified by the Stream parameter. The character is written at the position at which the file pointer is currently pointing, if defined.

The putchar macro is the same as the putc macro except that putchar writes to the standard output.

The fputc subroutine works the same as the putc macro, but fputc is a true subroutine rather than a macro. It runs more slowly than putc, but takes less space per invocation.

Because putc is implemented as a macro, it incorrectly treats a Stream parameter with side effects, such as putc(C, f++)). For such cases, use the fputc subroutine instead. Also, use fputc whenever you need to pass a pointer to this subroutine as a parameter to another subroutine.

The putc and putchar macros have also been implemented as subroutines for ANSI compatibility. To access the subroutines instead of the macros, insert #undef putc or #undef putchar at the beginning of the source file.

The putw subroutine writes the word (int data type) specified by the Word parameter to the output specified by the Stream parameter. The word is written at the position at which the file pointer, if defined, is pointing. The size of a word is the size of an integer and varies from machine to machine. The putw subroutine does not assume or cause special alignment of the data in the file.

After the fputcw, putwc, fputc, putchar, fputs, puts, or putw subroutine runs successfully, and before the next successful completion of a call either to the fflush or fclose subroutine on the same stream or to the exit or abort subroutine, the st_ctime and st_mtime fields of the file are marked for update.

Because of possible differences in word length and byte ordering, files written using the putw subroutine are machine-dependent, and may not be readable using the getw subroutine on a different processor.
With the exception of stderr, output streams are, by default, buffered if they refer to files, or line-buffered if they refer to terminals. The standard error output stream, stderr, is unbuffered by default, but using the freopen subroutine causes it to become buffered or line-buffered. Use the setbuf subroutine to change the stream buffering strategy.

When an output stream is unbuffered, information is queued for writing on the destination file or terminal as soon as it is written. When an output stream is buffered, many characters are saved and written as a block. When an output stream is line-buffered, each line of output is queued for writing on the destination terminal as soon as the line is completed (that is, as soon as a new-line character is written or terminal input is requested).

Parameters

Stream Points to the file structure of an open file.
Character Specifies a character to be written.
Word Specifies a word to be written (not portable because word length and byte-ordering are machine-dependent).

Return Values

Upon successful completion, these functions each return the value written. If these functions fail, they return the constant EOF. They fail if the Stream parameter is not open for writing, or if the output file size cannot be increased. Because the EOF value is a valid integer, you should use the ferror subroutine to detect putw errors.

Error Codes

The fputc subroutine will fail if either the Stream is unbuffered or the Stream buffer needs to be flushed, and:

EAGAIN The O_NONBLOCK flag is set for the file descriptor underlying Stream and the process would be delayed in the write operation.
EBADF The file descriptor underlying Stream is not a valid file descriptor open for writing.
EFBIG An attempt was made to write a file that exceeds the file size of the process limit or the maximum file size.
EFBIG The file is a regular file and an attempt was made to write at or beyond the offset maximum.
EINTR The write operation was terminated due to the receipt of a signal, and either no data was transferred or the implementation does not report partial transfers for this file.

Note: Depending upon which library routine the application binds to, this subroutine may return EINTR. Refer to the [signal Subroutine regarding sa_restart.
EIO A physical I/O error has occurred, or the process is a member of a background process group attempting to perform a write subroutine to its controlling terminal, the TOSTOP flag is set, the process is neither ignoring nor blocking the SIGTTOU signal and the process group of the process is orphaned. This error may also be returned under implementation-dependent conditions.
ENOSPC There was no free space remaining on the device containing the file.
EPIPE An attempt is made to write to a pipe or first-in-first-out (FIFO) that is not open for reading by any process. A SIGPIPE signal will also be sent to the process.

The fputc subroutine may fail if:

ENOMEM Insufficient storage space is available.
ENXIO A request was made of a nonexistent device, or the request was outside the capabilities of the device.
Related Information

The fclose or fflush subroutine, feof, ferror, clearerr, or fileno subroutine, fopen, freopen, or fdopen subroutine, feof, ferror, clearerr, or fileno subroutine, fread or fwrite subroutine, getc, fgetc, getchar, or getw subroutine, getwc, fgetwc, or getwchar subroutine, printf, fprintf, sprintf, NLprintf, NLfprintf, NLsprintf, or wsprintf subroutine, putwc, fputwc, or putwchar subroutine, puts or fputs subroutine.

putconfattrs Subroutine

Purpose

Accesses system information in the system information database.

Library

Security Library (libc.a)

Syntax

```c
#include <usersec.h>
#include <userconf.h>

int putconfattrs (Table, Attributes, Count)
char * Table;
dbattr_t * Attributes;
int Count;
```

Description

The putconfattrs subroutine writes one or more attributes into the system information database. If the database is not already open, the subroutine does an implicit open for reading and writing. Data changed by putconfattrs must be explicitly committed by calling the putconfattr subroutine with a Type parameter specifying the SEC_COMMIT value. Until the data is committed, only get subroutine calls within the process return the written data.

The Attributes array contains information about each attribute that is to be written. The dbattr_t data structure contains the following fields:

- **attr_name**
  The name of the desired attribute.

- **attr_idx**
  Used internally by the putconfattrs subroutine.

- **attr_type**
  The type of the desired attribute. The list of attribute types is defined in the usersec.h header file.

- **attr_flag**
  The results of the request to write the desired attribute.

- **attr_un**
  A union containing the values to be written. Its union members that follow correspond to the definitions of the attr_char, attr_int, attr_long, and attr_llong macros, respectively:

  - **un_char**
    Attributes of type SEC_CHAR and SEC_LIST store a pointer to the value to be written.
Attributes of type **SEC_INT** and **SEC_BOOL** contain the value of the attribute to be written.

Attributes of type **SEC_LONG** contain the value of the attribute to be written.

Attributes of type **SEC_LLONG** contain the value of the attribute to be written.

The authentication domain containing the attribute. The **putconfattrs** subroutine stores the name of the authentication domain that was used to write this attribute if it is not initialized by the caller.

The **putconfattrs** subroutine is responsible for managing the memory referenced by this pointer.

Use the **setuserdb** and **enduserdb** subroutines to open and close the system information database. Failure to explicitly open and close the system information database can result in loss of memory and performance.

**Parameters**

**Table**

The system information table containing the desired attributes. The list of valid system information tables is defined in the **userconf.h** header file.

**Attributes**

A pointer to an array of one or more elements of type **dbattr_t**. The list of system attributes is defined in the **usersec.h** header file.

**Count**

The number of array elements in **Attributes**.

**Security**

Files accessed:

<table>
<thead>
<tr>
<th>Mode</th>
<th>File</th>
</tr>
</thead>
<tbody>
<tr>
<td>rw</td>
<td>/etc/security/ids</td>
</tr>
<tr>
<td>rw</td>
<td>/etc/security/audit/config</td>
</tr>
<tr>
<td>rw</td>
<td>/etc/security/audit/events</td>
</tr>
<tr>
<td>rw</td>
<td>/etc/security/audit/objects</td>
</tr>
<tr>
<td>rw</td>
<td>/etc/security/login.cfg</td>
</tr>
<tr>
<td>rw</td>
<td>/etc/security/portlog</td>
</tr>
<tr>
<td>rw</td>
<td>/etc/security/roles</td>
</tr>
<tr>
<td>rw</td>
<td>/usr/lib/security/methods.cfg</td>
</tr>
<tr>
<td>rw</td>
<td>/usr/lib/security/mkuser.sys</td>
</tr>
</tbody>
</table>

**Return Values**

The **putconfattrs** subroutine, when successfully completed, returns a value of 0. Otherwise, a value of -1 is returned and the **errno** global variable is set to indicate the error.

**Error Codes**

The **putconfattrs** subroutine fails if one or more of the following are true:

**EACCESS**

The system information database could not be accessed for writing.

**EINVAL**

The **Table** parameter is the NULL pointer.

The **Attributes** parameter does not point to valid data for the requested attribute. Limited testing is possible and all errors might not be detected.

**EINVAL**

The **Count** parameter is less than or equal to 0.

**ENOENT**

The specified **Table** does not exist.
If the `putconfattrs` subroutine fails to write an attribute, one or more of the following errors is returned in the `attr_flag` field of the corresponding `Attributes` element:

- **EACCESS** The user does not have access to the attribute specified in the `attr_name` field.
- **EINVAL** The `attr_type` field in the `Attributes` entry contains an invalid type.
- **EINVAL** The `attr_un` field in the `Attributes` entry does not point to a valid buffer or to valid data for this type of attribute. Limited testing is possible and all errors might not be detected.
- **ENOATTR** The `attr_name` field in the `Attributes` entry specifies an attribute that is not defined for this system table.

**Related Information**

The `putuserattr` subroutine ("getuserattr, IDtouser, nextuser, or putuserattr Subroutine" on page 449) subroutine.

The `setuserdb` Subroutine

- List of Security and Auditing Subroutines
- Subroutines Overview
  in AIX 5L Version 5.3 General Programming Concepts: Writing and Debugging Programs.

**putenv Subroutine**

**Purpose**
Sets an environment variable.

**Library**
Standard C Library (`libc.a`)

**Syntax**

```c
int putenv (char *String);
```

**Description**

**Attention:** Unpredictable results can occur if a subroutine passes the `putenv` subroutine a pointer to an automatic variable and then returns while the variable is still part of the environment.

The `putenv` subroutine sets the value of an environment variable by altering an existing variable or by creating a new one. The `String` parameter points to a string of the form `Name=Value`, where `Name` is the environment variable and `Value` is the new value for it.

The memory space pointed to by the `String` parameter becomes part of the environment, so that altering the string effectively changes part of the environment. The space is no longer used after the value of the environment variable is changed by calling the `putenv` subroutine again. Also, after the `putenv` subroutine is called, environment variables are not necessarily in alphabetical order.

The `putenv` subroutine manipulates the `environ` external variable and can be used in conjunction with the `getenv` subroutine. However, the `EnvironmentPointer` parameter, the third parameter to the main subroutine, is not changed.

The `putenv` subroutine uses the `malloc` subroutine to enlarge the environment.

**Parameters**

`String` A pointer to the `Name=Value` string.
**Return Values**
Upon successful completion, a value of 0 is returned. If the `malloc` subroutine is unable to obtain sufficient space to expand the environment, then the `putenv` subroutine returns a nonzero value.

**Related Information**
The `exec: execl, execv, execle, execvp, or execvp, or exact Subroutine` on page 235 subroutine, `getenv` ("getenv Subroutine" on page 360) subroutine, `malloc` ("malloc, free, realloc, calloc, mallopt, mallinfo, mallinfo_heap, alloca, valloc, or posix_memalign Subroutine" on page 769) subroutine.

---

**putgrent Subroutine**

**Purpose**
Updates group descriptions.

**Library**
Standard C Library (`libc.a`)

**Syntax**
```c
int putgrent (grp, fp);
struct group *grp;
FILE *fp;
```

**Description**
The `putgrent` subroutine updates group descriptions. The `grp` parameter is a pointer to a group structure, as created by the `getgrent`, `getgrgid`, and `getgrnam` subroutines.

The `putgrent` subroutine writes a line on the stream specified by the `fp` parameter. The stream matches the format of `/etc/group`.

The `gr_passwd` field of the line written is always set to `!` (exclamation point).

**Parameters**

- `grp`  
  Pointer to a group structure.

- `fp`  
  Specifies the stream to be written to.

**Return Values**
The `putgrent` subroutine returns a value of 0 upon successful completion. If `putgrent` fails, a nonzero value is returned.

**Files**

- `/etc/group`

- `/etc/security/group`

**Related Information**
The "getgrent, getgrgid, getgrnam, setgrent, or endgrent Subroutine" on page 366.
putgroupattrs Subroutine

Purpose
Stores multiple group attributes in the group database.

Library
Security Library (libc.a)

Syntax
```c
#include <usersec.h>

int putgroupattrs (Group, Attributes, Count)
char * Group;
dbattr_t * Attributes;
int Count;
```

Description
The `putgroupattrs` subroutine writes multiple group attributes into the group database. If the database is not already open, this subroutine does an implicit open for reading and writing. Data changed by `putgroupattrs` must be explicitly committed by calling the `putgroupattr` subroutine with a `Type` parameter specifying the `SEC_COMMIT` value. Until the data is committed, only `get` subroutine calls within the process return the written data.

The `Attributes` array contains information about each attribute that is to be written. Each element in the `Attributes` array must be examined upon a successful call to `putgroupattrs` to determine if the `Attributes` array entry was successfully put. The `dbattr_t` data structure contains the following fields:

- `attr_name`
  The name of the desired attribute.

- `attr_idx`
  Used internally by the `putgroupattrs` subroutine.

- `attr_type`
  The type of the desired attribute. The list of attribute types is defined in the `usersec.h` header file.

- `attr_flag`
  The results of the request to write the desired attribute.

- `attr_un`
  A union containing the values to be written. Its union members that follow correspond to the definitions of the `attr_char`, `attr_int`, `attr_long`, and `attr_llong` macros, respectively:
    - `un_char`
      Attributes of type `SEC_CHAR` and `SEC_LIST` store a pointer to the value to be written.
    - `un_int`
      Attributes of type `SEC_INT` and `SEC_BOOL` contain the value of the attribute to be written.
    - `un_long`
      Attributes of type `SEC_LONG` contain the value of the attribute to be written.
    - `un_llong`
      Attributes of type `SEC_LLONG` contain the value of the attribute to be written.
attr_domain

The authentication domain containing the attribute. The putgroupattrs subroutine stores the name of the authentication domain that was used to write this attribute if it is not initialized by the caller. The putgroupattrs subroutine is responsible for managing the memory referenced by this pointer. If attr_domain is specified for an attribute, the put request is sent only to that domain. If attr_domain is not specified (that is, set to NULL), putgroupattrs attempts to put the attributes to the first domain associated with the user. All put requests for the attributes with a NULL attr_domain are sent to the same domain. In other words, values cannot be put into different domains where attr_domain is unspecified; attr_domain is set to the name of the domain where the value is put and returned to the invoker.

When attr_domain is not specified, the list of searchable domains can be restricted to a particular domain by using the setauthdb function call.

Use the setuserdb and enduserdb subroutines to open and close the group database. Failure to explicitly open and close the group database can result in loss of memory and performance.

Parameters

Group

Specifies the name of the group for which the attributes are to be written.

Attributes

A pointer to an array of one or more elements of type dbattr_t. The list of group attributes is defined in the usersec.h header file.

Count

The number of array elements in Attributes.

Security

Files accessed:

<table>
<thead>
<tr>
<th>Mode</th>
<th>File</th>
</tr>
</thead>
<tbody>
<tr>
<td>rw</td>
<td>/etc/group</td>
</tr>
<tr>
<td>rw</td>
<td>/etc/security/group</td>
</tr>
<tr>
<td>rw</td>
<td>/etc/security/smitacl.group</td>
</tr>
</tbody>
</table>

Return Values

The putgroupattrs subroutine returns a value of 0 if the Group exists, even in the case when no attributes in the Attributes array were successfully updated. Otherwise, a value of -1 is returned and the errno global variable is set to indicate the error.

Error Codes

The putgroupattrs subroutine fails if one or more of the following are true:

EACCESS

The system information database could not be accessed for writing.

EINVAL

The Group parameter is the NULL pointer.

EINVAL

The Attributes parameter does not point to valid data for the requested attribute. Limited testing is possible and all errors might not be detected.

EINVAL

The Count parameter is less than or equal to 0.

ENOENT

The specified Group does not exist.

If the putgroupattrs subroutine fails to write an attribute, one or more of the following errors is returned in the attr_flag field of the corresponding Attributes element:

EACCESS

The user does not have access to the attribute specified in the attr_name field.

EINVAL

The attr_type field in the Attributes entry contains an invalid type.

EINVAL

The attr_un field in the Attributes entry does not point to a valid buffer or to valid data for this type of attribute. Limited testing is possible and all errors might not be detected.
The attr_name field in the Attributes entry specifies an attribute that is not defined for this group.

Examples
The following sample test program displays the output to a call to putgroupattrs. In this example, the system has a user named foo and a group named bar.

```c
#include <stdio.h>
#include <strings.h>
#include <string.h>
#include <usersec.h>

char * CommaToNSL(char *);

#define NATTR 2    /* Number of attributes to be put. */
#define GROUPNAME "bar"    /* Group name. */
#define DOMAIN "files"    /* Domain where attributes are going to put. */

main(int argc, char *argv[]) {
    int rc;
    int i;
    dbattr_t attributes[NATTR];
    /* Open the group database */
    setuserdb(S_WRITE);
    /* Valid put */
    attributes[0].attr_name = S_ADMIN;
    attributes[0].attr_type = SEC_BOOL;
    attributes[0].attr_domain = DOMAIN;
    attributes[0].attr_char = strdup("false");
    /* Valid put */
    attributes[1].attr_name = S_USERS;
    attributes[1].attr_type = SEC_LIST;
    attributes[1].attr_domain = DOMAIN;
    attributes[1].attr_char = CommaToNSL("foo");

    rc = putgroupattrs(GROUPNAME, attributes, NATTR);
    if (rc) {
        printf("putgroupattrs failed \n");
        goto clean_exit;
    }
    for (i = 0; i < NATTR; i++) {
        if (attributes[i].attr_flag)
            printf("Put failed for attribute %s. errno = %d \n",
                   attributes[i].attr_name, attributes[i].attr_flag);
        else
            printf("Put succeeded for attribute %s \n",
                   attributes[i].attr_name);
    }
    clean_exit:
    enduserdb();
    if (attributes[0].attr_char)
        free(attributes[0].attr_char);
    if (attributes[1].attr_char)
```
free(attributes[1].attr_char);
exit(rc);
}

/*
* Returns a new NSL created from a comma separated list.
* The comma separated list is unmodified.
* */
char *
CommaToNSL(char *CommaList)
{
    char  *NSL = (char *) NULL;
    char  *s;
    if (!CommaList)
        return(NSL);
    if (!NSL = (char *) malloc(strlen(CommaList) + 2))
        return(NSL);
    strcpy(NSL, CommaList);
    for (s = NSL; *s; s++)
        if (*s == ',')
            *s = '\0';
    *(++s) = '\0';
}

The following output for the call is expected:
Put succeeded for attribute admin
Put succeeded for attribute users

Related Information
The setuserdb Subroutine

List of Security and Auditing Subroutines [Subroutines Overview] in AIX 5L Version 5.3 General Programming Concepts: Writing and Debugging Programs.

puts or fputs Subroutine

Purpose
Writes a string to a stream.

Library
Standard I/O Library (libc.a)

Syntax
#include <stdio.h>

int puts (const char *String);
int fputs (const char *String, FILE *Stream);
Description
The **puts** subroutine writes the string pointed to by the **String** parameter to the standard output stream, `stdout`, and appends a new-line character to the output.

The **fputs** subroutine writes the null-terminated string pointed to by the **String** parameter to the output stream specified by the **Stream** parameter. The **fputs** subroutine does not append a new-line character.

Neither subroutine writes the terminating null character.

After the **fputwc**, **putwc**, **fputc**, **fputs**, **puts**, or **putw** subroutine runs successfully, and before the next successful completion of a call either to the **flush** or **fclose** subroutine on the same stream or a call to the **exit** or **abort** subroutine, the **st_ctime** and **st_mtime** fields of the file are marked for update.

Parameters

**String** Points to a string to be written to output.

**Stream** Points to the **FILE** structure of an open file.

Return Values
Upon successful completion, the **puts** and **fputs** subroutines return the number of characters written. Otherwise, both subroutines return **EOF**, set an error indicator for the stream and set the **errno** global variable to indicate the error. This happens if the routines try to write to a file that has not been opened for writing.

Error Codes
If the **puts** or **fputs** subroutine is unsuccessful because the output stream specified by the **Stream** parameter is unbuffered or the buffer needs to be flushed, it returns one or more of the following error codes:

- **EAGAIN** Indicates that the **O_NONBLOCK** flag is set for the file descriptor specified by the **Stream** parameter and the process would be delayed in the write operation.
- **EBADF** Indicates that the file descriptor specified by the **Stream** parameter is not a valid file descriptor open for writing.
- **EFILE** Indicates that an attempt was made to write to a file that exceeds the process’ file size limit or the systemwide maximum file size.
- **EINTR** Indicates that the write operation was terminated due to receipt of a signal and no data was transferred. **Note:** Depending upon which library routine the application binds to, this subroutine may return **EINTR**. Refer to the [signal subroutine](#) regarding the **SA_RESTART** bit.
- **EIO** Indicates that the process is a member of a background process group attempting to perform a write to its controlling terminal, the **TOSTOP** flag is set, the process is neither ignoring or blocking the **SIGTTOU** signal, and the process group of the process has no parent process.
- **ENOSPC** Indicates that there was no free space remaining on the device containing the file specified by the **Stream** parameter.
- **EPipe** Indicates that an attempt is made to write to a pipe or first-in-first-out (FIFO) that is not open for reading by any process. A **SIGPIPE** signal will also be sent to the process.
- **ENOMEM** Indicates that insufficient storage space is available.
- **ENXIO** Indicates that a request was made of a nonexistent device, or the request was outside the capabilities of the device.

Related Information
The **fopen**, **freopen**, or **fdopen** subroutine, **fread**, or **fwrite** subroutine, **fgets**, or **fgetws** subroutine, **getw**, or **getws** subroutine, **getwc**, or **getwchar** subroutine, **fopen64**, **freopen64**, or **fdopen64** subroutine, **fread64**, **fwrite64**, **fgets64**, or **fgetws64** subroutine, **getw64**, or **getline** subroutine, **getline64** subroutine, **fopen64**, **freopen64**, or **fdopen64** subroutine, **fread**, or **fwrite** subroutine, **fgets**, or **fgetws** subroutine, **getw**, or **getws** subroutine, **getwc**, or **getwchar** subroutine, **fprintf**, or **fputs** subroutine,
subroutine, *printf, fprintf, and sprintf* subroutine, *putc, putchar, fputc, or putw* subroutine, *putwc, putwchar, or fputwc* subroutine, *putws or fputws* subroutine.

The *feof, ferror, clearerr*, or *fileno* macros.

List of String Manipulation Services

Subroutines Overview in AIX 5L Version 5.3 General Programming Concepts: Writing and Debugging Programs.

**putuserattrs Subroutine**

**Purpose**
Stores multiple user attributes in the user database.

**Library**
Security Library (*libc.a*)

**Syntax**

```c
#include <usersec.h>

int putuserattrs (User, Attributes, Count)
char * User;
dbattr_t * Attributes;
int Count;
```

**Description**

The *putuserattrs* subroutine writes multiple user attributes into the user database. If the database is not already open, this subroutine does an implicit open for reading and writing. Data changed by *putuserattrs* must be explicitly committed by calling the *putuserattr* subroutine with a *Type* parameter specifying the *SEC_COMMIT* value. Until the data is committed, only *get* subroutine calls within the process return the written data.

The *Attributes* array contains information about each attribute that is to be written. Each element in the *Attributes* array must be examined upon a successful call to *putuserattrs* to determine if the *Attributes* array entry was successfully put. Please see *putuserattr* man page for the supported attributes. The *dbattr_t* data structure contains the following fields:

- **attr_name**
  The name of the desired attribute.

- **attr_idx**
  Used internally by the *putuserattrs* subroutine.

- **attr_type**
  The type of the desired attribute. The list of attribute types is defined in the *usersec.h* header file.

- **attr_flag**
  The results of the request to write the desired attribute.

- **attr_un**
  A union containing the returned values. Its union members that follow correspond to the definitions of the *attr_char, attr_int, attr_long,* and *attr_llong* macros, respectively:
Attributes of type `SEC_CHAR` and `SEC_LIST` contain a pointer to the value to be written.

Attributes of type `SEC_INT` and `SEC_BOOL` contain the value of the attribute to be written.

Attributes of type `SEC_LONG` contain the value of the attribute to be written.

Attributes of type `SEC_LLONG` contain the value of the attribute to be written.

The authentication domain containing the attribute. The `putuserattrs` subroutine stores the name of the authentication domain that was used to write this attribute if it is not initialized by the caller. The `putuserattrs` subroutine is responsible for managing the memory referenced by this pointer. If `attr_domain` is specified for an attribute, the put request is sent only to that domain. If `attr_domain` is not specified (that is, set to NULL), `putuserattrs` attempts to put the attributes to the first domain associated with the user. All put requests for the attributes with a NULL `attr_domain` are sent to the same domain. In other words, values cannot be put into different domains where `attr_domain` is unspecified; `attr_domain` is set to the name of the domain where the value is put and returned to the invoker.

When `attr_domain` is not specified, the list of searchable domains can be restricted to a particular domain by using the `setauthdb` function call.

Use the `setuserdb` and `enduserdb` subroutines to open and close the user database. Failure to explicitly open and close the user database can result in loss of memory and performance.

### Parameters

**User**
Specifies the name of the user for which the attributes are to be written.

**Attributes**
A pointer to an array of one or more elements of type `dbattr_t`. The list of user attributes is defined in the `usersec.h` header file.

**Count**
The number of array elements in `Attributes`.

### Security

Files accessed:

<table>
<thead>
<tr>
<th>Mode</th>
<th>File</th>
</tr>
</thead>
<tbody>
<tr>
<td>rw</td>
<td>/etc/group</td>
</tr>
<tr>
<td>rw</td>
<td>/etc/passwd</td>
</tr>
<tr>
<td>rw</td>
<td>/etc/security/audit/config</td>
</tr>
<tr>
<td>rw</td>
<td>/etc/security/environ</td>
</tr>
<tr>
<td>rw</td>
<td>/etc/security/group</td>
</tr>
<tr>
<td>rw</td>
<td>/etc/security/lastlog</td>
</tr>
<tr>
<td>rw</td>
<td>/etc/security/limits</td>
</tr>
<tr>
<td>rw</td>
<td>/etc/security/passwd</td>
</tr>
<tr>
<td>rw</td>
<td>/etc/security/pwdhist.dir</td>
</tr>
<tr>
<td>rw</td>
<td>/etc/security/pwdhist.pag</td>
</tr>
<tr>
<td>rw</td>
<td>/etc/security/smitacl.user</td>
</tr>
<tr>
<td>rw</td>
<td>/etc/security/user.roles</td>
</tr>
</tbody>
</table>

### Return Values

The `putuserattrs` subroutine returns a value of 0 if the `User` exists, even in the case when no attributes in the `Attributes` array were successfully updated. Otherwise, a value of -1 is returned and the `errno` global variable is set to indicate the error.
Error Codes

The *putuserattrs* subroutine fails if one or more of the following is true:

- **EACCESS**
  The system information database could not be accessed for writing.

- **EINVAL**
  The *User* parameter is NULL.

- **EINVAL**
  The *Attributes* parameter does not point to valid data for the requested attribute. Limited testing is possible and all errors might not be detected.

- **EINVAL**
  The *Attributes* parameter does not point to valid data for the requested attribute. Limited testing is possible and all errors might not be detected.

- **ENOENT**
  The specified *User* parameter does not exist.

If the *putuserattrs* subroutine fails to write an attribute, one or more of the following errors is returned in the *attr_flag* field of the corresponding *Attributes* element:

- **EACCESS**
  The user does not have access to the attribute specified in the *attr_name* field.

- **EINVAL**
  The *attr_type* field in the *Attributes* entry contains an invalid type.

- **EINVAL**
  The *attr_un* field in the *Attributes* entry does not point to a valid buffer or to valid data for this type of attribute. Limited testing is possible and all errors might not be detected.

- **ENOATTR**
  The *attr_name* field in the *Attributes* entry specifies an attribute that is not defined for this user.

Examples

The following sample test program displays the output to a call to *putuserattrs*. In this example, the system has a user named foo.

```c
#include <stdio.h>
#include <strings.h>
#include <string.h>
#include <usersec.h>

char *CommaToNSL(char *);

#define NATTR 4    /* Number of attributes to be put */
#define USERNAME "foo" /* User name */
#define DOMAIN "files" /* domain where attributes are going to put. */

main(int argc, char *argv[]) {
    int rc;
    int i;
    dbattr_t attributes[NATTR];

    /* Open the user database */
    setuserdb(S_WRITE);

    /* Valid put */
    attributes[0].attr_name = S_GECOS;
    attributes[0].attr_type = SEC_CHAR;
    attributes[0].attr_domain = DOMAIN;
    attributes[0].attr_char = strdup("I am foo");

    /* Invalid put */
    attributes[1].attr_name = S_LOGINCHK;
    attributes[1].attr_type = SEC_BOOL;
    attributes[1].attr_domain = DOMAIN;
    attributes[1].attr_char = strdup("allow");

    /* Valid put */
```

Base Operating System (BOS) Runtime Services (A-P) 1313
attributes[2].attr_name = S_MAXAGE;
attributes[2].attr_type  = SEC_INT;
attributes[2].attr_domain = DOMAIN;
attributes[2].attr_int  = 10;

/* Valid put */
attributes[3].attr_name = S_GROUPS;
attributes[3].attr_type = SEC_LIST;
attributes[3].attr_domain = DOMAIN;
attributes[3].attr_char  = CommaToNSL("staff,system");

crc = putuserattrs(USERNAME, attributes, NATTR);

if (rc) {
    printf("putuserattrs failed \n");
    goto clean_exit;
}

for (i = 0; i < NATTR; i++) {
    if (attributes[i].attr_flag)
        printf("Put failed for attribute %s. errno = %d \n", attributes[i].attr_name, attributes[i].attr_flag);
    else
        printf("Put succeded for attribute %s \n", attributes[i].attr_name);
}

clean_exit:
    enduserdb();

    if (attributes[0].attr_char)
        free(attributes[0].attr_char);

        if (attributes[1].attr_char)
            free(attributes[1].attr_char);

        if (attributes[3].attr_char)
            free(attributes[3].attr_char);

    exit(rc);

/*
 * Returns a new NSL created from a comma separated list.
 * The comma separated list is unmodified.
 * */
char * CommaToNSL(char * CommaList)
{
    char  *NSL = (char *) NULL;
    char  *s;

    if (!CommaList)
        return(NSL);

    if (!(*NSL = (char *) malloc(strlen(CommaList) + 2)))
        return(NSL);

    strcpy(NSL, CommaList);

    for (s = NSL; *s; s++)
        if (*s == ',')
The following output for the call is expected:

Put succeeded for attribute gecos
Put failed for attribute login (errno = 22)
Put succeeded for attribute maxage
Put succeeded for attribute groups

Related Information
The `setuserdb` Subroutine

List of Security and Auditing Subroutines | Subroutines Overview | in AIX 5L Version 5.3 General Programming Concepts: Writing and Debugging Programs.

**putuserpwx Subroutine**

**Purpose**
Accesses the user authentication data.

**Library**
Security Library (`libc.a`)

**Syntax**
```
#include <userpw.h>

int putuserpwx (Password)
struct userpw *Password;
```

**Description**
The `putuserpwx` subroutine modifies user authentication information. It can be used with those administrative domains that support modifying the user’s encrypted password with the `putuserattrs` subroutine. The `chpassx` subroutine must be used to modify authentication information for administrative domains that do not support that functionality.

The `putuserpwx` subroutine updates or creates password authentication data for the user defined in the `Password` parameter in the administrative domain that is specified. The password entry created by the `putuserpwx` subroutine is used only if there is an `!` (exclamation point) in the user’s password (`S_PWD`) attribute. The user application can use the `putuserattrs` subroutine to add an `!` to this field.

The `putuserpwx` subroutine opens the authentication database read-write if no other access has taken place, but the program should call `setpwd` (`S_READ | S_WRITE`) before calling the `putuserpwx` subroutine and `endpwd` when access to the authentication information is no longer required.

The administrative domain specified in the `upw_authdb` field is set by the `putuserpwx` subroutine. It must be specified by the application program if the `putuserpwx` subroutine is not used to produce the `Password` parameter.
Parameters

Password

Specifies the password structure used to update the password information for this user. The fields in a userpw structure are defined in the userpw.h file and contains the following members:

upw_name
Specifies the user’s name.

upw_passwd
Specifies the user’s encrypted password.

upw_lastupdate
Specifies the time, in seconds, since the epoch (that is, 00:00:00 GMT, 1 January 1970), when the password was last updated.

upw_flags
Specifies attributes of the password. This member is a bit mask of one or more of the following values, defined in the userpw.h file:

PW_NOCHECK
Specifies that new passwords need not meet password restrictions in effect for the system.

PW_ADMCHG
Specifies that the password was last set by an administrator and must be changed at the next successful use of the login or su command.

PW_ADMIN
Specifies that password information for this user can only be changed by the root user.

upw_authdb
Specifies the administrative domain containing the authentication data.

Security

Files accessed:

<table>
<thead>
<tr>
<th>Mode</th>
<th>File</th>
</tr>
</thead>
<tbody>
<tr>
<td>rw</td>
<td>/etc/security/passwd</td>
</tr>
</tbody>
</table>

Return Values

If successful, the putuserpwx subroutine returns a value of 0. If the subroutine failed to update or create the password information, the putuserpwx subroutine returns a nonzero value.

Error Codes

The getuserpwx subroutine fails if the following value is true:

ENOENT
The user does not have an entry in the /etc/security/passwd file.

Subroutines invoked by the putuserpwx subroutine can also set errors.

Files

/etc/security/passwd Contains user passwords.
Related Information
The getuserattr, IDtouser, nextuser, or putuserattr Subroutines on page 449, putgroupattrs Subroutine on page 1306, putuserattrs Subroutine on page 1311, setpwdb Subroutine, setuserdb Subroutine.

putwc, putwchar, or fputwc Subroutine

Purpose
Writes a character or a word to a stream.

Library
Standard I/O Library (libc.a)

Syntax
#include <stdio.h>

wint_t putwc(Character, Stream)

FILE *Stream;

wint_t putwchar(Character)

wint_t fputwc(Character, Stream)

FILE Stream;

Description
The putwc subroutine writes the wide character specified by the Character parameter to the output stream pointed to by the Stream parameter. The wide character is written as a multibyte character at the associated file position indicator for the stream, if defined. The subroutine then advances the indicator. If the file cannot support positioning requests, or if the stream was opened with append mode, the character is appended to the output stream.

The putwchar subroutine works like the putwc subroutine, except that putwchar writes the specified wide character to the standard output.

The fputwc subroutine works the same as the putwc subroutine.

Output streams, with the exception of stderr, are buffered by default if they refer to files, or line-buffered if they refer to terminals. The standard error output stream, stderr, is unbuffered by default, but using the freopen subroutine causes it to become buffered or line-buffered. Use the setbuf subroutine to change the stream’s buffering strategy.

After the fputwc, putwc, fputc, putc, fputs, puts, or putw subroutine runs successfully, and before the next successful completion of a call either to the fflush or fclose subroutine on the same stream or to the exit or abort subroutine, the st_ctime and st_mtime fields of the file are marked for update.

Parameters

Character Specifies a wide character of type wint_t.
Stream Specifies a stream of output data.
Return Values
Upon successful completion, the putwc, putwchar, and fputwc subroutines return the wide character that is written. Otherwise WEOF is returned, the error indicator for the stream is set, and the errno global variable is set to indicate the error.

Error Codes
If the putwc, putwchar, or fputwc subroutine fails because the stream is not buffered or data in the buffer needs to be written, it returns one or more of the following error codes:

EAGAIN Indicates that the O_NONBLOCK flag is set for the file descriptor underlying the Stream parameter, delaying the process during the write operation.

EBADF Indicates that the file descriptor underlying the Stream parameter is not valid and cannot be updated during the write operation.

EFBIG Indicates that the process attempted to write to a file that already equals or exceeds the file-size limit for the process. The file is a regular file and an attempt was made to write at or beyond the offset maximum associated with the corresponding stream.

EILSEQ Indicates that the wide-character code does not correspond to a valid character.

EINTR Indicates that the process has received a signal that terminates the read operation.

EIO Indicates that the process is in a background process group attempting to perform a write operation to its controlling terminal. The TOSTOP flag is set, the process is not ignoring or blocking the SIGTTOU flag, and the process group of the process is orphaned.

ENOMEM Insufficient storage space is available.

ENOSPC Indicates that no free space remains on the device containing the file.

ENXIO Indicates a request was made of a non-existent device, or the request was outside the capabilities of the device.

EPIPE Indicates that the process has attempted to write to a pipe or first-in-first-out (FIFO) that is not open for reading. The process will also receive a SIGPIPE signal.

Related Information
Other wide character I/O subroutines: fgetwc ("getwc, fgetwc, or getwchar Subroutine" on page 472) subroutine, fgetws ("getws or fgetws Subroutine" on page 475) subroutine, fputwc ("putwc or fputwc Subroutine" on page 472) subroutine, getwc ("getwc, fgetwc, or getwchar Subroutine" on page 472) subroutine, fputws ("putws or fputws Subroutine" on page 475) subroutine, getwchar ("getwc, fgetwc, or getwchar Subroutine" on page 472) subroutine, getws ("getws or fgetws Subroutine" on page 475) subroutine, putwc ("putwc or fputwc Subroutine" on page 1319) subroutine, ungetwc subroutine.

Related standard I/O subroutines: fdopen ("fopen, fopen64, freopen, freopen64 or fdopen Subroutine" on page 284) subroutine, fgets ("gets or fgets Subroutine" on page 429) subroutine, fopen ("fopen, fopen64, freopen, freopen64 or fdopen Subroutine" on page 284) subroutine, fprintf ("printf, fprintf, sprintf, snprintf, wprintf, vprintf, vsprintf, or vswprintf Subroutine" on page 1148) subroutine, putc ("putc, putchar, fputc, or putw Subroutine" on page 1309) subroutine, read ("fread or fwrite Subroutine" on page 307) subroutine, freopen ("fopen, fopen64, freopen, freopen64 or fdopen Subroutine" on page 284) subroutine, fwrite ("fread or fwrite Subroutine" on page 307) subroutine, gets ("gets or fgets Subroutine" on page 429) subroutine, printf ("printf, fprintf, sprintf, snprintf, wprintf, vprintf, vsprintf, or vswprintf Subroutine" on page 1148) subroutine, putwc ("putc, putchar, fputc, or putw Subroutine" on page 1309) subroutine, putc ("putc, putchar, fputc, or putw Subroutine" on page 1309) subroutine, putws ("putwc or fputwc Subroutine" on page 1309) subroutine, sprintf ("printf, fprintf, sprintf, snprintf, wprintf, vprintf, vsprintf, or vswprintf Subroutine" on page 1148) subroutine.

Subroutines, Example Programs, and Libraries in AIX 5L Version 5.3 General Programming Concepts: Writing and Debugging Programs.

putws or fputws Subroutine

Purpose
Writes a wide-character string to a stream.

Library
Standard I/O Library (libc.a)

Syntax
#include <stdio.h>

int putws (const wchar_t *String);

int fputws (const wchar_t *String, FILE *Stream);

Description
The putws subroutine writes the const wchar_t string pointed to by the String parameter to the standard output stream (stdout) as a multibyte character string and appends a new-line character to the output. In all other respects, the putws subroutine functions like the puts subroutine.

The fputws subroutine writes the const wchar_t string pointed to by the String parameter to the output stream as a multibyte character string. In all other respects, the fputws subroutine functions like the fputs subroutine.

After the putws or fputws subroutine runs successfully, and before the next successful completion of a call to the fflush or fclose subroutine on the same stream or a call to the exit or abort subroutine, the st_ctime and st_mtime fields of the file are marked for update.

Parameters
String Points to a string to be written to output.
Stream Points to the FILE structure of an open file.

Return Values
Upon successful completion, the putws and fputws subroutines return a nonnegative number. Otherwise, a value of -1 is returned, and the errno global variable is set to indicate the error.

Error Codes
The putws or fputws subroutine is unsuccessful if the stream is not buffered or data in the buffer needs to be written, and one of the following errors occur:

EAGAIN The O_NONBLOCK flag is set for the file descriptor underlying the Stream parameter, which delays the process during the write operation.
EBADF The file descriptor underlying the Stream parameter is not valid and cannot be updated during the write operation.
EFBIG The process attempted to write to a file that already equals or exceeds the file-size limit for the process.
EINTR The process has received a signal that terminates the read operation.
EIO   The process is in a background process group attempting to perform a write operation to its controlling
terminal. The TOSTOP flag is set, the process is not ignoring or blocking the SIGTTOU flag, and the
process group of the process is orphaned.
ENOSPC  No free space remains on the device containing the file.
EPIPE  The process has attempted to write to a pipe or first-in-first-out (FIFO) that is not open for reading. The
process also receives a SIGPIPE signal.
EILSEQ  The wc wide-character code does not correspond to a valid character.

Related Information
Other wide-character I/O subroutines: “getwc, fgetwc, or getwchar Subroutine” on page 472, “getws or
fgetws Subroutine” on page 475, “putwc, putwchar, or fprintf Subroutine” on page 1317, and “ungetwc
subroutine.
Related standard I/O subroutines: “fopen, fopen64, freopen, freopen64 or fdopen Subroutine” on page 284,
“gets or fgets Subroutine” on page 429, “printf, fprintf, sprintf, snprintf, vsprintf, vprintf, vfprintf, or
vswprintf Subroutine” on page 1148, “puts or fputs Subroutine” on page 1309, “fread or fwrite Subroutine” on page 307.

Subroutines, Example Programs, and Libraries in AIX 5L Version 5.3 General Programming Concepts:
Writing and Debugging Programs.
National Language Support Overview and Multibyte Code and Wide Character Code Conversion

pwdrestrict_method Subroutine

Purpose  Defines loadable password restriction methods.

Library

Syntax
int pwdrestrict_method (UserName, newPassword, OldPassword, Message)

char * UserName;
char * NewPassword;
char * OldPassword;
char ** Message;

Description
The pwdrestrict_method subroutine extends the capability of the password restrictions software and lets
an administrator enforce password restrictions that are not provided by the system software.

Whenever users change their passwords, the system software scans the pwdchecks attribute defined for
that user for site specific restrictions. Since this attribute field can contain load module file names, for
example, methods, it is possible for the administrator to write and install code that enforces site specific
password restrictions.

The system evaluates the pwdchecks attribute’s value field in a left to right order. For each method that
the system encounters, the system loads and invokes that method. The system uses the load subroutine
to load methods. It invokes the load subroutine with a Flags value of 1 and a LibraryPath value of /usr/lib.
Once the method is loaded, the system invokes the method.
To create a loadable module, use the `-e` flag of the `ld` command. Note that the name `pwdrestrict_method` given in the syntax is a generic name. The actual subroutine name can be anything (within the compiler’s name space) except `main`. What is important is, that for whatever name you choose, you must inform the `ld` command of the name so that the `load` subroutine uses that name as the entry point into the module. In the following example, the C compiler compiles the `pwdrestrict.c` file and pass `-e pwdrestrict_method` to the `ld` command to create the method called `pwdrestrict`:

```
cc -e pwdrestrict_method -o pwdrestrict pwdrestrict.c
```

The convention of all password restriction methods is to pass back messages to the invoking subroutine. Do not print messages to stdout or stderr. This feature allows the password restrictions software to work across network connections where stdout and stderr are not valid. Note that messages must be returned in dynamically allocated memory to the invoking program. The invoking program will deallocate the memory once it is done with the memory.

There are many caveats that go along with loadable subroutine modules:

1. The values for `NewPassword` and `OldPassword` are the actual clear text passwords typed in by the user. If you copy these passwords into other parts of memory, clear those memory locations before returning back to the invoking program. This helps to prevent clear text passwords from showing up in core dumps. Also, do not copy these passwords into a file or anywhere else that another program can access. Clear text passwords should never exist outside of the process space.
2. Do not modify the current settings of the process’ signal handlers.
3. Do not call any functions that will terminate the execution of the program (for example, the `exit` subroutine, the `exec` subroutine). Always return to the invoking program.
4. The code must be thread-safe.
5. The actual load module must be kept in a write protected environment. The load module and directory should be writable only by the root user.

One last note, all standard password restrictions are performed before any of the site specific methods are invoked. Thus, methods are the last restrictions to be enforced by the system.

**Parameters**

- **UserName** Specifies a "local" user name.
- **NewPassword** Specifies the new password in clear text (not encrypted). This value may be a NULL pointer. Clear text passwords are always in 7 bit ASCII.
- **OldPassword** Specifies the current password in clear text (not encrypted). This value may be a NULL pointer. Clear text passwords are always in 7 bit ASCII.
- **Message** Specifies the address of a pointer to `malloc`'ed memory containing an NLS error message. The method is expected to supply the `malloc`'ed memory and the message.

**Return Values**

The method is expected to return the following values. The return values are listed in order of precedence.

-1 Internal error. The method could not perform its password evaluation. The method must set the `errno` variable. The method must supply an error message in `Message` unless it can’t allocate memory for the message. If it cannot allocate memory, then it must return the NULL pointer in `Message`.
1 Failure. The password change did not meet the requirements of the restriction. The password restriction was properly evaluated and the password change was not accepted. The method must supply an error message in `Message`. The `errno` variable is ignored. Note that composition failures are cumulative, thus, even though a failure condition is returned, trailing composition methods will be invoked.
0 Success. The password change met the requirements of the restriction. If necessary, the method may supply a message in `Message`; otherwise, return the NULL pointer. The `errno` variable is ignored.
Appendix A. Base Operating System Error Codes for Services That Require Path-Name Resolution

The following errors apply to any service that requires path name resolution:

**EACCES** Search permission is denied on a component of the path prefix.

**EFAULT** The *Path* parameter points outside of the allocated address space of the process.

**EIO** An I/O error occurred during the operation.

**ELOOP** Too many symbolic links were encountered in translating the *Path* parameter.

**ENAMETOOLONG** A component of a path name exceeded 255 characters and the process has the *DisallowTruncation* attribute (see the *ulimit* subroutine) or an entire path name exceeded 1023 characters.

**ENOENT** A component of the path prefix does not exist.

**ENOENT** A symbolic link was named, but the file to which it refers does not exist.

**ENOENT** The path name is null.

**ENOTDIR** A component of the path prefix is not a directory.

**ESTALE** The root or current directory of the process is located in a virtual file system that is unmounted.

Related Information

[List of File and Directory Manipulation Services]
### Appendix B. ODM Error Codes

When an ODM subroutine is unsuccessful, a value of -1 is returned and the `odmerrno` variable is set to one of the following values:

<table>
<thead>
<tr>
<th>Error Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ODMI_BAD_CLASSNAME</td>
<td>The specified object class name does not match the object class name in the file. Check path name and permissions.</td>
</tr>
<tr>
<td>ODMI_BAD_CLXNNAME</td>
<td>The specified collection name does not match the collection name in the file.</td>
</tr>
<tr>
<td>ODMI_BAD_CRIT</td>
<td>The specified search criteria is incorrectly formed. Make sure the criteria contains only valid descriptor names and the search values are correct. For information on qualifying criteria, see &quot;Understanding ODM Object Searches&quot; in AIX 5L Version 5.3 General Programming Concepts: Writing and Debugging Programs.</td>
</tr>
<tr>
<td>ODMI_BAD_LOCK</td>
<td>Cannot set a lock on the file. Check path name and permissions.</td>
</tr>
<tr>
<td>ODMI_BAD_TIMEOUT</td>
<td>The time-out value was not valid. It must be a positive integer.</td>
</tr>
<tr>
<td>ODMI_BAD_TOKEN</td>
<td>Cannot create or open the lock file. Check path name and permissions.</td>
</tr>
<tr>
<td>ODMI_CLASS_DNE</td>
<td>The specified object class does not exist. Check path name and permissions.</td>
</tr>
<tr>
<td>ODMI_CLASS_EXISTS</td>
<td>The specified object class already exists. An object class must not exist when it is created.</td>
</tr>
<tr>
<td>ODMI_CLASS_PERMS</td>
<td>The object class cannot be opened because of the file permissions.</td>
</tr>
<tr>
<td>ODMI_CLXNMAGICNO_ERR</td>
<td>The specified collection is not a valid object class collection.</td>
</tr>
<tr>
<td>ODMI_FORK</td>
<td>Cannot fork the child process. Make sure the child process is executable and try again.</td>
</tr>
<tr>
<td>ODMI_INTERNAL_ERR</td>
<td>An internal consistency problem occurred. Make sure the object class is valid or contact the person responsible for the system.</td>
</tr>
<tr>
<td>ODMI_INVALID_CLASS</td>
<td>The specified file is not an object class.</td>
</tr>
<tr>
<td>ODMI_INVALID_CLXN</td>
<td>Either the specified collection is not a valid object class collection or the collection does not contain consistent data.</td>
</tr>
<tr>
<td>ODMI_INVALID_PATH</td>
<td>The specified path does not exist on the file system. Make sure the path is accessible.</td>
</tr>
<tr>
<td>ODMI_LINK_NOT_FOUND</td>
<td>The object class that is accessed could not be opened. Make sure the linked object class is accessible.</td>
</tr>
<tr>
<td>ODMI_LOCK_BLOCKED</td>
<td>Cannot grant the lock. Another process already has the lock.</td>
</tr>
<tr>
<td>ODMI_LOCK_ENV</td>
<td>Cannot retrieve or set the lock environment variable. Remove some environment variables and try again.</td>
</tr>
<tr>
<td>ODMI_LOCK_ID</td>
<td>The lock identifier does not refer to a valid lock. The lock identifier must be the same as what was returned from the <code>odm_lock</code> subroutine on page 913.</td>
</tr>
<tr>
<td>ODMI_MAGICNO_ERR</td>
<td>The class symbol does not identify a valid object class.</td>
</tr>
<tr>
<td>ODMI_MALLOC_ERR</td>
<td>Cannot allocate sufficient storage. Try again later or contact the person responsible for the system.</td>
</tr>
<tr>
<td>ODMI_NO_OBJECT</td>
<td>The specified object identifier did not refer to a valid object.</td>
</tr>
<tr>
<td>ODMI_OPEN_ERR</td>
<td>Cannot open the object class. Check path name and permissions.</td>
</tr>
<tr>
<td>ODMI_OPEN_PIPE</td>
<td>Cannot open a pipe to a child process. Make sure the child process is executable and try again.</td>
</tr>
<tr>
<td>ODMI_PARAMS</td>
<td>The parameters passed to the subroutine were not correct. Make sure there are the correct number of parameters and that they are valid.</td>
</tr>
<tr>
<td>ODMI_READ_ONLY</td>
<td>The specified object class is opened as read-only and cannot be modified.</td>
</tr>
<tr>
<td>ODMI_READ_PIPE</td>
<td>Cannot read from the pipe of the child process. Make sure the child process is executable and try again.</td>
</tr>
<tr>
<td>ODMI_TOOMANYCLASSES</td>
<td>Too many object classes have been accessed. An application can only access less than 1024 object classes.</td>
</tr>
<tr>
<td>ODMI_UNLINKCLASS_ERR</td>
<td>Cannot remove the object class from the file system. Check path name and permissions.</td>
</tr>
<tr>
<td>ODMI_UNLINKCLXN_ERR</td>
<td>Cannot remove the object class collection from the file system. Check path name and permissions.</td>
</tr>
</tbody>
</table>
Cannot unlock the lock file. Make sure the lock file exists.

Related Information

List of ODM Commands and Subroutines in AIX 5L Version 5.3 General Programming Concepts: Writing and Debugging Programs.
Appendix C. Notices

This information was developed for products and services offered in the U.S.A.

IBM® may not offer the products, services, or features discussed in this document in other countries. Consult your local IBM representative for information on the products and services currently available in your area. Any reference to an IBM product, program, or service is not intended to state or imply that only that IBM product, program, or service may be used. Any functionally equivalent product, program, or service that does not infringe any IBM intellectual property right may be used instead. However, it is the user’s responsibility to evaluate and verify the operation of any non-IBM product, program, or service.

IBM may have patents or pending patent applications covering subject matter described in this document. The furnishing of this document does not give you any license to these patents. You can send license inquiries, in writing, to:

IBM Director of Licensing
IBM Corporation
North Castle Drive
Armonk, NY 10504-1785
U.S.A.

The following paragraph does not apply to the United Kingdom or any other country where such provisions are inconsistent with local law: INTERNATIONAL BUSINESS MACHINES CORPORATION PROVIDES THIS PUBLICATION "AS IS" WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESS OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF NON-INFRINGEMENT, MERCHANTABILITY OR FITNESS FOR A PARTICULAR PURPOSE. Some states do not allow disclaimer of express or implied warranties in certain transactions, therefore, this statement may not apply to you.

This information could include technical inaccuracies or typographical errors. Changes are periodically made to the information herein; these changes will be incorporated in new editions of the publication. IBM may make improvements and/or changes in the product(s) and/or the program(s) described in this publication at any time without notice.

Licensees of this program who wish to have information about it for the purpose of enabling: (i) the exchange of information between independently created programs and other programs (including this one) and (ii) the mutual use of the information which has been exchanged, should contact:

IBM Corporation
Dept. LRAS/Bldg. 003
11400 Burnet Road
Austin, TX 78758-3498
U.S.A.

Such information may be available, subject to appropriate terms and conditions, including in some cases, payment of a fee.

The licensed program described in this document and all licensed material available for it are provided by IBM under terms of the IBM Customer Agreement, IBM International Program License Agreement or any equivalent agreement between us.

For license inquiries regarding double-byte (DBCS) information, contact the IBM Intellectual Property Department in your country or send inquiries, in writing, to:
IBM World Trade Asia Corporation
Licensing
2-31 Roppongi 3-chome, Minato-ku
Tokyo 106, Japan

IBM may use or distribute any of the information you supply in any way it believes appropriate without incurring any obligation to you.

Information concerning non-IBM products was obtained from the suppliers of those products, their published announcements or other publicly available sources. IBM has not tested those products and cannot confirm the accuracy of performance, compatibility or any other claims related to non-IBM products. Questions on the capabilities of non-IBM products should be addressed to the suppliers of those products.

Any references in this information to non-IBM Web sites are provided for convenience only and do not in any manner serve as an endorsement of those Web sites. The materials at those Web sites are not part of the materials for this IBM product and use of those Web sites is at your own risk.

This information contains examples of data and reports used in daily business operations. To illustrate them as completely as possible, the examples include the names of individuals, companies, brands, and products. All of these names are fictitious and any similarity to the names and addresses used by an actual business enterprise is entirely coincidental.

COPYRIGHT LICENSE:

This information contains sample application programs in source language, which illustrates programming techniques on various operating platforms. You may copy, modify, and distribute these sample programs in any form without payment to IBM, for the purposes of developing, using, marketing or distributing application programs conforming to the application programming interface for the operating platform for which the sample programs are written. These examples have not been thoroughly tested under all conditions. IBM, therefore, cannot guarantee or imply reliability, serviceability, or function of these programs. You may copy, modify, and distribute these sample programs in any form without payment to IBM for the purposes of developing, using, marketing, or distributing application programs conforming to IBM's application programming interfaces.

Trademarks

The following terms are trademarks of International Business Machines Corporation in the United States, other countries, or both:

AIX
AIX 5L
IBM
POWER4
POWER5
POWER6
PTX

UNIX is a registered trademark of The Open Group in the United States and other countries.

Linux is a trademark of Linus Torvalds in the United States, other countries, or both.

Other company, product, or service names may be trademarks or service marks of others.
Index

Special characters
_atojis macro 571
_check_lock Subroutine 127
_clear_lock Subroutine 128
_edata identifier 223
_end identifier 223
_exit subroutine 242
_Subroutine 242
_extext identifier 223
_jisto macro 571
_lazySetErrorHandler Subroutine 579
_tojlower macro 571
_tojupper macro 571
_tolower subroutine 183
_toupper subroutine 183
/etc/filesystems file
   accessing entries 364
/etc/hosts file
   closing 901
   retrieving host entries 900
/etc/utmp file
   accessing entries 469

Numerics
3-byte integers
   converting 581

A
a64l subroutine 1
abort subroutine 2
abs subroutine 3
absinterval subroutine 382
absolute path names
   copying 474
   determining 474
absolute value subroutines
   cabs 129
   cabsf 129
   cabsl 129
   fabsf 248
absolute values
   computing complex 523
   imaxabs 530
access control attributes
   setting 144
access control information
   changing 8
   retrieving 10
   setting 12, 14, 17, 25
access control subroutines
   acl_chg 8
   acl_fchg 8
   aclhdl 10
   acl_fput 12
   acl_hdl 14
   acl_get subroutine 10
   acl_hdl subroutine 12
   acl_hdl subroutine 14
   acl_get subroutine 10
   acl_put subroutine 12
   acl_set subroutine 14
   aclx_convert subroutine 16
   aclx_fget subroutine 17
   aclx_fput subroutine 25
   aclx_get subroutine 17
   aclx_gettypeinfo subroutine 20
   aclx_gettypes subroutine 21
   aclx_print subroutine 23
   aclx_print subroutine 23
   aclx_put subroutine 25
   aclx_scan subroutine 27
   aclx_scan subroutine 27
   chacl subroutine 144
   chmod subroutine 148
   chown subroutine 151
   chown subroutine 151
   fchacl subroutine 144
   fchmod subroutine 148
   fchown subroutine 151
   fchown subroutine 151
   frevoke subroutine 310
access subroutine 4
access subroutine 4
access subroutine 4
accounting subroutines
   addproj subroutine 31
   addproj subroutine 32
   chprojattr subroutine 158
   chprojattr subroutine 159
   getfirstprojdb subroutine 363
   getnextprojdb subroutine 391
   getproj subroutine 413
   getproj subroutine 414
   getproj subroutine 415
   proj_execve subroutine 1157
   projdballoc subroutine 1158
   projdbfinit subroutine 1159
   projdbfree subroutine 1160
acct subroutine 7
acl_chg subroutine 8
acl_fchg subroutine 8
acl_hdl subroutine 10
acl_fput subroutine 12
acl_hdl subroutine 14
acl_get subroutine 10
acl_put subroutine 12
acl_set subroutine 14
aclx_convert subroutine 16
aclx_fget subroutine 17
aclx_fput subroutine 25
aclx_get subroutine 17
aclx_gettypeinfo subroutine 20
aclx_gettypes subroutine 21
aclx_print subroutine 23
aclx_print subroutine 23
aclx_put subroutine 25

© Copyright IBM Corp. 1994, 2006
1329
aclx_scan subroutine 27
aclx_scanStr subroutine 27
acos subroutine 29
acosf subroutine 29
acosh subroutine 30
acoshf subroutine 30
acoshl subroutine 30
acosl subroutine 29
addproj subroutine 31
addprojdb subroutine 32
address identifiers 223
addssys subroutine 33
adjtime subroutine 35
advance subroutine 177
Advanced Accounting subroutines
  agg_arm_stat subroutine 36
  agg_lpar_stat subroutine 36
  agg_proc_stat subroutine 36
  buildproclist subroutine 125
  buildtranlist subroutine 126
  free_agg_list subroutine 36
  freetranslist subroutine 126
  getarmist subroutine 409
  getfilehdr subroutine 362
  getlparlist subroutine 409
  getproclist subroutine 409
  agg_arm_stat subroutine 36
  agg_lpar_stat subroutine 36
  agg_proc_stat subroutine 36
  aio_cancel subroutine 38
  aio_error subroutine 42
  aio_fsing subroutine 45
  aio_nwait subroutine 47
  aio_nwait_timeout subroutine 49
  aio_read subroutine 50
  aio_return subroutine 55
  aio_suspend subroutine 58
  aio_write subroutine 61
alarm subroutine 382
alloca subroutine 769
allocaLmb Subroutine 68
Application Programming Interface
  perfstat
    cpu 992
    cpu_total 994, 1002, 1006
    disk_total 995, 1000, 1004
    diskpath 998
    netbuffer 1003
    patingspace 1007
    protocol 1011
    reset 1013
arc sine subroutines
  asinf 91
arc tangent subroutines
  atan2f subroutine 93
  atan2l subroutine 93
  atanf subroutine 94
  atanl subroutine 94
archive files
  reading headers 691
ARM Subroutines
  arm_end subroutine 69
  arm_end Dual Call subroutine 71
  arm_getid subroutine 73
  arm_getid Dual Call subroutine 75
  arm_init subroutine 77
  arm_init Dual Call subroutine 79
  arm_start subroutine 81
  arm_start Dual Call subroutine 82
  arm_stop subroutine 84
  arm_stop Dual Call subroutine 86
  arm_update subroutine 88
  arm_update Dual Call subroutine 89
ASCII strings
  converting to floating-point numbers subroutine 96
  converting to Internet addresses subroutine 552
asctime subroutine subroutine 199
asctime_r subroutine 206
asctime64 subroutine subroutine 202
asctime64_r subroutine subroutine 204
asin subroutine subroutine 91
asinf subroutine subroutine 91
asinhf subroutine subroutine 90
asinhl subroutine subroutine 90
asinl subroutine subroutine 91
assert macro subroutine 92
asynchronous I/O
  reading subroutine 50
  writing subroutine 61
asynchronous I/O requests
  canceling subroutine 38
  listing subroutine 713
  retrieving error status subroutine 42
  retrieving return status subroutine 55
  suspending subroutine 58
  synchronizing asynchronous files subroutine 45
atan subroutine subroutine 94
atan2 subroutine subroutine 93
atan2f subroutine subroutine 93
atan2l subroutine subroutine 93
atanf subroutine subroutine 94
atanhl subroutine subroutine 95
atanh subroutine subroutine 95
atanhf subroutine subroutine 95
atanhl subroutine subroutine 95
atant subroutine subroutine 94
atantf subroutine subroutine 94
atant2 subroutine subroutine 94
atant2f subroutine subroutine 94
atantl subroutine subroutine 94
atant2l subroutine subroutine 94
atexit subroutine subroutine 242
atof subroutine subroutine 96
atoff subroutine subroutine 96
atojis subroutine subroutine 571
atol subroutine subroutine 98
atoll subroutine subroutine 98
atomic access subroutines
  compare_and_swap subroutine 176
  fetch_and_add subroutine 268
  fetch_and_or subroutine 269
audit bin files
  compressing and uncompressing subroutine 108
  establishing subroutine 100
audit records
  generating 104
  reading 111
  writing 112
audit subroutine 98
audit trail files
  appending records 104
auditbin subroutine 100
auditevents subroutine 102
auditing modes 105
auditing subroutines
  audit 98
  auditbin 100
  auditevents 102
  auditlog 104
  auditobj 105
  auditpack 108
  auditproc 109
  auditread 111
  auditwrite 112
auditlog subroutine 104
auditobj subroutine 105
auditpack subroutine 108
auditproc subroutine 109
auditread, auditread_r subroutines 111
auditwrite subroutine 112
authenticate 113
authenticatex subroutine 115
authentication subroutines
  ckuseracct 164
  ckuserID 166
  crypt 191
  encrypt 191
  getlogin 389
  getpass 397
  getuserpw 463
  newpass 891
  putuserpw 463
  setkey 191
authorizations 462
authorizations, compare 781
auxiliary areas
  creating 532
  destroying 532
  drawing 533
  hiding 534
  processing 546

B
base 10 logarithm functions
  log10f 736
base 2 logarithm functions
  log2 738
  log2f 738
  log2l 738
basename Subroutine 117
baud rates
  getting and setting 142
bcmp subroutine 118
bcopy subroutine 118
beep levels
  setting 534
BeginCriticalSection Subroutine 225
Bessel functions
  computing 119
binary files
  reading 307
  binary searches 123
  binding a process to a processor 120
  bit string operations 118
  box characters
    shaping 686
  brk subroutine 122
  bsearch subroutine 125
  bswrite subroutine 124
  buffered data
    writing to streams 252
buildproclist subroutine 125
buildtranlist subroutine 126
byte string operations 118
bzero subroutine 118

C
cabs subroutine 129
cabsf subroutine 129
cabsl subroutine 129
cacos subroutine 129
cacosf subroutine 129
cacosh subroutines 130
cacoshf subroutine 130
cacoshl subroutine 130
cacosl subroutine 129
calloc subroutine 769
carg subroutine 131
cargf subroutine 131
cargl subroutine 131
casin subroutine 131
casin subroutine 131
casinlh subroutine 132
catan subroutine 132
catan subroutine 132
catanh subroutine 133
catanhf subroutine 133
catanhl subroutine 133
catanl subroutine 132
catclose subroutine 134
catgets subroutine 135
catopen subroutine 136
cbrt subroutine 137
cbrtf subroutine 137
cbrtl subroutine 137
ccos, subroutine 138
cosf subroutine 138
cosh subroutine 139
coshf subroutine 139
coshl subroutine 139
cosl subroutine 138
CCSIDs
   converting 139
   ccsidtocs subroutine 139
   ceil subroutine 140
   ceilf subroutine 140
   ceiling value function
      ceil 140
      ceil 140
   ceil subroutine 140
   cexp subroutine 141
   cexpf subroutine 141
   cexpl subroutine 141
   cfgetospeed subroutine 142
   chacl subroutine 144
character conversion
   8-bit processing codes and 570
   code set converters 526, 527
   conv subroutines 183
   Japanese 571
   Kanji-specific 570
   multibyte to wide 795, 796
   translation operations 183
character manipulation subroutines
   _atojis 571
   _jistoa 571
   _tojlower 571
   _tojupper 571
   _tolower 183
   _toupper 183
   atojis 571
   conv 183
   ctype 573
   fgetc 343
   fputc 1299
   getc 343
   getchar 343
   getw 343
   isalnum 208
   isalpha 208
   isascii 208
   iscntrl 208
   isdigit 208
   isgraph 208
   isjalnum 573
   isjalpha 573
   isjdigit 573
   isjgraph 573
   isjhira 573
   isjis 573
   isjkanji 573
   isjkata 573
   isjlbytekana 573
   isjlower 573
   isjparen 573
   isparent 573
character manipulation subroutines (continued)
   isprint 208
   ispunct 208
   isspace 208
   isupper 208
   isxdigit 208
   jistoa 571
   kutentojis 571
   NCesc 183
   NCf1atchr 183
   NCtolower 183
   NCtoNLchar 183
   NCtoupper 183
   NCunescape 183
   putc 1299
   putchar 1299
   putw 1299
   toascii 183
   tojhirra 571
   tojkanji 571
   tojkanji 571
   tojlower 571
   tojupper 571
   tolower 183
toujirs 571
toupper 183
character shaping 679
character testing
   isblank 559
characters
   classifying 208, 573
   returning from input streams 343
   writing to streams 1299
charsetID
   multibyte character 192
   chdir subroutine 147
   chmod subroutine 148
   chown subroutine 151
   chownx subroutine 151
   chpass subroutine 154
   chpassx subroutine 156
   chprojattr subroutine 158
   chprojattrdb subroutine 159
   chroot subroutine 160
   chsys subroutine 162
   cimag subroutine 163
   cimag subroutine 163
   cimag subroutine 163
   cimagt subroutine 163
   cintosj subroutine 570
   cuseracct subroutine 164
   cuserID subroutine 166
   class subroutine 167
   clearerr macro 267
   clock subroutine 169
   clock subroutines
      clock_gettime subroutine 169
      clock_gettime subroutine 169
      clock_gettime subroutine 169
      clock_gettime subroutine 169
      clock_gettime subroutine 169
      clock_gettime subroutine 169
      clock_gettime subroutine 169
      clock_gettime subroutine 169
      clock_gettime subroutine 169
clock_settime subroutine 170
clog subroutine 174
clogf subroutine 174
clogl subroutine 174
close subroutine 175
closedir subroutine 933
closedir64 subroutine 933
code sets
closing converters 526
converting names 139
opening converters 527
coded character set IDs
converting 139
code sets
command-line flags
returning 392
Common Host Bus Adapter library
HBA_SetRNIDMgmtInfo 518
compare_and_swap subroutine
atomic access 176
compile subroutine 177
complementary error subroutines
erfcl 227
complex arc cosine subroutines
cacos 129
cacosf 129
cacosl 129
complex arc hyperbolic cosine subroutines
cacosh 130
cacoshf 130
cacoshl 130
complex arc hyperbolic sine subroutines
casin 132
casinf 132
casinl 132
complex arc hyperbolic tangent subroutines
catanh 133
catanhf 133
catanhl 133
complex arc sine subroutines
casin 131
casinf 131
casinl 131
complex argument subroutines
carg 131
cargf 131
cargl 131
complex conjugate subroutines
cconj 182
conjg 182
complex cosine functions
ccos 138
ccosf 138
ccosl 138
complex exponential functions
cexp 141
cexpf 141
cexpl 141
complex hyperbolic cosine functions
ccosh 139
ccoshf 139
complex hyperbolic cosine functions (continued)
ccoshl 139
complex hyperbolic sine subroutines
csinh 194
csinhf 194
csinhl 194
complex hyperbolic tangent subroutines
cctanh 198
cctanhf 198
cctanhl 198
complex imaginary functions
cimag 163
cimagf 163
cimagl 163
complex natural logarithm functions
clog 174
clogf 174
clogl 174
complex power subroutines
cpow 189
cpowf 189
cpowl 189
complex projection subroutines
cproj 189
cprojf 189
cprojl 189
complex tangent functions
cctan 132
cctanf 132
cctanl 132
Complex tangent subroutines
ctan 197
cctanf 197
cctanl 197
Computes the base 2 exponential.
exp2 246
exp2f 246
exp2l 246
confstr subroutine 181
complex conjugate subroutines
convert subroutines 182
complex conjugate subroutines
using asctime subroutine 206
using ctime subroutine 206
using gmtime subroutine 206
using localtime subroutine 206
conversion
date and time representations 206
date and time to string representation
using asctime subroutine 206
using ctime subroutine 206
using gmtime subroutine 206
using localtime subroutine 206
converter subroutines
btowc 124
fwscanf 327
iconv_close 526
iconv_open 527
jcode 570
mbrlen 783
mbrtowc 784
mbstowcs 789
mbsinit 788
mbsrtowcs 793
converter subroutines *(continued)*

swscanf 327
wscanf 327
copysignf subroutine 185
copysignl subroutine 185
core files

coredump subroutine 333
gencore subroutine 333
coredump subroutine 333
cos subroutine 187
cosf subroutine 187
cosh subroutine 188
coshf subroutine 188
coshl subroutine 188
cosine subroutines

acosf 29
acosl 29
cosf 187
cosl 187
cosl subroutine 187
cpow subroutine 189
cpowf subroutine 189
cpowl subroutine 189
cproj subroutine 189
cprojf subroutine 189
cprojl subroutine 189
creal subroutine 190
crealf subroutine 190
crel subroutine 190
creat subroutine 925

Critical Section Subroutines

BeginCriticalSection Subroutine 225
EnableCriticalSections Subroutine 225
EndCriticalSection Subroutine 225
crypt subroutine 191
csid subroutine 192
csin subroutine 193
csinf subroutine 193
csinf subroutine 194
csinhf subroutine 194
csinhl subroutine 194
csinl subroutine 193
csitojis subroutine 570
csitoj subroutine 570
csqrt subroutine 194
csqrtf subroutine 194
csqrtl subroutine 194
cstoccsid subroutine 139
ct_gen 195
c_hookx 195
c_trcon 197
ctan subroutine 197
ctanf subroutine 197
ctanf subroutine 198
ctanhf subroutine 198
ctanhl subroutine 198
ctanl subroutine 197
ctermid subroutine 198
ctime subroutine 199
ctime_r subroutine 206
ctime64 subroutine 202
cftime64_r subroutine 204
cftime subroutines 208
cube root functions

cbrtf 137
cbrtl 137
cu_tojis subroutine 570
cu_toj subroutine 570
current process credentials

reading 398

current process environment

reading 400
current processes

getting user name 210
group ID

initializing 553
returning 378
path name of controlling terminal 198
user ID
returning 447
current working directory
getting path name 354
cursor positions

setting 549
cuserid subroutine 210
data arrays

encrypting 191
data locks 1015
data sorting subroutines

bsearch 123
tfw 320
hcreate 522
hdestroy 522
hsearch 522
insque 554
lfind 754
lsearch 754
remque 554
data space segments

changing allocation 122
date

changing 440
date format conversions 199
defect 219851 1241
defect 220239 407
defssys subroutine 211
delssys subroutine 211
descriptor tables
getting size 358
diffmtime subroutine 199
diffmtime64 subroutine 202
directories

changing 147
changing root 160
creating 826
directory stream operations 933
generating path names 477
gcurrent path name of current directory 354
expm1l subroutine 247  
exponential functions  
  computing 244  
exponential subroutines  
  expf 244  
  expml, 247  
  expml 247  
extended attribute subroutines  
  getea 359  
  listea 718  

F  

f_hpmsgetcounters subroutine 519  
f_hpmsgettimeandcounters subroutine 519  
f_hpminit subroutine 519  
f_hpmstart subroutine 519  
f_hpmstop subroutine 519  
f_hpmterminate subroutine 519  
f_hpmstart subroutine 519  
f_hpmstop subroutine 519  
fabs subroutine 248  
fabsf subroutine 248  
fabsl subroutine 248  
facessx subroutine 4  
fattach Subroutine 249  
fcacat subroutine 144  
fcdir Subroutine 250  
fchmod subroutine 148  
fchown subroutine 151  
fchownx subroutine 151  
fclear subroutine 251  
fclose subroutine 252  
fcntl subroutine 254  
fcvt subroutine 224  
fdetach Subroutine 260  
fdim subroutine 261  
fdimf subroutine 261  
fdiml subroutine 261  
fopen subroutine 284  
feclaisncept subroutine 262  
fegetenv subroutine 263  
fegetexceptflag subroutine 263  
fegetround subroutine 264  
feholdexcept subroutine 265  
feof macro 267  
feralseaexcept subroutine 268  
ferror macro 267  
fesetqenv subroutine 263  
fesetqexceptflag subroutine 263  
fesetqround subroutine 264  
fetch_and_add subroutine  
  atomic access 268  
fetch_and_and subroutine  
  atomic access 269  
fetch_and_or subroutine  
  atomic access 269  
fetestexcept subroutine 270  
feupdateenv subroutine 271  
fflush subroutine 252  
ffs subroutine 118  
fgetc subroutine 343  
fgetpos subroutine 314  
fgets subroutine 429  
fgetwc subroutine 472  
fgetws subroutine 475  
FIFO files  
  creating 828  
file access permissions  
  changing 144, 148  
file descriptors  
  checking I/O status 1118  
  closing associated files 175  
  controlling 254  
  establishing connections 925  
  performing control functions 556  
file names  
  constructing unique 830  
file ownership  
  changing 151  
file permissions  
  changing 144, 148  
file pointers  
  moving read-write 756  
file subroutines  
  access 4  
  accessx 4  
  dup 254  
  dup2 254  
  endutent 469  
  facessx 4  
  fclear 251  
  fcntl 254  
  ffinfo 272  
  finfo 272  
  flock 733  
  flockfile 273  
  fpathconf 969  
  fsync 317  
  fsync_range 317  
  ftrylockfile 273  
  funlockfile 273  
  getc_unlocked 345  
  getchar_unlocked 345  
  getenv 360  
  getutent 469  
  getutid 469  
  getutline 469  
  lockf 733  
  lockfx 733  
  lseek 756  
  mkfifo 828  
  mknod 828  
  mkstemp 830  
  mktemp 830  
  nlist 898  
  nlist64 898  
  pathconf 969  
  pclose 991  
  pipe 1014  
  popen 1124
file subroutines (continued)
  putc_unlocked 345
  putchar_unlocked 345
  putenv 1304
  pututline 469
  setutent 469
  utmpname 469

file system subroutines
  confstr 181
  endfsent 364
  endvfsent 471
  fscntl 312
  getfsent 364
  getfsspec 364
  getfstype 364
  getvfsbyflag 471
  getvfsbyname 471
  getvfsbytype 471
  getvfsent 471
  mntctl 838
  setfsent 364
  setvfsent 471

file systems
  controlling operations 312
  retrieving information 364
  returning mount status 838

file trees
  searching recursively 320

file-implementation characteristics 969

fileno macro 267

files
  binary 307
  closing 175
  creating 828
  creating links 712
  creating space at pointer 251
  determining accessibility 4
  establishing connections 925
  generating path names 477
  getting name list 898
  locking and unlocking 733
  opening 925
  opening streams 284
  reading 307
  reading asynchronously 50
  repositioning pointers 314
  revoking access 310
  systems
    getting information about 430
    writing asynchronously 61
    writing binary 307

finfo subroutine 272

finite subroutine 167

finite testing
  isfinite 561

first-in-first-out files 828

flags
  returning 392

floating point multiply-add (continued)
  fmaf 276
  fmal 276

floating point numbers
  ldexpf 693
  ldexpl 693
  nextafterf 889
  nextafterl 889
  nexttoward 889
  nexttowardl 889

floating-point absolute value functions
  computing 274

floating-point environment
  feholdexcept 265
  feupdateenv 271

floating-point environment variables
  fegetenv, 263
  fesetenv 263

floating-point exception
  feraiseexcept 268
  fetestexcept 270

floating-point exceptions 294, 298, 302
  changing floating point status and control
    register 300
    feclearexcept 262
    flags 292
    querying process state 302
    testing for occurrences 296, 297

floating-point number subroutines
  fdim 261
  fdimf 261
  fdiml 261

floating-point numbers
  converting to strings 224
  determining classifications 167
  fmax 277
  fmaxf 277
  fmaxl 277
  fminf 278
  fminl 278
  fmodf 279
  manipulating 839
  modff 839
  reading and setting rounding modes 299
  rounding 274

floating-point rounding subroutines
  nearbyint 888
  nearbyintf 888
  nearbyintl 888

floating-point status flags
  fegetexceptflag 263
  fesetexceptflag 263

floating-point subroutines 294, 298, 300, 302, 304
  fp_sh_info 300
  fp_sh_trap_info 300

floating-point trap control 290

flock subroutine 733

flockfile subroutine 273

floor functions
  floor 274
floor subroutine 274
floorf subroutine 274
floorl subroutine 274
fma subroutine 276
fmaf subroutine 276
fmal subroutine 276
fmax subroutine 277
fmaxf subroutine 277
fmaxl subroutine 277
fmin subroutine 277
fminf subroutine 278
fminl subroutine 278
fmod subroutine 279
fmodf subroutine 279
fmodl subroutine 279
fputc subroutine 1299
fputc subroutine 1148
fputs subroutine 1309
fputwc subroutine 1317
fputws subroutine 1319
fread subroutine 307
free subroutine 769
free_arg_list subroutine 36
freeimb Subroutine 310
freetranlist subroutine 126
freopen subroutine 284
frevoke subroutine 310
frexp subroutine 311
frezp subroutine 311
fresx subroutine 311
fscntl subroutine 312
fseek subroutine 314
fsetpos subroutine 318
fsetpos subroutine 314
ftime subroutine 440
ftok subroutine 318
ftok subroutine 318
fwide subroutine 322
fwprintf subroutine 323
fwrite subroutine 323
fwrite subroutine 307
fscanf subroutine 327
fseek subroutine 314
fseek subroutine 314
ftime subroutine 440
ftok subroutine 318
ftok subroutine 318
fwide subroutine 322
fwprintf subroutine 323
fwrite subroutine 307
fscanf subroutine 327
G

gai_strerror subroutine 331
gamma functions
computing natural logarithms 332
gamma subroutine 332
gcd subroutine 776
gcvt subroutine 224
gencode subroutine 333
genpavgvalue Subroutine 335
get_malloc_log subroutine 336
get_malloc_log_subroutine 337
get_speed subroutine 338
getargs Subroutine 339
getarmalist subroutine 409
getaudithostattr, IDtohost, hosttoID, nexthost or
putaudithostattr subroutine 340
getauthdb subroutine 342
getauthdb_r subroutine 342
getc subroutine 343
getc subroutine 343
getc_unlocked subroutine 345
getchar subroutine 343
getchar subroutine 343
getchar_unlocked subroutine 345
getconfattr subroutine 346
getconfattr subroutine 346
getcontext or setcontext Subroutine 353
getcwd subroutine 354
getdate Subroutine 355
getdtables subroutine 358
getenv subroutine 360
getenv subroutine 360
getenv subroutine 360
HBA subroutines (continued)
  HBA_SendRPS 515
  HBA_CloseAdapter Subroutine 481
  HBA_FreeLibrary Subroutine 482
  HBA_GetAdapterAttributes Subroutine 482
  HBA_GetAdapterName Subroutine 485
  HBA_GetDiscoveredPortAttributes Subroutine 482
  HBA_GetEventBuffer subroutine 486
  HBA_GetFC4Statistics subroutine 487
  HBA_GetFCPStatistics subroutine 489
  HBA_GetFcpTargetMapping Subroutine 491
  HBA_GetFcpTargetMappingV2 subroutine 490
  HBA_GetNumberOfAdapters Subroutine 492
  HBA_GetPortAttributes Subroutine 482
  HBA_GetPortAttributesByWWN Subroutine 482
  HBA_GetPortStatistics Subroutine 494
  HBA_GetRNIDMgmtInfo Subroutine 495
  HBA_GetVersion Subroutine 496
  HBA_LoadLibrary Subroutine 497
  HBA_OpenAdapter Subroutine 498
  HBA_OpenAdapterByWWN subroutine 498
  HBA_RefreshInformation Subroutine 499
  HBA_ScsiInquiryV2 subroutine 500
  HBA_ScsiReadCapacityV2 subroutine 502
  HBA_ScsiReportLUNsV2 subroutine 504
  HBA_SendCTPassThru Subroutine 505
  HBA_SendCTPassThruV2 subroutine 506
  HBA_SendReadCapacity Subroutine 507
  HBA_SendReportLUNs Subroutine 508
  HBA_SendRNID Subroutine 511
  HBA_SendRNIDV2 subroutine 512
  HBA_SendRPL subroutine 514
  HBA_SendRPS subroutine 515
  HBA_SendScsiInquiry Subroutine 516
  HBA_SetRNIDMgmtInfo Subroutine 518
  hcreate subroutine 522
  hdestroy subroutine 522

Host Bus Adapter API (continued)
  HBA_SendScsiInquiry 516
  HBA_SetRNIDMgmtInfo 518
  hpmGetCounters subroutine 519
  hpmGetTimeAndCounters subroutine 519
  hpmInit subroutine 519
  hpmStart subroutine 519
  hpmStop subroutine 519
  hpmTerminate subroutine 519
  hpmTstart subroutine 519
  hpmTstop subroutine 519
  hsearch subroutine 522
  hyperbolic cosine subroutines
    coshf 188
    coshl 188
  hypot subroutine 523
  hypotf subroutine 523
  hypotl subroutine 523

I

I/O asynchronous subroutines
  aio_cancel 38
  aio_error 42
  aio_fsync 45
  aio_nwait 47
  aio_nwait_timeout 49
  aio_read 50
  aio_return 55
  aio_suspend 58
  aio_write 61
  lio_listio 713
  poll 1118

I/O low-level subroutines 175, 925
  creat 925
  open 925

I/O requests
  canceling 38
  listing 713
  retrieving error status 42
  retrieving return status 55
  suspending 58

I/O stream macros
  clearerr 267
  feof 267
  ferror 267
  fileno 267

I/O stream subroutines
  fclose 252
  fdopen 284
  fflush 252
  fgetc 343
  fgetpos 314
  fgets 429
  fgetwc 472
  fgetws 475
  fopen 284
  fprintf 1148
  fputc 1299
  puts 1309
  putwc 1317
I/O stream subroutines (continued)
- fputws 1319
- fread 307
- freopen 284
- fseek 314
- fsetpos 314
- ftell 314
- fwrite 307
- getc 343
- getchar 343
- gets 429
- getw 343
- getwc 472
- getwchar 472
- getws 475
- printf 1148
- putc 1299
- putchar 1299
- puts 1309
- putw 1299
- putwc 1317
- putwchar 1317
- putws 1319
- rewind 314
- sprintf 1148
- swprintf 1148
- vprintf 1148
- vsprintf 1148
- wprintf 323
- vsprint 1148
- wprintf 1148

I/O terminal subroutines
- cfsetispeed 142
- ioctl 556
- ioctl32 556
- ioctl32x 556
- ioctlx 556

iconv_close subroutine 526
iconv_open subroutine 527

identification subroutines
- endgrent 366
- endw 417
- getconfattr 346
- getgrent 366
- getgrgid 366
- getgrnam 366
- getgroupattr 370
- getpwent 417
- getpwnam 417
- getpwpuid 417
- getwcattr 435
- getuinfo 448
- getuserattr 346, 449
- IDTogroup 370
- IDTouser 449
- nextgrent 370
- nextw 449
- putconfattr 346

identification subroutines (continued)
- putgroupattr 370
- putpwent 417
- putwcattr 435
- putuserattr 449
- setgrent 366
- setpwent 417
- idpthreadsa 191
- IDTogroup subroutine 370
- IDTouser subroutine 449
- IEE Remainders
- computing 222
- ilogb subroutine 529
- ilogbf subroutine 529
- ilogbl subroutine 529
- IMAIXMapping subroutine 531
- IMAuxCreate callback subroutine 532
- IMAuxDestroy callback subroutine 532
- IMAuxDraw callback subroutine 533
- IMAuxHide callback subroutine 534
- imaxabs subroutine 530
- imaxdiv subroutine 530
- IMBeep callback subroutine 534
- IMClose subroutine 535
- IMCreate subroutine 536
- IMDestroy subroutine 536
- IMFilter subroutine 537
- IMFreeKeymap subroutine 538
- IMIndicatorDraw callback subroutine 538
- IMIndicatorHide callback subroutine 539
- IMInitialize subroutine 540
- IMInitializeKeymap subroutine 541
- IMioctl subroutine 542
- IMLookupString subroutine 544
- IMProcess subroutine 545
- IMProcessAuxiliary subroutine 546
- IMQueryLanguage subroutine 547
- IMSimpleMapping subroutine 548
- IMTextCursor callback subroutine 549
- IMTextDraw callback subroutine 550
- IMTextGet callback subroutine 550
- IMTextStart callback subroutine 551
- imul_dbl subroutine 3
- incinterval subroutine 382
- inet_aton subroutine 552

infinity values
- isnf 563

initgroups subroutine 553

initialize subroutine 553

input method
- checking language support 547
- closing 535
- control and query operations 542
- creating instance 536
- destroying instance 536
- initializing for particular language 540

input method keymap
- initializing 538, 541

- mapping key and state pair to string 531, 544, 548
input method subroutines
  callback functions
    IMAuxCreate 532
    IMAuxDestroy 532
    IMAuxDraw 533
    IMAuxHide 534
    IMBeep 534
    IMIndicatorDraw 538
    IMIndicatorHide 539
    IMTextCursor 549
    IMTextDraw 550
    IMTextHide 550
    IMTextStart 551
  IMAIXMapping 531
  IMClose 535
  IMCreate 536
  IMDestroy 536
  IMFilter 537
  IMFreeKeymap 538
  IMInitialize 540
  IMInitializeKeymap 541
  IMoctl 542
  IMLookupString 544
  IMProcess 545
  IMProcessAuxiliary 546
  IMQueryLanguage 547
  IMSimpleMapping 548

input streams
  reading character string from 475
  reading single character from 472
  returning characters or words 343
  insque subroutine 554
  install_lwcf_handler() subroutine 555

integers
  computing absolute values 3
  computing division 3
  computing double-precision multiplication 3
  performing arithmetic 776

Internet addresses
  converting to ASCII strings 552

interoperability subroutines
  ccsidtocs 139
  cstoccsid 139

interprocess channels
  creating 1014
  interprocess communication keys 318
  interval timers
    allocating per process 444
    manipulating expiration time 382
    returning values 382
  inverse hyperbolic cosine subroutines
    acoshf 30
    acoshl 30
  inverse hyperbolic functions
    computing 30, 90
  inverse hyperbolic sine subroutines
    asinhf 90
    asinhl 90
  inverse hyperbolic tangent subroutines
    atanhf 95
    atanhl 95
  invert subroutine 776
  ioctl subroutine 556
  ioctl32 subroutine 556
  ioctl32x subroutine 556
  ioctlx subroutine 556
  is wctype subroutine 569
  isalnum subroutine 208
  isalpha subroutine 208
  isascii subroutine 208
  isblank subroutine 559
  iscntrl subroutine 208
  isdigit subroutine 208
  isendwin Subroutine 560
  isfinite macro 561
  isgraph subroutine 208
  isgreater macro 561
  isgreaterequal subroutine 562
  isinf subroutine 563
  isless macro 563
  islessequal macro 564
  islessgreater macro 565
  islower subroutine 208
  isnan subroutine 167
  isnormal macro 565
  isprint subroutine 208
  ispunct subroutine 208
  isspace subroutine 208
  isunordered macro 566
  isupper subroutine 208
  iswalnum subroutine 567
  iswalpha subroutine 567
  iswblank subroutine 568
  iswcntrl subroutine 567
  iswctype subroutine 569
  iswdigit subroutine 567
  iswgraph subroutine 567
  iswlower subroutine 567
  iswprint subroutine 567
  iswpunct subroutine 567
  iswspace subroutine 567
  iswupper subroutine 567
  iswxdigit subroutine 567
  isxdigit subroutine 208
  itom subroutine 776
  itrunc subroutine 274

J
  j0 subroutine 119
  j1 subroutine 119
  Japanese conv subroutines 571
  Japanese ctype subroutines 573
  jcode subroutines 570
  JFS
    controlling operations 312
    JIS character conversions 570
  jistoa subroutine 571
  jistosj subroutine 570
  jistouj subroutine 570
  jn subroutine 119
  Journaled File System 254
K
Kanji character conversions 570
keyboard events
    processing 537, 545
kill subroutine 575
killpg subroutine 575
kleenup subroutine 576
knlist subroutine 577
kpidstate subroutine 579
kutentojis subroutine 571
L
l3tol subroutine 581
l64a subroutine 1
l64a_r subroutine 582
labs subroutine 3
LAPI_Addr_get subroutine 583
LAPI_Addr_set subroutine 584
LAPI_Address subroutine 586
LAPI_Address_init subroutine 587
LAPI_Address_init64 589
LAPI_Amsend subroutine 591
LAPI_Amsendv subroutine 596
LAPI_Get subroutine 605
LAPI_Getcnt subroutine 608
LAPI_Getv subroutine 609
LAPI_Gfence subroutine 613
LAPI_Init subroutine 614
LAPI_Msg_string subroutine 619
LAPI_Msgpoll subroutine 621
LAPI_Nopoll_wait subroutine 623
LAPI_Probe subroutine 624
LAPI_Purge_totask subroutine 625
LAPI_Put subroutine 627
LAPI_Putv subroutine 629
LAPI_Qenv subroutine 633
LAPI_Resume_totask subroutine 636
LAPI_Rmw subroutine 637
LAPI_Rmw64 subroutine 641
LAPI_Senv subroutine 645
LAPI_Setcnt subroutine 647
LAPI_Setcntnt_wstatus subroutine 649
LAPI_Term subroutine 650
LAPI_Util subroutine 652
LAPI_Waitcnt subroutine 663
LAPI_Xfer structure types 666
LAPI_Xfer subroutine 665
lapi_xfer_type_t 666
layout values
    querying 682
    setting 684
    transforming text 687
LayoutObject
    creating 678
    freeing 690
lcorg48 subroutine 220
ldaclose subroutine 692
ldahread subroutine 691
ldahread subroutine 701
ldclose subroutine 692
ldexplo subroutine 693
ldexpf subroutine 693
ldexpl subroutine 693
ldfread subroutine 694
ldgetname subroutine 696
ldiv subroutine 3
ldlimit subroutine 698
lditem subroutine 698
ldinseek subroutine 699
ldread subroutine 698
ldseek subroutine 699
ldnseek subroutine 703
ldnsfread subroutine 704
ldnsfseek subroutine 706
ldohseek subroutine 700
ldopen subroutine 701
ldrseek subroutine 703
ldshread subroutine 704
ldseek subroutine 706
ldtindex subroutine 707
ldtread subroutine 708
ldtseek subroutine 709
lfind subroutine 754
lgamma subroutine 710
lgammaf subroutine 710
lgamml subroutine 710
libhpm subroutines
    f_hpmgetcounters 519
    f_hpmgettimeandcounters 519
    f_hpminit 519
    f_hpmstart 519
    f_hpmstop 519
    f_hpmterminate 519
    f_hpmtstart 519
    f_hpmtstop 519
    hpmGetCounters 519
    hpmGetTimeAndCounters 519
    hpmInit 519
    hpmStart 519
    hpmStop 519
    hpmTerminate 519
    hpmTstart 519
    hpmTstop 519
linear searches 754
lineout subroutine 711
link subroutine 712
lio_listio subroutine 713
listea subroutine 718
llabs subroutine 3
lldiv subroutine 3
llrint subroutine 719
llrintf subroutine 719
llrintl subroutine 719
llround subroutine 720
llroundf subroutine 720
llroundl subroutine 720
load subroutine 721
loadbind subroutine 725
loadquery subroutine 726
locale subroutines
localeconv 728
nl_langinfo 897
locale-dependent conventions 728
localeconv subroutine 728
locales
returning language information 897
localtime subroutine 199
localtime_r subroutine 206
localtime64 subroutine 202
localtime64_r subroutine 204
lockf subroutine 733
lockfx subroutine 733
log gamma functions
lgamma 710
lgammaf 710
lgammal 710
log subroutine 740
log10 subroutine 736
log10f subroutine 736
log1p subroutine 737
log1pf subroutine 737
log1pl subroutine 737
log2 subroutine 738
log2f subroutine 738
log2l subroutine 738
logarithmic functions
computing 244
logb subroutine 739
logbf subroutine 739
logbl subroutine 739
logf subroutine 740
logical volumes
querying 757
login name
getting 389, 390
loginfailed Subroutine 741
loginrestrictions Subroutine 743
loginrestrictionsx subroutine 746
loginsuccess Subroutine 748
long integers
converting to strings 582
long integers, converting
  to 3-byte integers 581
  to base-64 ASCII strings 1
lpar_get_info subroutine 750
lpar_set_resources subroutine 751
rand48 subroutine 220
rint subroutine 752
rintf subroutine 752
rintl subroutine 752
round subroutine 753
roundf subroutine 753
roundl subroutine 753
isearch subroutine 754
iseek subroutine 756
itoi3 subroutine 581
LVM logical volume subroutines
  lvm_querypv 761
LVM physical volume subroutines
  lvm_querypv 761
LVM volume group subroutines
  lvm_queryvg 765
  lvm_queryvgs 768
lvm_querylv subroutine 757
lvm_querypv subroutine 761
lvm_queryvg subroutine 765
lvm_queryvgs subroutine 768
M
m_in subroutine 776
m_out subroutine 776
macros
  assert 92
madd subroutine 776
madvise subroutine 778
makecontext Subroutine 779
mallinfo subroutine 769
mallinfo_heap subroutine 769
malloc subroutine 769
mallopt subroutine 769
mapped files
  synchronizing 881
MatchAllAuths Subroutine 781
MatchAllAuthsList Subroutine 781
MatchAnyAuthsList Subroutine 781
math errors
  handling 780
matherr subroutine 780
mblen subroutine 782
mbrlen subroutine 783
mbrtowc subroutine 784
mbsadvance subroutine 785
mbscat subroutine 786
mbschr subroutine 787
mbscmp subroutine 786
mbscmpy subroutine 786
mbsinit subroutine 788
mbsinval subroutine 789
mbslen subroutine 789
mbsncat subroutine 790
mbsncmp subroutine 790
mbsncpy subroutine 790
mbspbrk subroutine 791
mbsrchr subroutine 792
mbsrtowcs subroutine 793
mbstomb subroutine 794
mbstowcs subroutine 795
mbswidth subroutine 796
mbtowc subroutine 796
mcmp subroutine 776
mdiv subroutine 776
memccpy subroutine 798
memchr subroutine 798
memcmp subroutine 798
memcpncpy subroutine 798
memmove subroutine 798
memory allocation 769
memory area operations 798
memory management  
controlling execution profiling  840, 841, 847  
defining addresses  223  
defining available paging space  1161  
disclaiming memory content  214  
generating IPC keys  847  
returning system page size  395  
memory management subroutines  
alloca  769  
calloc  769  
disclaim  214  
free  769  
ftok  318  
gai_strerror  331  
getpagesize  395  
madvise  778  
mallinfo  769  
mallinfo_heap  769  
malloc  769  
mallopt  769  
memccpy  798  
memchr  798  
memcmp  798  
memcpy  798  
memmove  798  
memset  798  
mincore  799  
mmap  769  
mallinfo_heap  769  
malloc  769  
mallopt  769  
memccpy  798  
memchr  798  
memcmp  798  
memcpy  798  
memmove  798  
memset  798  
malloc  769  
malloc  769  
malloc  769  
memccpy  798  
memchr  798  
memcmp  798  
memcpy  798  
memmove  798  
memset  798  
mmap  834  
moncontrol  840  
monitor  841  
monstartup  840  
mprotect  850  
msem_init  865  
msem_lock  866  
msem_remove  868  
msem_unlock  869  
sleep  880  
msync  881  
munmap  884  
mwakeup  885  
psdanger  1161  
realloc  769  
memory mapping  
adviseing system of paging behavior  778  
checking I/O status  1118  
determining page residency status  799  
determining page residency status  799  
file-system objects  834  
modifying access protections  850  
putting a process to sleep  880  
semaphores  
initializing  865  
locking  866  
removing  868  
unlocking  869  
synchronizing mapped files  881  
unmapping regions  884  
waking a process  885  
memory pages  
determining residency  799  
memory semaphores  
initializing  865  
locking  866  
memory semaphores (continued)  
putting a process to sleep  880  
removing  868  
unlocking  869  
waking a process  885  
memory subroutines  
allocaclmb  68  
freelmmb  310  
memset subroutine  798  
message catalogs  
closing  134  
opening  136  
retrieving messages  135  
message control operations  870  
message facility subroutines  
catclose  134  
catgets  135  
catopen  136  
message queue identifiers  872  
message queue subroutines  
mq_receive  861  
mq_send  863  
mq_timedreceive  861  
mq_timedsend  863  
message queues  
checking I/O status  1118  
reading messages from  873  
receiving messages from  878  
sending messages to  876  
Micro-Partitioning  
lpar_get_info  750  
min subroutine  776  
mcore subroutine  799  
kdir subroutine  826  
mkfifo subroutine  828  
mknod subroutine  828  
mkstemp subroutine  830  
mktemp subroutine  830  
mktime subroutine  199  
mktime64 subroutine  202  
mlockall subroutine  831, 832  
mmap subroutine  834  
mntctl subroutine  838  
mod subroutine  839  
modff subroutine  839  
modfl subroutine  839  
modulo remainders  
computing  274  
moncontrol subroutine  840  
monitor subroutine  841  
monstartup subroutine  847  
mot subroutine  776  
move subroutine  776  
mprotect subroutine  850  
mq_close subroutine  852  
mq_setattr subroutine  853  
mq_notify subroutine  854  
mq_open subroutine  855  
mq_receive subroutine  857, 861  
mq_send subroutine  858, 863  
mq_setattr subroutine  860  
Index  1345
mq_timedreceive subroutine 861
mq_timedsend subroutine 863
mq.unlink subroutine 864
mrand48 subroutine 220
msem init subroutine 865
msem lock subroutine 866
msem remove subroutine 868
msem unlock subroutine 869
msgctl subroutine 870
msgget subroutine 872
msgget subroutine 873
msgsnd subroutine 876
msgxrcv subroutine 878
msleep subroutine 880
msqrt subroutine 776
msub subroutine 776
msync subroutine 881
mt__trce() subroutine 882
mult subroutine 776
multibyte character subroutines
  csid 192
  mbcat 786
  mbsadvan se 785
  mbscmp 786
  mbscp y 786
  mbsinvalid 789
  mbslen 789
  mbsncmp 790
  mbsncpy 790
  mbstruck 787
  mbstrchr 792
  mbstomb 794
  mbstowcs 795
  mbstowc 796
multibyte characters
  converting to wide 795, 796
determining display width of 796
determining length of 782
determining number of 789
extracting from string 794
locating character sequences 791
locating next character 785
locating single characters 787, 792
operations on null-terminated strings 786, 790
returning charsetID 192
validating 789
munlockall subroutine 831, 832
munmap subroutine 884
mwakeup subroutine 885

nan subroutine 886

NaN
  nan 886
  nanf 886
  nanl 886
  nan subroutine 886

natural logarithm functions
  log 740
  logf 740
  log1pf 737
  log1pl 737

NCesc subroutine 183
NCfCfchr subroutine 183
NCtolower subroutine 183
NCtoNwchar subroutine 183
NCtolower subroutine 183
nextbyint subroutine 888
nextbyintf subroutine 888
nextbyintl subroutine 888
nearby subroutine 274
network host entries
  retrieving 900
new-process image file 235
newpass subroutine 891
newpassx subroutine 893
nextafter subroutine 889
nextafter subroutine 889
nextafter subroutine 889
nextafter subroutine 889
nextgroup subroutine 370
nextgroup subroutine 370
nextgroup subroutine 379
nextrole subroutine 426
nexttocl subroutine 379
nexttoward subroutine 379
nexttoward subroutine 379
nexttoward subroutine 379
nexttoward subroutine 379
nexttowardl subroutine 379
nextuser subroutine 449
nextuser subroutine 449
nextuser subroutine 449
nextuser subroutine 449
nextusracl subroutine 467
nftw subroutine 894
nice subroutine 407
nl_langinfo subroutine 897
nlist subroutine 898
nlist64 subroutine 898
nrand48 subroutine 220
number manipulation function
  copysign 185
  copysignl 185

numbers
  generating pseudo-random 220
numerical manipulation subroutines 332
  a64l 1
  abs 3
  acos 29
  acosf 29
  acosh 30
  acosl 29
  asin 91
  asinh 90
  asinl 91
  atan 94
  atan2 93
  atan2f 93
  atan2l 93

1346  Technical Reference, Volume 1: Base Operating System and Extensions
### Numerical Manipulation Subroutines

<table>
<thead>
<tr>
<th>Subroutine</th>
</tr>
</thead>
<tbody>
<tr>
<td>atanf 94</td>
</tr>
<tr>
<td>atanl 94</td>
</tr>
<tr>
<td>atanh 95</td>
</tr>
<tr>
<td>atanhf 95</td>
</tr>
<tr>
<td>atanhl 95</td>
</tr>
<tr>
<td>atof 96</td>
</tr>
<tr>
<td>atoll 98</td>
</tr>
<tr>
<td>cabs 523</td>
</tr>
<tr>
<td>ceil 140</td>
</tr>
<tr>
<td>ceilf 140</td>
</tr>
<tr>
<td>ceilh 140</td>
</tr>
<tr>
<td>class 167</td>
</tr>
<tr>
<td>cos 187</td>
</tr>
<tr>
<td>div 3</td>
</tr>
<tr>
<td>drand48 220</td>
</tr>
<tr>
<td>drem 222</td>
</tr>
<tr>
<td>ecvt 224</td>
</tr>
<tr>
<td>erand48 220</td>
</tr>
<tr>
<td>erf 226</td>
</tr>
<tr>
<td>erfc 227</td>
</tr>
<tr>
<td>exp 244</td>
</tr>
<tr>
<td>expm1 247</td>
</tr>
<tr>
<td>fabs 248</td>
</tr>
<tr>
<td>fabsl 248</td>
</tr>
<tr>
<td>fcvt 224</td>
</tr>
<tr>
<td>finite 167</td>
</tr>
<tr>
<td>floor 274</td>
</tr>
<tr>
<td>floorl 274</td>
</tr>
<tr>
<td>fmin 776</td>
</tr>
<tr>
<td>fmod 279</td>
</tr>
<tr>
<td>fmodl 279</td>
</tr>
<tr>
<td>fp_any_enable 290</td>
</tr>
<tr>
<td>fp_any_xcp 296</td>
</tr>
<tr>
<td>fp_clr_flag 292</td>
</tr>
<tr>
<td>fp_disable 290</td>
</tr>
<tr>
<td>fp_disable_all 290</td>
</tr>
<tr>
<td>fp_divbyzero 296</td>
</tr>
<tr>
<td>fp_enable 290</td>
</tr>
<tr>
<td>fp_enable_all 290</td>
</tr>
<tr>
<td>fp_inexact 296</td>
</tr>
<tr>
<td>fp_invalid_op 296</td>
</tr>
<tr>
<td>fp_iop_convert 297</td>
</tr>
<tr>
<td>fp_iop_infdinf 297</td>
</tr>
<tr>
<td>fp_iop_infmzr 297</td>
</tr>
<tr>
<td>fp_iop_infsinf 297</td>
</tr>
<tr>
<td>fp_iop_invcmp 297</td>
</tr>
<tr>
<td>fp_iop_snan 297</td>
</tr>
<tr>
<td>fp_iop_sqrt 297</td>
</tr>
<tr>
<td>fp_iop_zrdmr 297</td>
</tr>
<tr>
<td>fp_is_enabled 290</td>
</tr>
<tr>
<td>fp_overflow 296</td>
</tr>
<tr>
<td>fp_read_flag 292</td>
</tr>
<tr>
<td>fp_read_mrd 299</td>
</tr>
<tr>
<td>fp_set_flag 292</td>
</tr>
<tr>
<td>fp_swap_flag 292</td>
</tr>
<tr>
<td>fp_swap_mrd 299</td>
</tr>
<tr>
<td>fp_underflow 296</td>
</tr>
<tr>
<td>frexp 311</td>
</tr>
<tr>
<td>frexpl 311</td>
</tr>
<tr>
<td>gamma 332</td>
</tr>
<tr>
<td>gcd 776</td>
</tr>
<tr>
<td>gcvt 224</td>
</tr>
<tr>
<td>hypot 523</td>
</tr>
<tr>
<td>ilogb 529</td>
</tr>
<tr>
<td>imul_dbl 3</td>
</tr>
<tr>
<td>invert 776</td>
</tr>
<tr>
<td>isnan 167</td>
</tr>
<tr>
<td>itom 776</td>
</tr>
<tr>
<td>itrunc 274</td>
</tr>
<tr>
<td>j0 119</td>
</tr>
<tr>
<td>j1 119</td>
</tr>
<tr>
<td>jn 119</td>
</tr>
<tr>
<td>jrand48 220</td>
</tr>
<tr>
<td>l3tol 581</td>
</tr>
<tr>
<td>l64a 1</td>
</tr>
<tr>
<td>labs 3</td>
</tr>
<tr>
<td>lcong48 220</td>
</tr>
<tr>
<td>ldexp 693</td>
</tr>
<tr>
<td>ldexpl 693</td>
</tr>
<tr>
<td>lddiv 3</td>
</tr>
<tr>
<td>llabs 3</td>
</tr>
<tr>
<td>lldiv 3</td>
</tr>
<tr>
<td>log 740</td>
</tr>
<tr>
<td>log10 736</td>
</tr>
<tr>
<td>log1p 737</td>
</tr>
<tr>
<td>logb 739</td>
</tr>
<tr>
<td>lrand48 220</td>
</tr>
<tr>
<td>ltol3 581</td>
</tr>
<tr>
<td>m_in 776</td>
</tr>
<tr>
<td>m_out 776</td>
</tr>
<tr>
<td>madd 776</td>
</tr>
<tr>
<td>matherr 780</td>
</tr>
<tr>
<td>mcmpl 797</td>
</tr>
<tr>
<td>mdiv 776</td>
</tr>
<tr>
<td>min 776</td>
</tr>
<tr>
<td>modf 839</td>
</tr>
<tr>
<td>modfl 839</td>
</tr>
<tr>
<td>mout 776</td>
</tr>
<tr>
<td>move 776</td>
</tr>
<tr>
<td>mrand48 220</td>
</tr>
<tr>
<td>msqrt 776</td>
</tr>
<tr>
<td>msqrt 776</td>
</tr>
<tr>
<td>mult 776</td>
</tr>
<tr>
<td>nearest 274</td>
</tr>
<tr>
<td>nextafter 889</td>
</tr>
<tr>
<td>nnrand48 220</td>
</tr>
<tr>
<td>omin 776</td>
</tr>
<tr>
<td>omout 776</td>
</tr>
<tr>
<td>pow 776, 1146</td>
</tr>
<tr>
<td>rpow 776</td>
</tr>
<tr>
<td>sdiv 776</td>
</tr>
<tr>
<td>seed48 220</td>
</tr>
<tr>
<td>srand48 220</td>
</tr>
<tr>
<td>trunc 274</td>
</tr>
<tr>
<td>uintrunc 274</td>
</tr>
<tr>
<td>umul_dbl 3</td>
</tr>
<tr>
<td>unordered 167</td>
</tr>
</tbody>
</table>
numerical manipulation subroutines (continued)
  y0 119
  y1 119
  yn 119

O

Object Data Manager 913

Object file access subroutines
  ldaclose 692
  ldahread 691
  ldaopen 701
  ldclose 692
  ldftime 694
  ldgetname 696
  ldinit 698
  ldltf 698
  ldltread 698
  ldlsseek 699
  ldlnseek 703
  ldlnseek 704
  ldlnseek 706
  ldlnseek 700
  ldopen 701
  ldseek 703
  ldshread 704
  ldssseek 706
  ldtsbindex 707
  ldtsbread 708
  ldtsseek 709

Object file subroutines
  load 721
  loadbind 725
  loadquery 726

Object files
  closing 692
  computing symbol table entries 707
  controlling run-time resolution 725
  listing 726
  loading and binding 721
  manipulating line number entries 698
  providing access 701
  reading file headers 694
  reading indexed section headers 704
  reading symbol table entries 708
  retrieving symbol names 696
  seeking to indexed sections 706
  seeking to line number entries 699
  seeking to optional file header 700
  seeking to relocation entries 703
  seeking to symbol tables 709

Objects
  setting locale-dependent conventions 728

ODM
  ending session 923
  error message strings 906
  freeing memory 907

ODM (Object Data Manager)
  initializing 913
  running specified method 921

ODM object classes
  adding objects 901
  changing objects 903
  closing 904
  creating 905
  locking 913
  opening 916
  removing 918
  removing objects 917, 919
  retrieving class symbol structures 915
  retrieving objects 908, 909, 911
  setting default path location 922
  setting default permissions 923
  unlocking 924

ODM subroutines
  odm_add_obj 901
  odm_change_obj 903
  odm_close_class 904
  odm_create_class 905
  odm_err_msg 906
  odm_free_list 907
  odm_get_by_id 908
  odm_get_first 911
  odm_get_list 909
  odm_get_next 911
  odm_get_obj 911
  odm_initialize 913
  odm_lock 913
  odm_mount_class 915
  odm_open_class 916
  odm_open_class_rdonly 916
  odm_rm_by_id 917
  odm_rm_class 918
  odm_rm_obj 919
  odm_run_method 921
  odm_set_path 922
  odm_set_perms 923
  odm_terminate 923
  odm_unlock 924

odm_add_obj subroutine 901
odm_change_obj subroutine 903
odm_close_class subroutine 904
odm_create_class subroutine 905
odm_err_msg subroutine 906
odm_free_list subroutine 907
odm_get_by_id subroutine 908
odm_get_first subroutine 911
odm_get_list subroutine 909
odm_get_next subroutine 911
odm_get_obj subroutine 911
odm_initialize subroutine 913
odm_lock subroutine 913
odm_mount_class subroutine 915
odm_open_class subroutine 916
odm_open_class_rdonly subroutine 916
odm_rm_by_id subroutine 917
odm_rm_class subroutine 918
odm_rm_obj subroutine 919
odm_run_method subroutine 921
PAG Services

genpagvalue 335

Paging memory

behavior 778

defining available space 1161

PAM subroutines

pam_acct_mgmt 936
pam_authenticate 937
pam_chauthtok 939
pam_close_session 940
pam_end 941
pam_get_data 942
pam_get_item 943
pam_get_user 944
pam_getenv 945
pam_getenvlist 946
pam_open_session 947
pam_putenv 948
pam_set_data 949
pam_set_item 950
pam_setcred 951
pam_sm_acct_mgmt 953
pam_sm_authenticate 954
pam_sm_chauthtok 955
pam_sm_close_session 957
pam_sm_open_session 958
pam_sm_setcred 959
pam_start 960
pam_strerror 963

Password subroutines

passwdpolicy 966
passwdstrength 968

Passwords

generating new 891
reading 397

Pathconf subroutine 969
pause subroutine 972
pcap_close 972
pcap_compile 973
pcap_datalink 973
pcap_dispatch 974
pcap_dump 975
pcap_dump_close 976
pcap_dump_open 977
pcap_file 978
pcap_fileno 978
pcap_geterr 979
pcap_is_swapped 980
pcap_lookupdev 980
pcap_lookupnet 981
pcap_loop 982
pcap_major_version 983
pcap_next 984
pcap_open_live 985
pcap_open_offline 986
pcap_perror 987
pcap_setfilter 988
pcap_snapshot 989
pcap_stats 989
pcap_strerror 990
pclose subroutine 991

Performance monitor subroutines

pm_delete_program_pgroup 1021
pm_delete_program_pthread 1022
pm_get_data_pgroup 1039
pm_get_data_pgroup_mx 1041
pm_get_data_pthread 1043
pm_get_data_pthread_mx 1044
pm_get_program_pgroup 1060
pm_get_program_pgroup_mx 1061
pm_get_program_pthread 1063
pm_get_program_pthread_mx 1064
performance monitor subroutines (continued)

<table>
<thead>
<tr>
<th>Subroutine</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>pm_get_tdata_pgroup</td>
<td>1039</td>
</tr>
<tr>
<td>pm_get_Tdata_pgroup</td>
<td>1039</td>
</tr>
<tr>
<td>pm_get_tdata_pgroup_mx</td>
<td>1041</td>
</tr>
<tr>
<td>pm_get_tdata_pthread</td>
<td>1043</td>
</tr>
<tr>
<td>pm_get_Tdata_pthread</td>
<td>1043</td>
</tr>
<tr>
<td>pm_get_tdata_pthread_mx</td>
<td>1044</td>
</tr>
<tr>
<td>pm_initialize</td>
<td>1071</td>
</tr>
<tr>
<td>pm_reset_data_pgroup</td>
<td>1076</td>
</tr>
<tr>
<td>pm_reset_data_pthread</td>
<td>1078</td>
</tr>
<tr>
<td>pm_set_program_pgroup</td>
<td>1092</td>
</tr>
<tr>
<td>pm_set_program_pgroup_mx</td>
<td>1093</td>
</tr>
<tr>
<td>pm_set_program_pthread</td>
<td>1095</td>
</tr>
<tr>
<td>pm_set_program_pthread_mx</td>
<td>1097</td>
</tr>
<tr>
<td>pm_start_pgroup</td>
<td>1106</td>
</tr>
<tr>
<td>pm_start_pthread</td>
<td>1107</td>
</tr>
<tr>
<td>pm_stop_pgroup</td>
<td>1114</td>
</tr>
<tr>
<td>pm_tstart_pgroup</td>
<td>1106</td>
</tr>
<tr>
<td>pm_tstart_pthread</td>
<td>1107</td>
</tr>
<tr>
<td>pm_tstop_pgroup</td>
<td>1114</td>
</tr>
</tbody>
</table>

perfstat

<table>
<thead>
<tr>
<th>Subroutine</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>perfstat_partition_total subroutine</td>
<td>1010</td>
</tr>
<tr>
<td>perfstat_cpu subroutine</td>
<td>992</td>
</tr>
<tr>
<td>perfstat_cpu_total subroutine</td>
<td>994</td>
</tr>
<tr>
<td>perfstat_disk subroutine</td>
<td>995</td>
</tr>
<tr>
<td>perfstat_disk_total subroutine</td>
<td>1000</td>
</tr>
<tr>
<td>perfstat_diskadapter subroutine</td>
<td>997</td>
</tr>
<tr>
<td>perfstat_diskpath subroutine</td>
<td>998</td>
</tr>
<tr>
<td>perfstat_memory_total subroutine</td>
<td>1002</td>
</tr>
<tr>
<td>perfstat_netbuffer subroutine</td>
<td>1003</td>
</tr>
<tr>
<td>perfstat_netinterface subroutine</td>
<td>1004</td>
</tr>
<tr>
<td>perfstat_netinterface_total subroutine</td>
<td>1006</td>
</tr>
<tr>
<td>perfstat_pagingspace subroutine</td>
<td>1007</td>
</tr>
<tr>
<td>perfstat_partial_reset subroutine</td>
<td>1008</td>
</tr>
<tr>
<td>perfstat_partition_total subroutine</td>
<td>1010</td>
</tr>
<tr>
<td>perfstat_protocol subroutine</td>
<td>1011</td>
</tr>
<tr>
<td>perfstat_reset subroutine</td>
<td>1013</td>
</tr>
</tbody>
</table>

permanent storage

writing file changes to 317
perror subroutine 1013
pglob parameter
freeing memory 480
physical volumes
querying 761
pipe subroutine 1014
pipes
  closing 991
  creating 1014, 1124
plock subroutine 1015

POSIX SPAWN subroutines

<table>
<thead>
<tr>
<th>Subroutine</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>posix_spawn</td>
<td>1129</td>
</tr>
<tr>
<td>posix_spawn_file_actions_addclose</td>
<td>1133</td>
</tr>
<tr>
<td>posix_spawn_file_actions_adddup2</td>
<td>1134</td>
</tr>
<tr>
<td>posix_spawn_file_actions_addopen</td>
<td>1133</td>
</tr>
<tr>
<td>posix_spawn_file_actions_destroy</td>
<td>1135</td>
</tr>
<tr>
<td>posix_spawn_file_actions_init</td>
<td>1135</td>
</tr>
<tr>
<td>posix_spawnattr_destroy</td>
<td>1136</td>
</tr>
<tr>
<td>posix_spawnattr_getflags</td>
<td>1137</td>
</tr>
<tr>
<td>posix_spawnattr_getgroupparam</td>
<td>1138</td>
</tr>
<tr>
<td>posix_spawnattr_getgrouppolicy</td>
<td>1139</td>
</tr>
<tr>
<td>posix_spawnattr_getsigdefault</td>
<td>1141</td>
</tr>
<tr>
<td>posix_spawnattr_getsigmask</td>
<td>1142</td>
</tr>
<tr>
<td>posix_spawnattr_init</td>
<td>1136</td>
</tr>
<tr>
<td>posix_spawnattr_setflags</td>
<td>1137</td>
</tr>
<tr>
<td>posix_spawnattr_setgroup</td>
<td>1138</td>
</tr>
<tr>
<td>posix_spawnattr_setsiglendparam</td>
<td>1139</td>
</tr>
<tr>
<td>posix_spawnattr_setsiglendpolicy</td>
<td>1140</td>
</tr>
<tr>
<td>posix_spawnattr_setsigdefault</td>
<td>1141</td>
</tr>
<tr>
<td>posix_spawnattr_setsigmask</td>
<td>1142</td>
</tr>
<tr>
<td>posix_spawnp</td>
<td>1129</td>
</tr>
<tr>
<td>posix_openpt subroutine</td>
<td>1128</td>
</tr>
</tbody>
</table>

peristate

<table>
<thead>
<tr>
<th>Subroutine</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>perfstat_partition_total subroutine</td>
<td>1010</td>
</tr>
<tr>
<td>perfstat_cpu subroutine</td>
<td>992</td>
</tr>
<tr>
<td>perfstat_cpu_total subroutine</td>
<td>994</td>
</tr>
<tr>
<td>perfstat_disk subroutine</td>
<td>995</td>
</tr>
<tr>
<td>perfstat_disk_total subroutine</td>
<td>1000</td>
</tr>
<tr>
<td>perfstat_diskadapter subroutine</td>
<td>997</td>
</tr>
<tr>
<td>perfstat_diskpath subroutine</td>
<td>998</td>
</tr>
<tr>
<td>perfstat_memory_total subroutine</td>
<td>1002</td>
</tr>
<tr>
<td>perfstat_netbuffer subroutine</td>
<td>1003</td>
</tr>
<tr>
<td>perfstat_netinterface subroutine</td>
<td>1004</td>
</tr>
<tr>
<td>perfstat_netinterface_total subroutine</td>
<td>1006</td>
</tr>
<tr>
<td>perfstat_pagingspace subroutine</td>
<td>1007</td>
</tr>
<tr>
<td>perfstat_partial_reset subroutine</td>
<td>1008</td>
</tr>
<tr>
<td>perfstat_partition_total subroutine</td>
<td>1010</td>
</tr>
<tr>
<td>perfstat_protocol subroutine</td>
<td>1011</td>
</tr>
<tr>
<td>perfstat_reset subroutine</td>
<td>1013</td>
</tr>
</tbody>
</table>

POSIX Realtime subroutines

<table>
<thead>
<tr>
<th>Subroutine</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>posix_fadvise</td>
<td>1125</td>
</tr>
<tr>
<td>posix_fallocate</td>
<td>1126</td>
</tr>
<tr>
<td>posix_madvise</td>
<td>1127</td>
</tr>
<tr>
<td>poll subroutine</td>
<td>1118</td>
</tr>
<tr>
<td>pollset_create subroutine</td>
<td>1121</td>
</tr>
<tr>
<td>pollset_ctl</td>
<td>1121</td>
</tr>
<tr>
<td>pollset_destroy</td>
<td>1121</td>
</tr>
<tr>
<td>pollset_poll</td>
<td>1121</td>
</tr>
<tr>
<td>pollset_query</td>
<td>1121</td>
</tr>
<tr>
<td>pollset_create subroutine</td>
<td>1121</td>
</tr>
<tr>
<td>pollset_ctl subroutine</td>
<td>1121</td>
</tr>
<tr>
<td>pollset_destroy subroutine</td>
<td>1121</td>
</tr>
<tr>
<td>pollset_poll subroutine</td>
<td>1121</td>
</tr>
<tr>
<td>pollset_query subroutine</td>
<td>1121</td>
</tr>
<tr>
<td>popen subroutine</td>
<td>1124</td>
</tr>
<tr>
<td>posix_openpt Subroutine</td>
<td>1128</td>
</tr>
<tr>
<td>posix_spawn subroutine</td>
<td>1129</td>
</tr>
<tr>
<td>posix_spawn_file_actions_addclose</td>
<td>1133</td>
</tr>
<tr>
<td>posix_spawn_file_actions_adddup2</td>
<td>1134</td>
</tr>
<tr>
<td>posix_spawn_file_actions_addopen</td>
<td>1133</td>
</tr>
<tr>
<td>posix_spawn_file_actions_destroy</td>
<td>1135</td>
</tr>
<tr>
<td>posix_spawn_file_actions_init</td>
<td>1135</td>
</tr>
<tr>
<td>posix_spawnattr_destroy</td>
<td>1136</td>
</tr>
<tr>
<td>posix_spawnattr_getflags</td>
<td>1137</td>
</tr>
<tr>
<td>posix_spawnattr_getgroupparam</td>
<td>1138</td>
</tr>
<tr>
<td>posix_spawnattr_getgrouppolicy</td>
<td>1139</td>
</tr>
<tr>
<td>posix_spawnattr_getsigdefault</td>
<td>1141</td>
</tr>
<tr>
<td>posix_spawnattr_getsigmask</td>
<td>1142</td>
</tr>
<tr>
<td>posix_spawnattr_init</td>
<td>1136</td>
</tr>
<tr>
<td>posix_spawnattr_setflags</td>
<td>1137</td>
</tr>
<tr>
<td>posix_spawnattr_setgroup</td>
<td>1138</td>
</tr>
<tr>
<td>posix_spawnattr_setsiglendparam</td>
<td>1139</td>
</tr>
<tr>
<td>posix_spawnattr_setsiglendpolicy</td>
<td>1140</td>
</tr>
<tr>
<td>posix_spawnattr_setsigdefault</td>
<td>1141</td>
</tr>
<tr>
<td>posix_spawnattr_setsigmask</td>
<td>1142</td>
</tr>
<tr>
<td>posix_spawnp</td>
<td>1129</td>
</tr>
</tbody>
</table>
processes subroutines (continued)

getrlimit64 419
getusage 423
getusage64 423
kill 575
killpg 575
msgctl 870
msgget 872
msgrcv 873
msgsnd 876
msgxrcv 878
nice 407
pause 972
plock 1015
profil 1155
psignal 1162
ptrace 1286
sbrk 122
setpriority 407
setrlimit 419
setrlimit64 419
times 423
unatexit 242
vfork 287
vlimit 419
vtimes 423

profil subroutine 1155
program assertion
  verifying 92
  proj_execve subroutine 1157
  projdballoc subroutine 1158
  projdbfinit subroutine 1159
  projdbfree subroutine 1160
  psdanger subroutine 1161
  psignal subroutine 1162

pthread subroutines
  pthread_attr_getinheritsched subroutine 1193
  pthread_attr_getschedparam subroutine 1194
  pthread_attr_getschedpolicy subroutine 1194
  pthread_attr_getsockopt subroutine 1199
  pthread_attr_getstackaddr subroutine 1195
  pthread_attr_getstacksize subroutine 1196
  pthread_attr_init subroutine 1197
  pthread_attr_setdetachstate subroutine 1198
  pthread_attr_setsigguardsize subroutine 1199
  pthread_attr_setsiginheritsched subroutine 1192
  pthread_attr_setsigpolicy subroutine 1201
  pthread_attr_setsigpolicy subroutine 1201
  pthread_attr_setsigscope subroutine 1199
  pthread_attr_setstackaddr subroutine 1202
  pthread_attr_setstacksize subroutine 1203
  pthread_attr_setsuspendstate_np and
    pthread_attr_getsuspendstate_np subroutine 1204
  pthread_cancel subroutine 1209
  pthread_cleanup_pop subroutine 1211
  pthread_cleanup_push subroutine 1211
  pthread_cond_broadcast subroutine 1214
  pthread_cond_destroy subroutine 1212
  PTHREAD_COND_INITIALIZER macro 1213
  pthread_cond_signal subroutine 1214
  pthread_cond_timedwait subroutine 1215
  pthread_cond_timedwait subroutine 1215
  pthread_cond_setstate subroutine 1217
  pthread_condattr_destroy subroutine 1217
  pthread_condattr_getclock subroutine 1218
  pthread_condattr_getpshared subroutine 1219
  pthread_condattr_init subroutine 1218
  pthread_condattr_setclock subroutine 1218
  pthread_condattr_setpshared subroutine 1221
  pthread_create subroutine 1222
  pthread_create subroutine 1222
  pthread_create_withcred_np subroutine 1224
  pthread_create_withcred_np subroutine 1224
  pthread_create_withcred_np subroutine 1225
  pthread_equal subroutine 1226
  pthread_exit subroutine 1227
  pthread_getattr_np subroutine 1228
  pthread_getcpclockid subroutine 1229
  pthread_getpriority_np subroutine 1231
  pthread_getschedparam subroutine 1231
  pthread_getspecific subroutine 1234
  pthread_join subroutine 1235
  pthread_join subroutine 1240
  pthread_key_create subroutine 1240
  pthread_key_create subroutine 1240
  pthread_key_delete subroutine 1243
  pthread_key_delete subroutine 1243
  pthread_key_delete subroutine 1244
  pthread_key_delete subroutine 1244
  pthread_mutex_destroy subroutine 1245
  pthread_mutex_destroy subroutine 1245
  pthread_mutex_init subroutine 1246
  pthread_mutex_init subroutine 1246
  PTHREAD_MUTEX_INITIALIZER macro 1249
  pthread_mutex_destroy subroutine 1249
  pthread_mutex_destroy subroutine 1249
  pthread_mutex_destroy subroutine 1251
  pthread_mutex_init subroutine 1249
  pthread_mutex_init subroutine 1249
  pthread_mutexattr_destroy subroutine 1252
  pthread_mutexattr_destroy subroutine 1252
  pthread_mutexattr_getkind_np subroutine 1254
  pthread_mutexattr_getkind_np subroutine 1254
  pthread_mutexattr_setkind_np subroutine 1259
  pthread_mutexattr_setkind_np subroutine 1259
  pthread_mutexattr_init subroutine 1252
  pthread_mutexattr_init subroutine 1252
  pthread_mutexattr_setkind_np subroutine 1260
  pthread_mutexattr_setkind_np subroutine 1260
  pthread_mutexattr_settype subroutine 1259
  pthread_mutexattr_settype subroutine 1259
  pthread_mutexattr_settype subroutine 1262
  PTHREAD_ONCE_INIT macro 1263

1352  Technical Reference, Volume 1: Base Operating System and Extensions
Q

queues
inserting and removing elements 554
quotient and remainder
imaxdiv 530

R

radix-independent exponents
logbf 739
radix-independent exponents (continued)
logbl 739
read operations
asynchronous 50
binary files 307
read-write file pointers
moving 756
readdir subroutine 933
readdir64 subroutine 933
real floating types
fpclassify 306
real value subroutines
creal 190
crealf 190
creall 190
realloc subroutine 769
regular expressions
matching patterns 177
remqueue subroutine 554
resabs subroutine 382
reset_speed subroutine 338
resinc subroutine 382
resource information 1231
resources subroutines
pthread_getrusage_np 1231
restimer subroutine 441
rewind subroutine 314
rewinddir subroutine 933
rewinddir64 subroutine 933
rounding direction
fegetround 264
gesetround 264
rounding numbers
llrint 719
llrintf 719
llrintl 719
llround 720
llroundf 720
llroundl 720
lrint 752
lrintf 752
lrintl 752
lround 753
lroundf 753
lroundl 753
rpc file
handling 422
rpow subroutine 776
run-time environment
initializing 576

S

sbrk subroutine 122
sdiv subroutine 776
security library subroutines
authenticatex 115
chpassx 156
getconfattrs 350
getgroupattrs 373
gouserattrs 455
security library subroutines (continued)
  getuserpw 465
  loginrestrictions 746
  newpass 893
  passwdexpired 964
  putconfattr 1302
  putgroupsattr 1306
  putuserattr 1311
  putuserpw 1315

security subroutines
  getuinfo 448
  seed48 subroutine 220
  seekdir subroutine 933
  seekdir64 subroutine 933
  set_speed subroutine 338
  setfsent subroutine 364
  setfsent_r subroutine 430
  setgrent subroutine 366
  setitimer subroutine 382
  setkey subroutine 191
  setpriority subroutine 407
  setpwnent subroutine 417
  setrlimit subroutine 419
  setrlimit64 subroutine 419
  setrcents subroutine 422
  setsockopt subroutine 524
  settimeofday subroutine 440
  setutent subroutine 469
  setvsent subroutine 471
  shell command-line flags 392
  SIGALRM signal 383
  SIGIOIT signal 2
  signal names
    formatting 1162
  sine functions
    csin 193
    csinf 193
    csinl 193
  single-byte to wide-character conversion 124
  SJIS character conversions 570
  stojis subroutine 570
  stouj subroutine 570
  snprintf subroutine 1148
  socket options
    setting 524
  sockets kernel service subroutines
    setssockopt 524
  sockets network library subroutines
    endhostent 901
    gethostent 900
    inet_aton 552
  special files
    creating 828
  sprintf subroutine 1148
  square root subroutines
    csqrt 194
    csqrf 194
    csqrtl 194
  srand48 subroutine 220

SRC subroutines
  addsys 33
  chssys 162
  delssys 211
  getsys 432
  SRC subsys record
    adding 33
  SRC subsys structure
    initializing 211
  Statistics subroutines
    perfstat_cpu 992
    perfstat_cpu_total 994
    perfstat_disk 995
    perfstat_disk_total 1000
    perfstat_diskadapter 997
    perfstat_diskpath 998
    perfstat_memory_total 1002
    perfstat_netbuffer 1003
    perfstat_netinterface 1004
    perfstat_netadapter 1006
    perfstat_pagingspace 1007
    perfstat_protocol 1011
    perfstat_reset 1013
  status indicators
    beeping 534
    drawing 538
    hiding 539
  step subroutine 177
  stime subroutine 441
  streams
    checking status 267
    closing 252
    flushing 252
    opening 284
    repositioning file pointers 314
    writing to 252
  string conversion
    long integers to base-64 ASCII 1
  string manipulation subroutines
    advance 177
    bcmp 118
    bcopy 118
    bzero 118
    compile 177
    ffs 118
    fgets 429
    fnmatch 282
    fputs 1309
    gets 429
    puts 1309
    step 177
  strings
    bit string operations 118
    byte string operations 118
    copying 118
    drawing text strings 550
    matching against pattern parameters 282
    reading bytes into arrays 429
    writing to standard output streams 1309
    zeroing out 118
subroutine
pcap_close 972
pcap_compile 973
pcap_datalink 973
pcap_dispatch 974
pcap_dump 975
pcap_dump_close 976
pcap_dump_open 977
pcap_file 978
pcap_fileno 978
pcap_geterr 979
pcap_is_swapped 980
pcap_lookupdev 980
pcap_lookupnet 981
pcap_loop 982
pcap_major_version 983
pcap_next 984
pcap_open_live 985
pcap_open_offline 986
pcap_perror 987
pcap_setfilter 988
pcap_snapshot 989
pcap_stats 989
pcap_strerror 990

Subroutines
LAPI_Addr_get 583
LAPI_Addr_set 584
LAPI_Address 586
LAPI_Address_init 587
LAPI_Address_init64 589
LAPI_Amsend 591
LAPI_Amsendv 596
LAPI_Fence 604
LAPI_Get 605
LAPI_Getcntr 608
LAPI_Gfence 613
LAPI_Init 614
LAPI_Msg_string 619
LAPI_Msgpoll 621
LAPI_Nopoll_wait 623
LAPI_PROBE 624
LAPI_Purge_totask 625
LAPI_Put 627
LAPI_Putv 629
LAPI_Qenv 633
LAPI_Resume_totask 636
LAPI_Rmw 637
LAPI_Rmw64 641
LAPI_Serv 645
LAPI_Setcntr 647
LAPI_Setcntr_wstatus 649
LAPI_Term 650
LAPI_Util 652
LAPI_Waitcntr 663
LAPI_Xfer 665

Subroutines (continued)
perfsstat_cpu 992
perfsstat_cpu_total 994
perfsstat_disk_total 995, 1000
perfsstat_diskpath 998

subsystem objects
modifying 162
removing 211

subsystem records
reading 432, 434

supplementary process group IDs
getting 378
initializing 553

swapcontext Subroutine 779

swprintf subroutine 323

symbols
translating names to addresses 577

sys_siglist vector 1162
SYSP_V_IOSTRUN
sys_parm 995, 997, 998

system auditing 98

system data objects
auditing modes 105

system event audits
getting or setting status 102

system resources
setting maximums 419

system signal messages 1162

system variables
determining values 181

T
telldir subroutine 933
telldir64 subroutine 933
terminal baud rate
get 338
set 338
text area
hiding 550
text locks 1015
text strings
drawing 550

Thread-safe C Library
subroutines
164_r 582

Thread-Safe C Library 368, 369, 430
subroutines
getfsent_r 430
getlogin_r 390
getsfile_r 430

threads
getting thread table entries 438

Threads Library 1276

condition variables
creation and destruction 1212, 1213
creation attributes 1217, 1219, 1221
signalling a condition 1214
waiting for a condition 1215
Threads Library (continued)

DCE compatibility subroutines
  pthread_delay_np 1225
  pthread_get_expiration_np 1228
  pthread_getunique_np 1240
  pthread_lock_global_np 1245
  pthread_mutexattr_getkind_np 1254
  pthread_mutexattr_setkind_np 1260
  pthread_signal_to_cancel_np 1279
  pthread_unlock_global_np 1284

mutexes
  creation and destruction 1249
  creation attributes 1259
  locking 1249
  pthread_mutexattr_destroy 1252
  pthread_mutexattr_init 1252

process creation
  pthread_atfork subroutine 1188
  pthread_attr_getguardsize subroutine 1190
  pthread_attr_setguardsize subroutine 1190
  pthread_getconcurrency subroutine 1229
  pthread_mutex_destroy 1246
  pthread_mutex_init 1246

scheduling
  dynamic thread control 1234, 1285
  thread creation attributes 1193, 1201

signal, sleep, and timer handling
  pthread_kill subroutine 1244

thread-specific data
  pthread_getspecific subroutine 1235
  pthread_key_create subroutine 1242
  pthread_key_delete subroutine 1243
  pthread_setspecific subroutine 1235

threads
  cancellation 1209, 1275
  creation 1222
  creation attributes 1189, 1195, 1196, 1197, 1198,
     1199, 1202, 1203, 1204, 1283
  ID handling 1226, 1274
  initialization 1262, 1263
  termination 1211, 1227, 1241

time
  displaying and setting 440
  reporting used CPU time 169
  synchronizing system clocks 35

time format conversions 199

time manipulation subroutines (continued)
  getinterval 382
  gettimeofday 440
U
ualarm subroutine 382
uitrunc subroutine 274
UJIS character conversions 570
ujtojis subroutine 570
ujtosj subroutine 570
umul_dbl subroutine 3
unatexit subroutine 242
unbiased exponents
   ilogbf 529
   ilogbl 529
unordered subroutine 167
user accounts
   checking validity 164
user authentication data
   accessing 463
user database
   accessing group information 366, 370
   accessing user information 346, 417, 449
user information
   accessing 346, 417, 449
   accessing group information 366, 370
   searching buffer 448
user login name
   getting 389
users
   authenticating 166
utmpname subroutine 469

V
vectors
   sys_siglist 1162
vfork subroutine 287
vfprintf subroutine 1148
VFS (Virtual File System)
   getting file entries 471
   returning mount status 838
virtual memory
   mapping file-system objects 834
vlimit subroutine 419
volume groups
   querying 765
   querying all varied on-line 768
vprintf subroutine 323
vsprintf subroutine 1148
vtimes subroutine 423
vwsprintf subroutine 1148

W
wide character subroutines
   (continued)
   iswalnum 567
   iswalpha 567
   iswcntl 567
   iswctype subroutine 569
   iswdigit 567
   iswgraph 567
   iswlower 567
   iswprint 567
   iswпunct 567
   iswspace 567
   iswupper 567
   iswxdigit 567
   putwc 1317
   putwchar 1317
   putws 1319
wide characters
   checking character class 567
   converting
      from multibyte 795, 796
   determining properties 569
   reading from input stream 472, 475
   writing to output stream 1317, 1319
words
   returning from input streams 343
wprintf subroutine 323
write operations
   asynchronous 61
   binary files 307
wscanf subroutine 327
wsprintf subroutine 1148

Y
y0 subroutine 119
y1 subroutine 119
yn subroutine 119
Readers’ Comments — We’d Like to Hear from You

AIX 5L Version 5.3
Technical Reference: Base Operating System and Extensions, Volume 1

Publication No. SC23-4913-03

Overall, how satisfied are you with the information in this book?

<table>
<thead>
<tr>
<th>Overall satisfaction</th>
<th>Very Satisfied</th>
<th>Satisfied</th>
<th>Neutral</th>
<th>Dissatisfied</th>
<th>Very Dissatisfied</th>
</tr>
</thead>
</table>

How satisfied are you that the information in this book is:

<table>
<thead>
<tr>
<th></th>
<th>Very Satisfied</th>
<th>Satisfied</th>
<th>Neutral</th>
<th>Dissatisfied</th>
<th>Very Dissatisfied</th>
</tr>
</thead>
<tbody>
<tr>
<td>Accurate</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Complete</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Easy to find</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Easy to understand</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Well organized</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Applicable to your tasks</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Please tell us how we can improve this book:

Thank you for your responses. May we contact you?  □ Yes  □ No

When you send comments to IBM, you grant IBM a nonexclusive right to use or distribute your comments in any way it believes appropriate without incurring any obligation to you. IBM or any other organizations will only use the personal information that you supply to contact you about the issues that you state on this form.

Name

Address

Company or Organization

Phone No.