Technical Reference: Communications, Volume 2
Technical Reference: Communications, Volume 2
Fourth Edition (July 2006)

This edition applies to AIX 5L Version 5.3 and to all subsequent releases of this product until otherwise indicated in new editions.

A reader's comment form is provided at the back of this publication. If the form has been removed, address comments to Information Development, Department 04XA-905-6C006, 11501 Burnet Road, Austin, Texas 78758-3493. To send comments electronically, use this commercial Internet address: aix6kpub@austin.ibm.com. Any information that you supply may be used without incurring any obligation to you.

© Copyright International Business Machines Corporation 1997, 2006. All rights reserved.

US Government Users Restricted Rights – Use, duplication or disclosure restricted by GSA ADP Schedule Contract with IBM Corp.
## Contents

**About This Book** ........................................... xi
Highlighting .................................................. xi
Case-Sensitivity in AIX ..................................... xi
ISO 9000 ........................................................... xi
32-Bit and 64-Bit Support for the Single UNIX Specification .................................................. xii
Related Publications ........................................... xii

### Chapter 1. Simple Network Management Protocol (SNMP)

- getsmuxEntrybyname or getsmuxEntrybyidentity Subroutine .................................................. 1
- isodetailor Subroutine ........................................ 2
- ll_hdinit, ll_dbinit, ll_log, or ll_log Subroutine ................................................................. 3
- o_number, o_integer, o_string, o_igeneric, o_generic, o_specific, or o_ipaddr Subroutine ............... 5
- oid_cmp, oid_cpy, oid_free, sprintoid, str2oid, ode2oid, oid2ode, oid2ode_aux, prim2oid, or oid2prim Subroutine ................................................................. 7
- oid_extend or oid_normalize Subroutine ................................................................. 9
- readobjects Subroutine ........................................ 10
- s_generic Subroutine ........................................ 11
- smux_close Subroutine ........................................ 12
- smux_error Subroutine ........................................ 13
- smux_free_tree Subroutine ................................... 14
- smux_init Subroutine .......................................... 15
- smux_register Subroutine .................................... 16
- smux_response Subroutine .................................... 17
- smux_simple_open Subroutine ................................ 18
- smux_trap Subroutine .......................................... 20
- smux_wait Subroutine ......................................... 21
- text2inst, name2inst, next2inst, or nextot2inst Subroutine ................................................ 22
- text2oid or text2obj Subroutine ................................ 23

### Chapter 2. Sockets

- _getlong Subroutine ........................................... 25
- _getshort Subroutine .......................................... 25
- _putlong Subroutine ........................................... 26
- _putshort Subroutine .......................................... 27
- accept Subroutine ............................................. 28
- arpresolve_common Subroutine ................................ 29
- arppupdate Subroutine ........................................ 30
- bind Subroutine ................................................ 31
- connect Subroutine ............................................ 33
- CreateloCompletionPort Subroutine ............................................. 36
- dn_comp Subroutine ............................................ 38
- dn_expand Subroutine .......................................... 39
- endhostent Subroutine ......................................... 40
- endhostent_r Subroutine ....................................... 41
- endnetent Subroutine .......................................... 42
- endnetent_r Subroutine ....................................... 43
- endnetgrent_r Subroutine ..................................... 44
- endprotoent Subroutine ....................................... 44
- endprotoent_r Subroutine ..................................... 45
- endservent Subroutine ......................................... 46
- endservent_r Subroutine ...................................... 47
- ether_ntoa, ether_aton, ether_ntohost, ether_hostton, or ether_line Subroutine ......................... 47
- FrcaCacheCreate Subroutine ................................ 49
<table>
<thead>
<tr>
<th>Function name</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>FrcaCacheDelete Subroutine</td>
<td>50</td>
</tr>
<tr>
<td>FrcaCacheLoadFile Subroutine</td>
<td>51</td>
</tr>
<tr>
<td>FrcaCacheUnloadFile Subroutine</td>
<td>53</td>
</tr>
<tr>
<td>FrcaCtrlCreate Subroutine</td>
<td>54</td>
</tr>
<tr>
<td>FrcaCtrlDelete Subroutine</td>
<td>57</td>
</tr>
<tr>
<td>FrcaCtrlLog Subroutine</td>
<td>58</td>
</tr>
<tr>
<td>FrcaCtrlStart Subroutine</td>
<td>59</td>
</tr>
<tr>
<td>FrcaCtrlStop Subroutine</td>
<td>60</td>
</tr>
<tr>
<td>freeaddrinfo Subroutine</td>
<td>61</td>
</tr>
<tr>
<td>getaddrinfo Subroutine</td>
<td>61</td>
</tr>
<tr>
<td>get_auth_method Subroutine</td>
<td>64</td>
</tr>
<tr>
<td>getdomainname Subroutine</td>
<td>65</td>
</tr>
<tr>
<td>gethostbyaddr Subroutine</td>
<td>66</td>
</tr>
<tr>
<td>gethostbyaddr_r Subroutine</td>
<td>67</td>
</tr>
<tr>
<td>gethostbyname Subroutine</td>
<td>68</td>
</tr>
<tr>
<td>gethostbyname_r Subroutine</td>
<td>70</td>
</tr>
<tr>
<td>gethostent Subroutine</td>
<td>72</td>
</tr>
<tr>
<td>gethostent_r Subroutine</td>
<td>72</td>
</tr>
<tr>
<td>gethostname Subroutine</td>
<td>73</td>
</tr>
<tr>
<td>getnameinfo Subroutine</td>
<td>74</td>
</tr>
<tr>
<td>getnetbyaddr Subroutine</td>
<td>75</td>
</tr>
<tr>
<td>getnetbyaddr_r Subroutine</td>
<td>76</td>
</tr>
<tr>
<td>getnetbyname Subroutine</td>
<td>77</td>
</tr>
<tr>
<td>getnetbyname_r Subroutine</td>
<td>78</td>
</tr>
<tr>
<td>getnetent Subroutine</td>
<td>79</td>
</tr>
<tr>
<td>getnetent_r Subroutine</td>
<td>80</td>
</tr>
<tr>
<td>getnetgrent_r Subroutine</td>
<td>81</td>
</tr>
<tr>
<td>getpeername Subroutine</td>
<td>82</td>
</tr>
<tr>
<td>getprotobyname Subroutine</td>
<td>83</td>
</tr>
<tr>
<td>getprotobyname_r Subroutine</td>
<td>85</td>
</tr>
<tr>
<td>getprotobynumber Subroutine</td>
<td>86</td>
</tr>
<tr>
<td>getprotobynumber_r Subroutine</td>
<td>87</td>
</tr>
<tr>
<td>getprotoent Subroutine</td>
<td>88</td>
</tr>
<tr>
<td>getprotoent_r Subroutine</td>
<td>89</td>
</tr>
<tr>
<td>GetQueuedCompletionStatus Subroutine</td>
<td>90</td>
</tr>
<tr>
<td>getservbyname Subroutine</td>
<td>91</td>
</tr>
<tr>
<td>getservbyname_r Subroutine</td>
<td>92</td>
</tr>
<tr>
<td>getservbyport Subroutine</td>
<td>93</td>
</tr>
<tr>
<td>getservbyport_r Subroutine</td>
<td>94</td>
</tr>
<tr>
<td>getservent Subroutine</td>
<td>95</td>
</tr>
<tr>
<td>getservent_r Subroutine</td>
<td>96</td>
</tr>
<tr>
<td>getsockname Subroutine</td>
<td>97</td>
</tr>
<tr>
<td>getsockopt Subroutine</td>
<td>98</td>
</tr>
<tr>
<td>GetQueuedCompletionStatus Subroutine</td>
<td>99</td>
</tr>
<tr>
<td>getsockname Subroutine</td>
<td>100</td>
</tr>
<tr>
<td>htol Subroutine</td>
<td>108</td>
</tr>
<tr>
<td>htonl Subroutine</td>
<td>108</td>
</tr>
<tr>
<td>htons Subroutine</td>
<td>109</td>
</tr>
<tr>
<td>if_freenameindex Subroutine</td>
<td>110</td>
</tr>
<tr>
<td>if_indextoname Subroutine</td>
<td>111</td>
</tr>
<tr>
<td>if_nameindex Subroutine</td>
<td>112</td>
</tr>
<tr>
<td>if_nametoindex Subroutine</td>
<td>112</td>
</tr>
<tr>
<td>inet6_opt_append Subroutine</td>
<td>113</td>
</tr>
<tr>
<td>inet6_opt_find Subroutine</td>
<td>114</td>
</tr>
<tr>
<td>inet6_opt_finish Subroutine</td>
<td>115</td>
</tr>
<tr>
<td>inet6_opt_get_val Subroutine</td>
<td>115</td>
</tr>
<tr>
<td>inet6_opt_init Subroutine</td>
<td>116</td>
</tr>
</tbody>
</table>
sethostname Subroutine .................................................. 198
setnetent Subroutine .................................................... 199
setnetent_r Subroutine ................................................... 200
setnetgrent_r Subroutine ................................................ 200
setprotoent Subroutine .................................................. 201
setprotoent_r Subroutine ................................................ 202
setservent Subroutine .................................................... 203
setservent_r Subroutine .................................................. 204
setsockopt Subroutine .................................................... 205
shutdown Subroutine ..................................................... 213
SLPAttrCallback Subroutine ............................................. 214
SLPClose Subroutine ..................................................... 215
SLEscape Subroutine ..................................................... 216
SLPFindAttns Subroutine ................................................ 217
SLPFindScopes Subroutine ............................................... 218
SLPFindSrvs Subroutine .................................................. 218
SLPFindSrvTypes Subroutine .......................................... 219
SLPFree Subroutine ....................................................... 220
SLGetProperty Subroutine ................................................. 221
SLPOpen Subroutine ........................................................ 222
SLPParseSrvURL Subroutine .............................................. 223
SLPSrvTypeCallback Subroutine ....................................... 223
SLPSrvURLCallback Subroutine ....................................... 224
SLPUnescape Subroutine .................................................. 225
socket Subroutine .......................................................... 226
socketpair Subroutine .................................................... 228
socks5_getserv Subroutine .............................................. 229
/etc/socks5c.conf File ....................................................... 231
socks5tcp_accept Subroutine .......................................... 232
socks5tcp_bind Subroutine .............................................. 234
socks5tcp_connect Subroutine ......................................... 236
socks5udp_associate Subroutine ...................................... 238
socks5udp_sendto Subroutine .......................................... 240
splice Subroutine .......................................................... 242
WriteFile Subroutine ...................................................... 243

Chapter 3. Streams ............................................................. 245
adjmsg Utility ............................................................... 245
allocb Utility ............................................................... 245
backq Utility ............................................................... 246
bcanput Utility ............................................................. 247
bufcall Utility .............................................................. 247
canput Utility .............................................................. 249
clone Device Driver ......................................................... 249
copyb Utility ............................................................... 250
copymsg Utility ............................................................. 250
datamsg Utility ............................................................. 251
dipl STREAMS Driver ...................................................... 252
dupb Utility ................................................................. 253
dupmsg Utility ............................................................... 253
enableok Utility ............................................................ 254
esballoc Utility ............................................................ 255
flushband Utility ........................................................... 256
flushq Utility ............................................................... 256
freeb Utility ................................................................. 257
freemsg Utility ............................................................. 258
<table>
<thead>
<tr>
<th>Utility/System Call</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>getadmin Utility</td>
<td>258</td>
</tr>
<tr>
<td>getmid Utility</td>
<td>259</td>
</tr>
<tr>
<td>getmsg System Call</td>
<td>259</td>
</tr>
<tr>
<td>getpmsg System Call</td>
<td>262</td>
</tr>
<tr>
<td>getq Utility</td>
<td>263</td>
</tr>
<tr>
<td>insq Utility</td>
<td>264</td>
</tr>
<tr>
<td>ioctl Streams Device Driver Operations</td>
<td>265</td>
</tr>
<tr>
<td>I_ATMARK streamio Operation</td>
<td>276</td>
</tr>
<tr>
<td>I_CANPUT streamio Operation</td>
<td>276</td>
</tr>
<tr>
<td>I_CKBAND streamio Operation</td>
<td>277</td>
</tr>
<tr>
<td>I_FDINSERT streamio Operation</td>
<td>277</td>
</tr>
<tr>
<td>I_FIND streamio Operation</td>
<td>278</td>
</tr>
<tr>
<td>I_FLUSH streamio Operation</td>
<td>279</td>
</tr>
<tr>
<td>I_FLUSHBAND streamio Operation</td>
<td>279</td>
</tr>
<tr>
<td>I_GETBAND streamio Operation</td>
<td>280</td>
</tr>
<tr>
<td>I_GETCLTIME streamio Operation</td>
<td>280</td>
</tr>
<tr>
<td>I_GETSIG streamio Operation</td>
<td>281</td>
</tr>
<tr>
<td>I_GRDOPT streamio Operation</td>
<td>281</td>
</tr>
<tr>
<td>I_LINK streamio Operation</td>
<td>281</td>
</tr>
<tr>
<td>I_LIST streamio Operation</td>
<td>282</td>
</tr>
<tr>
<td>I_LOOK streamio Operation</td>
<td>283</td>
</tr>
<tr>
<td>I_NREAD streamio Operation</td>
<td>283</td>
</tr>
<tr>
<td>I_PEEK streamio Operation</td>
<td>284</td>
</tr>
<tr>
<td>I_PLINK streamio Operation</td>
<td>284</td>
</tr>
<tr>
<td>I_POP streamio Operation</td>
<td>285</td>
</tr>
<tr>
<td>I_PUNLINK streamio Operation</td>
<td>286</td>
</tr>
<tr>
<td>I_PUSH streamio Operation</td>
<td>286</td>
</tr>
<tr>
<td>I_RECVFD streamio Operation</td>
<td>287</td>
</tr>
<tr>
<td>I_SENDFD streamio Operation</td>
<td>288</td>
</tr>
<tr>
<td>I_SETCLTIME streamio Operation</td>
<td>288</td>
</tr>
<tr>
<td>I_SETSIG streamio Operation</td>
<td>289</td>
</tr>
<tr>
<td>I_SRDOPT streamio Operation</td>
<td>290</td>
</tr>
<tr>
<td>I_STR streamio Operation</td>
<td>291</td>
</tr>
<tr>
<td>I_UNLINK streamio Operation</td>
<td>292</td>
</tr>
<tr>
<td>isastream Function</td>
<td>293</td>
</tr>
<tr>
<td>linkb Utility</td>
<td>293</td>
</tr>
<tr>
<td>mi_bufcall Utility</td>
<td>294</td>
</tr>
<tr>
<td>mi_close_comm Utility</td>
<td>295</td>
</tr>
<tr>
<td>mi_next_ptr Utility</td>
<td>296</td>
</tr>
<tr>
<td>mi_open_comm Utility</td>
<td>297</td>
</tr>
<tr>
<td>msgdszie Utility</td>
<td>298</td>
</tr>
<tr>
<td>noenable Utility</td>
<td>299</td>
</tr>
<tr>
<td>OTHERQ Utility</td>
<td>299</td>
</tr>
<tr>
<td>pfmod Packet Filter Module</td>
<td>300</td>
</tr>
<tr>
<td>pullupmsg Utility</td>
<td>303</td>
</tr>
<tr>
<td>putbq Utility</td>
<td>304</td>
</tr>
<tr>
<td>putctl1 Utility</td>
<td>304</td>
</tr>
<tr>
<td>putctl Utility</td>
<td>305</td>
</tr>
<tr>
<td>putmsg System Call</td>
<td>306</td>
</tr>
<tr>
<td>putnext Utility</td>
<td>308</td>
</tr>
<tr>
<td>putpmsg System Call</td>
<td>308</td>
</tr>
<tr>
<td>putq Utility</td>
<td>309</td>
</tr>
<tr>
<td>qenable Utility</td>
<td>311</td>
</tr>
<tr>
<td>qreply Utility</td>
<td>311</td>
</tr>
<tr>
<td>qsize Utility</td>
<td>312</td>
</tr>
<tr>
<td>RD Utility</td>
<td>312</td>
</tr>
<tr>
<td>Topic</td>
<td>Page</td>
</tr>
<tr>
<td>-------------------------------------------</td>
<td>------</td>
</tr>
<tr>
<td>rmvb Utility</td>
<td>313</td>
</tr>
<tr>
<td>rmvq Utility</td>
<td>313</td>
</tr>
<tr>
<td>sad Device Driver</td>
<td>314</td>
</tr>
<tr>
<td>splstr Utility</td>
<td>317</td>
</tr>
<tr>
<td>splx Utility</td>
<td>317</td>
</tr>
<tr>
<td>srv Utility</td>
<td>317</td>
</tr>
<tr>
<td>str_install Utility</td>
<td>319</td>
</tr>
<tr>
<td>streamio Operations</td>
<td>323</td>
</tr>
<tr>
<td>strlog Utility</td>
<td>325</td>
</tr>
<tr>
<td>strqget Utility</td>
<td>326</td>
</tr>
<tr>
<td>t_accept Subroutine for Transport Layer Interface</td>
<td>327</td>
</tr>
<tr>
<td>t_alloc Subroutine for Transport Layer Interface</td>
<td>329</td>
</tr>
<tr>
<td>t_bind Subroutine for Transport Layer Interface</td>
<td>331</td>
</tr>
<tr>
<td>t_close Subroutine for Transport Layer Interface</td>
<td>333</td>
</tr>
<tr>
<td>t_connect Subroutine for Transport Layer Interface</td>
<td>334</td>
</tr>
<tr>
<td>t_error Subroutine for Transport Layer Interface</td>
<td>336</td>
</tr>
<tr>
<td>t_free Subroutine for Transport Layer Interface</td>
<td>337</td>
</tr>
<tr>
<td>t_getinfo Subroutine for Transport Layer Interface</td>
<td>338</td>
</tr>
<tr>
<td>t_getstate Subroutine for Transport Layer Interface</td>
<td>341</td>
</tr>
<tr>
<td>t_listen Subroutine for Transport Layer Interface</td>
<td>342</td>
</tr>
<tr>
<td>t_look Subroutine for Transport Layer Interface</td>
<td>344</td>
</tr>
<tr>
<td>t_open Subroutine for Transport Layer Interface</td>
<td>345</td>
</tr>
<tr>
<td>t_optmgmt Subroutine for Transport Layer Interface</td>
<td>347</td>
</tr>
<tr>
<td>t_rcv Subroutine for Transport Layer Interface</td>
<td>349</td>
</tr>
<tr>
<td>t_rcvconnect Subroutine for Transport Layer Interface</td>
<td>350</td>
</tr>
<tr>
<td>t_rcvdis Subroutine for Transport Layer Interface</td>
<td>352</td>
</tr>
<tr>
<td>t_rcvrel Subroutine for Transport Layer Interface</td>
<td>354</td>
</tr>
<tr>
<td>t_rcvudata Subroutine for Transport Layer Interface</td>
<td>355</td>
</tr>
<tr>
<td>t_rcvuderr Subroutine for Transport Layer Interface</td>
<td>356</td>
</tr>
<tr>
<td>t_snd Subroutine for Transport Layer Interface</td>
<td>357</td>
</tr>
<tr>
<td>t_snddis Subroutine for Transport Layer Interface</td>
<td>359</td>
</tr>
<tr>
<td>t_sndrel Subroutine for Transport Layer Interface</td>
<td>360</td>
</tr>
<tr>
<td>t_sndudata Subroutine for Transport Layer Interface</td>
<td>361</td>
</tr>
<tr>
<td>t_sync Subroutine for Transport Layer Interface</td>
<td>363</td>
</tr>
<tr>
<td>t_unbind Subroutine for Transport Layer Interface</td>
<td>364</td>
</tr>
<tr>
<td>testb Utility</td>
<td>365</td>
</tr>
<tr>
<td>timeout Utility</td>
<td>366</td>
</tr>
<tr>
<td>timod Module</td>
<td>367</td>
</tr>
<tr>
<td>tirdwr Module</td>
<td>369</td>
</tr>
<tr>
<td>unbufcall Utility</td>
<td>370</td>
</tr>
<tr>
<td>unlinkb Utility</td>
<td>370</td>
</tr>
<tr>
<td>untimeout Utility</td>
<td>371</td>
</tr>
<tr>
<td>unweldq Utility</td>
<td>372</td>
</tr>
<tr>
<td>wantio Utility</td>
<td>373</td>
</tr>
<tr>
<td>wantmsg Utility</td>
<td>374</td>
</tr>
<tr>
<td>weldq Utility</td>
<td>376</td>
</tr>
<tr>
<td>WR Utility</td>
<td>377</td>
</tr>
<tr>
<td>xtiso STREAMS Driver</td>
<td>377</td>
</tr>
<tr>
<td>t_accept Subroutine for X/Open Transport Interface</td>
<td>379</td>
</tr>
<tr>
<td>t_alloc Subroutine for X/Open Transport Interface</td>
<td>381</td>
</tr>
<tr>
<td>t_bind Subroutine for X/Open Transport Interface</td>
<td>383</td>
</tr>
<tr>
<td>t_close Subroutine for X/Open Transport Interface</td>
<td>386</td>
</tr>
<tr>
<td>t_connect Subroutine for X/Open Transport Interface</td>
<td>387</td>
</tr>
<tr>
<td>t_error Subroutine for X/Open Transport Interface</td>
<td>389</td>
</tr>
<tr>
<td>t_free Subroutine for X/Open Transport Interface</td>
<td>390</td>
</tr>
<tr>
<td>t_getinfo Subroutine for X/Open Transport Interface</td>
<td>392</td>
</tr>
<tr>
<td>Subroutine for X/Open Transport Interface</td>
<td>Page</td>
</tr>
<tr>
<td>----------------------------------------</td>
<td>------</td>
</tr>
<tr>
<td>t_getprotaddr</td>
<td>394</td>
</tr>
<tr>
<td>t_getstate</td>
<td>395</td>
</tr>
<tr>
<td>t_listen</td>
<td>396</td>
</tr>
<tr>
<td>t_look</td>
<td>398</td>
</tr>
<tr>
<td>t_open</td>
<td>399</td>
</tr>
<tr>
<td>t_optmgmt</td>
<td>403</td>
</tr>
<tr>
<td>t_rcv</td>
<td>409</td>
</tr>
<tr>
<td>t_rcvconnect</td>
<td>411</td>
</tr>
<tr>
<td>t_rcvdis</td>
<td>413</td>
</tr>
<tr>
<td>t_rcvrel</td>
<td>414</td>
</tr>
<tr>
<td>t_rcvudata</td>
<td>415</td>
</tr>
<tr>
<td>t_rcvuderr</td>
<td>417</td>
</tr>
<tr>
<td>t_snd</td>
<td>418</td>
</tr>
<tr>
<td>t_snddis</td>
<td>420</td>
</tr>
<tr>
<td>t_sndrel</td>
<td>421</td>
</tr>
<tr>
<td>t_sndudata</td>
<td>423</td>
</tr>
<tr>
<td>t_strerror</td>
<td>424</td>
</tr>
<tr>
<td>t_sync</td>
<td>425</td>
</tr>
<tr>
<td>t_unbind</td>
<td>426</td>
</tr>
</tbody>
</table>

**Chapter 4. Packet Capture Library Subroutines**

**Appendix. Notices**

**Index**
About This Book

This book provides experienced C programmers with complete detailed information about Simple Network Management Protocol, sockets, streams, and packet capture library subroutines for the AIX® operating system. 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
Chapter 1. Simple Network Management Protocol (SNMP)

getsmuxEntrybyname or getsmuxEntrybyidentity Subroutine

Purpose
Retrieves SNMP multiplexing (SMUX) peer entries from the /etc/snmpd.peers file or the local snmpd.peers file.

Library
SNMP Library (libsnmp.a)

Syntax
#include <isode/snmp/smux.h>
struct smuxEntry *getsmuxEntrybyname (name)
char *name;
struct smuxEntry *getsmuxEntrybyidentity (identity)
OID identity;

Description
The getsmuxEntrybyname and getsmuxEntrybyidentity subroutines read the snmpd.peers file and retrieve information about the SMUX peer. The sample peers file is /etc/snmpd.peers. However, these subroutines can also retrieve the information from a copy of the file that is kept in the local directory. The snmpd.peers file contains entries for the SMUX peers defined for the network. Each SMUX peer entry should contain:

- The name of the SMUX peer.
- The SMUX peer object identifier.
- An optional password to be used on connection initiation. The default password is a null string.
- The optional priority to register the SMUX peer. The default priority is 0.

The getsmuxEntrybyname subroutine searches the file for the specified name. The getsmuxEntrybyidentity subroutine searches the file for the specified object identifier.

These subroutines are part of the SNMP Application Programming Interface in the TCP/IP facility.

Parameters

name Points to a character string that names the SMUX peer.
identity Specifies the object identifier for a SMUX peer.

Return Values
If either subroutine finds the specified SMUX entry, that subroutine returns a structure containing the entry. Otherwise, a null entry is returned.

Files

/etc/snmpd.peers Contains the SMUX peer definitions for the network.
isodetailor Subroutine

Purpose
Initializes variables for various logging facilities.

Library
ISODE Library (libisode.a)

Syntax
```c
#include <isode/tailor.h>
void isodetailor (char *myname, int wantuser);
```

Description
The ISODE library contains internal logging facilities. Some of the facilities need to have their variables initialized. The isodetailor subroutine sets default or user-defined values for the logging facility variables. The logging facility variables are listed in the `usr/lpp/snmpd/smux/isodetailor` file.

The isodetailor subroutine first reads the `/etc/isodetailor` file. If the wantuser parameter is set to 0, the isodetailor subroutine ignores the myname parameter and reads the `/etc/isodetailor` file. If the wantuser parameter is set to a value greater than 0, the isodetailor subroutine searches the current user's home directory (`$HOME`) and reads a file based on the myname parameter. If the myname parameter is specified, the isodetailor subroutine reads a file with the name in the form `.myname_tailor`. If the myname parameter is null, the isodetailor subroutine reads a file named `.isode_tailor`. The _tailor file contents must be in the following form:

```plaintext
#comment <variable> : <value> # comment
<variable> : <value> # comment
<variable> : <value> # comment
```

The comments are optional. The isodetailor subroutine reads the file and changes the values. The latest entry encountered is the final value. The subroutine reads `/etc/isodetailor` first and then the $HOME directory, if told to do so. A complete list of the variables is in the `/usr/lpp/snmpd/smux/isodetailor` sample file.

This subroutine is part of the SNMP Application Programming Interface in the TCP/IP facility.

Parameters

- `myname` Contains a character string describing the SNMP multiplexing (SMUX) peer.
- `wantuser` Indicates that the isodetailor subroutine should check the $HOME directory for a isodetailor file if the value is greater than 0. If the value of the wantuser parameter is set to 0, the $HOME directory is not checked, and the myname parameter is ignored.
Files

/etc/isodetailor

Location of user’s copy of the /usr/lpp/snmpd/smux/isodetailor file.

Contains a complete list of all the logging parameters.

Related Information

The ll_hdinit, ll_dbinit, _ll_log, or ll_log subroutine.

SNMP Overview for Programmers in AIX 5L Version 5.3 Communications Programming Concepts.

II_hdinit, II_dbinit, _II_log, or II_log Subroutine

Purpose

Reports errors to log files.

Library

ISODE Library (libisode.a)

Syntax

#include <isode/logger.h>

void ll_hdinit (lp, prefix)
register LLog *lp;
char *prefix;

void ll_dbinit (lp, prefix)
register LLog *lp;
char *prefix;

int _ll_log (lp, event, ap)
register LLog *lp;
int event;
va_list ap;

int ll_log (va_alist)
va_dcl

Description

The ISODE library provides logging subroutines to put information into log files. The LLog data structure contains the log file information needed to control the associated log. The SMUX peer provides the log file information to the subroutines.

The LLog structure contains the following fields:

typedef struct ll_struct
{
    char *ll_file; /* path name to logging file */
    char *ll_hdr; /* text to put in opening line */
    char *ll_dhdr; /* dynamic header - changes */
    int ll_events; /* loggable events */
    int ll_systog; /* loggable events to send to syslog */
    int ll_msize; /* max size for log, in Kbytes */

    /* takes same values as ll_events */
    int ll_msize; /* if ll_msize < 0, then no checking */
    int ll_stat; /* assorted switches */
    int ll_fd; /* file descriptor */
} LLog;
The possible values for the `ll_events` and `ll_syslog` fields are:

- `LLOG_NONE` 0 /* No logging is performed */
- `LLOG_FATAL` 0x01 /* fatal errors */
- `LLOG_EXCEPTIONS` 0x02 /* exceptional events */
- `LLOG_NOTICE` 0x04 /* informational notices */
- `LLOG_PDU` 0x08 /* PDU printing */
- `LLOG_TRACE` 0x10 /* program tracing */
- `LLOG_DEBUG` 0x20 /* full debugging */
- `LLOG_ALL` 0xff /* All of the above logging */

The possible values for the `ll_stat` field are:

- `LLOGNIL` 0x00 /* No status information */
- `LLOGCLS` 0x01 /* keep log closed, except writing */
- `LLOGCRT` 0x02 /* create log if necessary */
- `LLOGZER` 0x04 /* truncate log when limits reach */
- `LLOGERR` 0x08 /* log closed due to (soft) error */
- `LLOGTTY` 0x10 /* also log to stderr */
- `LLOGHDR` 0x20 /* static header allocated/filled */
- `LLOGDHR` 0x40 /* dynamic header allocated/filled */

The `ll_hdinit` subroutine fills the `ll_hdr` field of the `LLog` record. The subroutine allocates the memory of the static header and creates a string with the information specified by the `prefix` parameter, the current user’s name, and the process ID of the SMUX peer. It also sets the static header flag in the `ll_stat` field. If the `prefix` parameter value is null, the header flag is set to the “unknown” string.

The `ll_dbinit` subroutine fills the `ll_file` field of the `LLog` record. If the `prefix` parameter is null, the `ll_file` field is not changed. The `ll_dbinit` subroutine also calls the `ll_hdinit` subroutine with the same `lp` and `prefix` parameters. The `ll_dbinit` subroutine sets the log messages to `stderr` and starts the logging facility at its highest level.

The `_ll_log` and `ll_log` subroutines are used to print to the log file. When the `LLog` structure for the log file is set up, the `_ll_log` or `ll_log` subroutine prints the contents of the string format, with all variables filled in, to the log specified in the `lp` parameter. The `LLog` structure passes the name of the target log to the subroutine.

The expected parameter format for the `_ll_log` and `ll_log` subroutines is:

- `_ll_log`(lp, event, what, string_format, ...);
- `ll_log`(lp, event, what, string_format, ...);

The difference between the `_ll_log` and `ll_log` subroutine is that the `_ll_log` uses an explicit listing of the `LLog` structure and the `event` parameter. The `ll_log` subroutine handles all the variables as a variable list.

The `event` parameter specifies the type of message being logged. This value is checked against the `events` field in the log record. If it is a valid event for the log, the other `LLog` structure variables are written to the log.

The `what` parameter variable is a string that explains what actions the subroutines have accomplished. The rest of the variables should be in the form of a `printf` statement, a string format and the variables to fill the various variable placeholders in the string format. The final output of the logging subroutine is in the following format:

```
mm/dd hh:mm:ss ll_hdr ll_dhdr string_format what: system_error
```

where:

<table>
<thead>
<tr>
<th>Variable</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>mm/dd</code></td>
<td>Specifies the date.</td>
</tr>
</tbody>
</table>
Variable | Description
--- | ---
hh:mm:ss | Specifies the time.
ll_hdr | Specifies the value of the ll_hdr field of the LLog structure.
ll_dhdr | Specifies the value of the ll_dhdr field of the LLog structure.
string_format | Specifies the string format passed to the ll_log subroutine, with the extra variables filled in.
what | Specifies the variable that tells what has occurred. The what variable often contains the reason for the failure. For example if the memory device, /dev/mem, fails, the what variable contains the name of the /dev/mem device.
system_error | Contains the string for the errno value, if it exists.

These subroutines are part of the SNMP Application Programming Interface in the TCP/IP facility.

### Parameters

- **lp**: Contains a pointer to a structure that describes a log file. The *lp* parameter is used to describe things entered into the log, the file name, and headers.
- **prefix**: Contains a character string that is used to represent the name of the SMUX peer in the ll_hdinit subroutine. In the ll_dbinit subroutine, the prefix parameter represents the name of the log file to be used. The new log file name will be /prefix.log.
- **event**: Specifies the type of message to be logged.
- **ap**: Provides a list of variables that is used to print additional information about the status of the logging process. The first argument needs to be a character string that describes what failed. The following arguments are expected in a format similar to the printf operation, which is a string format with the variables needed to fill the format variable places.
- **va_alist**: Provides a variable list of parameters that includes the *lp*, *event*, and *ap* variables.

### Return Values

The ll_dbinit and ll_hdinit subroutines have no return values. The _ll_log and ll_log subroutines return OK on success and NOTOK on failure.

### Related Information

The isodetailor subroutine.

Examples of SMUX Error Logging Routines and SNMP Overview for Programmers in AIX 5L Version 5.3.

### o_number, o_integer, o_string, o_igeneric, o_generic, o_specific, or o_ipaddr Subroutine

#### Purpose

Encodes values retrieved from the Management Information Base (MIB) into the specified variable binding.

#### Library

SNMP Library (libsnmp.a)

#### Syntax

```c
#include <isode/.Snmp/objects.h>
#include <isode/pepsy/SNMP-types.h>
#include <sys/types.h>
#include <netinet/in.h>
```
Description

The `o_number` subroutine assigns a number retrieved from the MIB to the variable binding used to request it. Once an MIB value has been retrieved, the value must be stored in the binding structure associated with the variable requested. The `o_number` subroutine places the integer `number` into the `v` parameter, which designates the binding for the variable. The `value` parameter type is defined by the `oi` parameter and is used to specify the encoding subroutine that stores the value. The `oi` parameter references a specific MIB variable and should be the same as the variable specified in the `v` parameter. The encoding functions are defined for each type of variable and are contained in the object identifier (OI) structure.

The `o_integer` macro is defined in the `/usr/include/snmp/objects.h` file. This macro casts the `number` parameter as an integer. Use the `o_integer` macro for types that are not integers but have integer values.

The `o_string` subroutine assigns a string that has been retrieved for a MIB variable to the variable binding used to request the string. Once a MIB variable has been retrieved, the value is stored in the binding structure associated with the variable requested. The `o_string` subroutine places the string, specified with the `base` parameter, into the variable binding in the `v` parameter. The length of the string represented in the `base` parameter equals the value of the `len` parameter. The length is used to define how much of the string is copied in the binding parameter of the variable. The `value` parameter type is defined by the `oi` parameter and is used to specify the encoding subroutine that stores the value. The `oi` parameter references a specific MIB variable and should be the same as the variable specified in the `v` parameter. The encoding subroutines are defined for each type of variable and are contained in the OI structure.

The `o_generic` and `o_igeneric` subroutines assign results that are already in the customer’s MIB database. These two subroutines do not retrieve values from any other source. These subroutines check
whether the MIB database has information on how and what to encode as the value. The o_generic and o_igeneric subroutines also ensure that the variable requested is an instance. If the variable is an instance, the subroutines encode the value and return OK. The subroutine has an added set of return codes. If there is not any information about the variable, the subroutine returns NOTOK on a get_next request and int_SNMP_error__status_noSuchName for the get and set requests. The difference between the o_generic and the o_igeneric subroutine is that the o_igeneric subroutine provides a method for users to define a generic subroutine.

The o_specific subroutine sets the binding value for a MIB variable with the value in a character pointer. The o_specific subroutine ensures that the data-encoding procedure is defined. The encode subroutine is always checked by all of the o_ subroutines. The o_specific subroutine returns the normal values.

The o_ipaddr subroutine sets the binding value for variables that are network addresses. The o_ipaddr subroutine uses the sin_addr field of the sockaddr_in structure to get the address. The subroutine does the normal checking and returns the results like the rest of the subroutines.

These subroutines are part of the SNMP Application Programming Interface in the TCP/IP facility.

**Parameters**

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>oi</td>
<td>Contains the OI data structure for the variable whose value is to be recorded into the binding structure.</td>
</tr>
<tr>
<td>v</td>
<td>Specifies the variable binding parameter, which is of type type_SNMP_VarBind. The v parameter contains a name and a value field. The value field contents are supplied by the o_ subroutines.</td>
</tr>
<tr>
<td>number</td>
<td>Contains an integer to store in the value field of the v (variable bind) parameter.</td>
</tr>
<tr>
<td>base</td>
<td>Points to the character string to store in the value field of the v parameter.</td>
</tr>
<tr>
<td>len</td>
<td>Designates the length of the integer character string to copy. The character string is described by the base parameter.</td>
</tr>
<tr>
<td>offset</td>
<td>Contains an integer value of the current type of request, for example: type_SNMP_PDUs_get_next_request</td>
</tr>
<tr>
<td>value</td>
<td>Contains a character pointer to a value.</td>
</tr>
<tr>
<td>netaddr</td>
<td>Points to a sockaddr_in structure. The subroutine only uses the sin_addr field of this structure.</td>
</tr>
</tbody>
</table>

**Return Values**

The return values for these subroutines are:

<table>
<thead>
<tr>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>int_SNMP_error__status_genErr</td>
<td>Indicates an error occurred when setting the v parameter value.</td>
</tr>
<tr>
<td>int_SNMP_error__status_noErr</td>
<td>Indicates no errors found.</td>
</tr>
</tbody>
</table>

**Related Information**

- List of Network Manager Programming References
- SNMP Overview for Programmers
- Working with Management Information Base (MIB) Variables

---

**oid_cmp, oid_cpy, oid_free, sprintoid, str2oid, ode2oid, oid2ode, oid2ode_aux, prim2oid, or oid2prim Subroutine**

**Purpose**

Manipulates the object identifier data structure.
Library

ISODE Library (libisode.a)

Syntax

```c
#include <isode/psap.h>

int oid_cmp (p, q)
   OID p, q;

OID oid_cpy (oid)
   OID oid;

void oid_free (oid)
   OID oid;

char *sprintoid (oid)
   OID oid;

OID str2oid (s)
   char *s;

OID ode2oid (descriptor)
   char *descriptor;

char *oid2ode (oid)
   OID oid;

OID *oid2ode_aux (descriptor, quote)
   char *descriptor;
   int quote;

OID prim2oid (pe)
   PE pe;

PE oid2prim (oid)
   OID oid;
```

Description

These subroutines are used to manipulate and translate object identifiers. The object identifier data (OID) structure and these subroutines are defined in the `/usr/include/isode/psap.h` file.

The `oid_cmp` subroutine compares two OID structures. The `oid_cpy` subroutine copies the object identifier, specified by the `oid` parameter, into a new structure. The `oid_free` procedure frees the object identifier and does not have any return parameters.

The `sprintoid` subroutine takes an object identifier and returns the dot-notation description as a string. The string is in static storage and must be copied to other user storage if it is to be maintained. The `sprintoid` subroutine takes the object data and converts it without checking for the existence of the `oid` parameter.

The `str2oid` subroutine takes a character string specifying an object identifier in dot notation (for example, `1.2.3.6.1.2`) and converts it into an OID structure. The space is static. To get a permanent copy of the OID structure, use the `oid_cpy` subroutine.

The `oid2ode` subroutine is identical to the `sprintoid` subroutine except that the `oid2ode` subroutine checks whether the `oid` parameter is in the `isobjects` database. The `oid2ode` subroutine is implemented as a macro call to the `oid2ode_aux` subroutine. The `oid2ode_aux` subroutine is similar to the `oid2ode` subroutine except for an additional integer parameter that specifies whether the string should be enclosed by quotes. The `oid2ode` subroutine always encloses the string in quotes.
The `ode2oid` subroutine retrieves an object identifier from the `isobjects` database.

These subroutines are part of the SNMP Application Programming Interface in the TCP/IP facility.

**Parameters**

| p | Specifies an OID structure. |
| q | Specifies an OID structure. |
| descriptor | Contains the object identifier descriptor data. |
| oid | Contains the object identifier data. |
| s | Contains a character string that defines an object identifier in dot notation. |
| descriptor | Contains the object identifier descriptor data. |
| quote | Specifies an integer that indicates whether a string should be enclosed in quotes. A value of 1 adds quotes; a value of 0 does not add quotes. |
| pe | Contains a presentation element in which the OID structure is encoded (as with the `oid2prim` subroutine) or decoded (as with the `prim2oid` subroutine). |

**Return Values**

The `oid_cmp` subroutine returns a 0 if the structures are identical, -1 if the first object is less than the second, and a 1 if any other conditions are found. The `oid_cpy` subroutine returns a pointer to the designated object identifier when the subroutine is successful.

The `oid2ode` subroutine returns the dot-notation description as a string in quotes. The `sprintoid` subroutine returns the dot-notation description as a string without quotes.

The `ode2oid` subroutine returns a static pointer to the object identifier. If the `ode2oid` and `oid_cpy` subroutines are not successful, the `NULLOID` value is returned.

**Related Information**

The `oid_extend` subroutine, `oid_normalize` subroutine.

List of Network Manager Programming References

**SNMP Overview for Programmers** in AIX 5L Version 5.3 Communications Programming Concepts.

**oid_extend or oid_normalize Subroutine**

**Purpose**

Extends the base ISODE library subroutines.

**Library**

SNMP Library (libsnmp.a)

**Syntax**

```c
#include <isode/snmp/objects.h>

OID oid_extend (q, howmuch)

OID q;
integer howmuch;
```
OID oid_normalize (q, howmuch, initial)
OID q;
integer howmuch, initial;

Description
The oid_extend subroutine is used to extend the current object identifier data (OID) structure. The OID structure contains an integer number of entries and an array of integers. The oid_extend subroutine creates a new, extended OID structure with an array of the size specified in the howmuch parameter plus the original array size specified in the q parameter. The original values are copied into the first entries of the new structure. The new values are uninitialized. The entries of the OID structure are used to represent the values of an Management Information Base (MIB) tree in dot notation. Each entry represents a level in the MIB tree.

The oid_normalize subroutine extends and adjusts the values of the OID structure entries. The oid_normalize subroutine extends the OID structure and then decrements all nonzero values by 1. The new values are initialized to the value of the initial parameter. This subroutine stores network address and netmask information in the OID structure.

These subroutines do not free the q parameter.

These subroutines are part of the SNMP Application Programming Interface in the TCP/IP facility.

Parameters
q Specifies the size of the original array.
howmuch Specifies the size of the array for the new OID structure.
initial Indicates the initialized value of the OID structure extensions.

Return Values
Both subroutines, when successful, return the pointer to the new object identifier structure. If the subroutines fail, the NULLOID value is returned.

Related Information
The oid_cmp, oid_cpy, oid_free, sprintoid, str2oid, ode2oid, oid2ode, oid2ode_aux, prim2oid, or oid2prim subroutine.

SNMP Overview for Programmers in AIX 5L Version 5.3 Communications Programming Concepts.

readobjects Subroutine

Purpose
Allows the SNMP multiplexing (SMUX) peer to read the Management Information Base (MIB) variable structure.

Library
SNMP Library (libsnmp.a)

Syntax
#include <isode/snmp/objects.h>
int
readobjects ([file]
char *file;

Description
The readobjects subroutine reads the file given in the file parameter. This file must contain the MIB variable descriptions that the SMUX peer supports. The SNMP library functions require basic information about the MIB tree supported by the SMUX peer. These structures are supplied from information in the readobjects file. The text2oid subroutine receives a string description and uses the object identifier information retrieved with the readobjects subroutine to return a MIB object identifier. The file designated in the file parameter must be in the following form:

<MIB directory> <MIB position> <MIB name> <MIB position> <MIB type> <MIB access> <MIB required?>

...

An example of a file that uses this format is /etc/mib.defs. The /etc/mib.defs file defines the MIBII tree used in the SNMP agent.

This subroutine is part of the SNMP Application Programming Interface in the TCP/IP facility.

Parameters
file Contains the name of the file to be read. If the value is NULL, the /etc/mib.defs file is read.

Return Values
If the subroutine is successful, OK is returned. Otherwise, NOTOK is returned.

Related Information
The text2oid subroutine.

RFC 1155 describes the basic MIB structure.

SNMP Overview for Programmers in AIX 5L Version 5.3 Communications Programming Concepts.

s_generic Subroutine

Purpose
Sets the value of the Management Information Base (MIB) variable in the database.

Library
The SNMP Library (libsnmp.a)

Syntax
#include <isode/objects.h>

int s_generic
    (OI oi, v, offset)
    OI oi;
    register struct type_SNMP_VarBind *v;
    int offset;
Description
The **s_generic** subroutine sets the database value of the MIB variable. The subroutine retrieves the information it needs from a value in a variable binding within the Protocol Data Unit (PDU). The **s_generic** subroutine sets the MIB variable, specified by the object identifier **oi** parameter, to the value field specified by the **v** parameter.

The **offset** parameter is used to determine the stage of the set process. If the **offset** parameter value is **type_SNMP_PDUs_set__request**, the value is checked for validity and the value in the **ot_save** field in the **OI** structure is set. If the **offset** parameter value is **type_SNMP_PDUs_commit**, the value in the **ot_save** field is freed and moved to the MIB **ot_info** field. If the **offset** parameter value is **type_SNMP_PDUs_rollback**, the value in the **ot_save** field is freed and no new value is written.

This subroutine is part of the SNMP Application Programming Interface in the TCP/IP facility.

Parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>oi</strong></td>
<td>Designates the <strong>OI</strong> structure representing the MIB variable to be set.</td>
</tr>
<tr>
<td><strong>v</strong></td>
<td>Specifies the variable binding that contains the value to be set.</td>
</tr>
<tr>
<td><strong>offset</strong></td>
<td>Contains the stage of the set. The possible values for the <strong>offset</strong> parameter are <strong>type_SNMP_PDUs_commit</strong>, <strong>type_SNMP_PDUs_rollback</strong>, or <strong>type_SNMP_PDUs_set__request</strong>.</td>
</tr>
</tbody>
</table>

Return Values
If the subroutine is successful, a value of **int_SNMP_error__status_noError** is returned. Otherwise, a value of **int_SNMP_error__status_badValue** is returned.

Related Information
The **o_number**, **o_integer**, **o_string**, **o_specific**, **o_igeneric**, **o_generic**, or **o_ipaddr** subroutines.

**SNMP Overview for Programmers** and **SNMP daemon processing** in *Networks and communication management*.

**smux_close Subroutine**

**Purpose**
Ends communications with the SNMP agent.

**Library**
SNMP Library (*libsnmp.a*)

**Syntax**
```c
#include <isode/snmp/smux.h>

int smux_close ( reason )
int reason;
```

**Description**
The **smux_close** subroutine closes the transmission control protocol (TCP) connection from the SNMP multiplexing (SMUX) peer. The **smux_close** subroutine sends the close protocol data unit (PDU) with the error code set to the **reason** value. The subroutine closes the TCP connection and frees the socket. This subroutine also frees information it was maintaining for the connection.
This subroutine is part of the SNMP Application Programming Interface in the TCP/IP facility.

Parameters

reason Indicates an integer value denoting the reason the close PDU message is being sent.

Return Values

If the subroutine is successful, OK is returned. Otherwise, NOTOK is returned.

Error Codes

If the subroutine returns NOTOK, the smux_errno global variable is set to one of the following values:

<table>
<thead>
<tr>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>invalidOperation</td>
<td>Indicates that the smux_init subroutine has not been executed successfully.</td>
</tr>
<tr>
<td>congestion</td>
<td>Indicates that memory could not be allocated for the close PDU. The TCP connection is closed.</td>
</tr>
<tr>
<td>youLoseBig</td>
<td>Indicates that the SNMP code has a problem. The TCP connection is closed.</td>
</tr>
</tbody>
</table>

Related Information

The smux_error subroutine, smux_init subroutine, smux_register subroutine, smux_response subroutine, smux_simple_open subroutine, smux_trap subroutine, smux_wait subroutine.

RFC 1227, SNMP MUX Protocol and MIB.

SNMP Overview for Programmers in AIX 5L Version 5.3 Communications Programming Concepts.

smux_error Subroutine

Purpose

Creates a readable string from the smux_errno global variable value.

Library

SNMP Library (libsnmp.a)

Syntax

```c
#include <isode/snmp/smux.h>

char *smux_error (error]
int error;
```

Description

The smux_error subroutine creates a readable string from error code values in the smux_errno global variable in the smux.h file. The smux global variable, smux_errno, is set when an error occurs. The smux_error subroutine can also get a string that interprets the value of the smux_errno variable. The smux_error subroutine can be used to retrieve any numbers, but is most useful interpreting the integers returned in the smux_errno variable.

This subroutine is part of the SNMP Application Programming Interface in the TCP/IP facility.
Parameters

error Contains the error to interpret. Usually called with the value of the smux_errno variable, but can be called with any error that is an integer.

Return Values
If the subroutine is successful, a pointer to a static string is returned. If an error occurs, a string of the type SMUX error %s(%d) is returned. The %s value is a string representing the explanation of the error. The %d is the number used to reference that error.

Related Information
The smux_close subroutine, smux_init subroutine, smux_register subroutine, smux_response subroutine, smux_simple_open subroutine, smux_trap subroutine, smux_wait subroutine.

RFC 1227, SNMP MUX Protocol and MIB.

SNMP Overview for Programmers in AIX 5L Version 5.3 Communications Programming Concepts.

smux_free_tree Subroutine

Purpose
Frees the object tree when a smux tree is unregistered.

Library
SNMP Library (libsnmp.a)

Syntax
#include <isode/snmp/smux.h>

void smux_free_tree (parent, child)
char *parent;
char *child;

Description
The smux_free_tree subroutine frees elements in the Management Information Base (MIB) list within an SNMP multiplexing (SMUX) peer. If the SMUX peer implements the MIB list with the readobjects subroutine, a list of MIBs is created and maintained. These MIBs are kept in the object tree (OT) data structures.

Unlike the smux_register subroutine, the smux_free_tree subroutine frees the MIB elements even if the tree is unregistered by the snmpd daemon. This functionality is not performed by the smux_register routine because the OT list is created independently of registering a tree with the snmpd daemon. The unregistered objects should be removed as the user deems appropriate. Remove the unregistered objects if the smux peer is highly dynamic. If the peer registers and unregisters many trees, it might be reasonable to add and delete the OT MIB list on the fly. The smux_free_tree subroutine expects the parent of the MIB tree in the local OT list to delete unregistered objects.

This subroutine does not return values or error codes.

This subroutine is part of the SNMP Application Programming Interface in the TCP/IP facility.
Parameters

parent  Contains a character string holding the immediate parent of the tree to be deleted.
child   Contains a character string holding the beginning of the tree to be deleted.

The character strings are names or dot notations representing object identifiers.

Related Information

The `snmpd` command.
The `readobjects` subroutine, `smux_register` subroutine.

SNMP Overview for Programmers in AIX 5L Version 5.3 Communications Programming Concepts.

smux_init Subroutine

Purpose

Initiates the transmission control protocol (TCP) socket that the SNMP multiplexing (SMUX) agent uses and clears the basic SMUX data structures.

Library

SNMP Library (libsnmp.a)

Syntax

```c
#include <isode/snmp/smux.h>

int smux_init (int debug)
int debug;
```

Description

The `smux_init` subroutine initializes the TCP socket used by the SMUX agent to talk to the SNMP daemon. The subroutine assumes that loopback will be used to define the path to the SNMP daemon. The subroutine also clears the base structures the SMUX code uses. This subroutine also sets the debug level that is used when running the SMUX subroutines.

This subroutine is part of the SNMP Application Programming Interface in the TCP/IP facility.

Parameters

`debug` Indicates the level of debug to be printed during SMUX subroutines.

Return Values

If the subroutine is successful, the socket descriptor is returned. Otherwise, the value of NOTOK is returned and the `smux_errno` global variable is set.
Error Codes

Possible values for the smux_errno global variable are:

<table>
<thead>
<tr>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>congestion</td>
<td>Indicates memory allocation problems</td>
</tr>
<tr>
<td>youLoseBig</td>
<td>Signifies problem with SNMP library code</td>
</tr>
<tr>
<td>systemError</td>
<td>Indicates TCP connection failure.</td>
</tr>
</tbody>
</table>

These are defined in the /usr/include/isode/snmp/smux.h file.

Related Information

The smux_close subroutine, smux_error subroutine, smux_register subroutine, smux_response subroutine, smux_simple_open subroutine, smux_trap subroutine, smux_wait subroutine.

RFC 1227, SNMP MUX Protocol and MIB.

SNMP Overview for Programmers in AIX 5L Version 5.3 Communications Programming Concepts.

smux_register Subroutine

Purpose

Registers a section of the Management Information Base (MIB) tree with the Simple Network Management Protocol (SNMP) agent.

Library

SNMP Library (libsnmp.a)

Syntax

```c
#include <isode/snmp/smux.h>

int smux_register (subtree, priority, operation);

OID subtree;
int priority;
int operation;
```

Description

The smux_register subroutine registers the section of the MIB tree for which the SMUX peer is responsible with the SNMP agent. Using the smux_register subroutine, the SMUX peer informs the SNMP agent of both the level of responsibility the SMUX peer has and the sections of the MIB tree for which it is responsible. The level of responsibility (priority) the SMUX peer sends determines which requests it can answer. Lower priority numbers correspond to higher priority.

If a tree is registered more than once, the SNMP agent sends requests to the registered SMUX peer with the highest priority. If the priority is set to -1, the SNMP agent attempts to give the SMUX peer the highest available priority. The operation parameter defines whether the MIB tree is added with readOnly or readWrite permissions, or if it should be deleted from the list of register trees. The SNMP agent returns an acknowledgment of the registration. The acknowledgment indicates the success of the registration and the actual priority received.

This subroutine is part of the SNMP Application Programming Interface in the TCP/IP facility.
Parameters

subtree
Indicates an object identifier that contains the root of the MIB tree to be registered.

priority
Indicates the level of responsibility that the SMUX peer has on the MIB tree. The priority levels
range from 0 to \((2^{31} - 2)\). The lower the priority number, the higher the priority. A priority of -1 tells
the SNMP daemon to assign the highest priority currently available.

operation
Specifies the operation for the SNMP agent to apply to the MIB tree. Possible values are delete,
readOnly, or readWrite. The delete operation removes the MIB tree from the SMUX peers in the
eyes of the SNMP agent. The other two values specify the operations allowed by the SMUX peer
on the MIB tree that is being registered with the SNMP agent.

Return Values

The values returned by this subroutine are OK on success and NOTOK on failure.

Error Codes

If the subroutine is unsuccessful, the smux_errno global variable is set to one of the following values:

<table>
<thead>
<tr>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>parameterMissing</td>
<td>Indicates a parameter was null. When the parameter is fixed, the smux_register subroutine can be reissued.</td>
</tr>
<tr>
<td>invalidOperation</td>
<td>Indicates that the smux_register subroutine is trying to perform this operation before a smux_init operation has successfully completed. Start over with a new smux_init subroutine call.</td>
</tr>
<tr>
<td>congestion</td>
<td>Indicates a memory problem occurred. The TCP connection is closed. Start over with a new smux_init subroutine call.</td>
</tr>
<tr>
<td>youLoseBig</td>
<td>Indicates an SNMP code problem has occurred. The TCP connection is closed. Start over with a new smux_init subroutine call.</td>
</tr>
</tbody>
</table>

Related Information

The smux_close subroutine, smux_error subroutine, smux_init subroutine, smux_response subroutine,
smux_simple_open subroutine, smux_trap subroutine, smux_wait subroutine.

RFC1227, SNMP MUX Protocol and MIB.

SNMP Overview for Programmers in AIX 5L Version 5.3 Communications Programming Concepts.

smux_response Subroutine

Purpose

Sends a response to a Simple Network Management Protocol (SNMP) agent.

Library

SNMP Library (libsnmp.a)

Syntax

```c
#include <isode/snmp/smux.h>

int smux_response (event)
struct type_SNMP_GetResponse__PDU *event;
```
Description
The **smux_response** subroutine sends a protocol data unit (PDU), also called an event, to the SNMP agent. The subroutine does not check whether the Management Information Base (MIB) tree is properly registered. The subroutine checks only to see whether a Transmission Control Protocol (TCP) connection to the SNMP agent exists and ensures that the `event` parameter is not null.

This subroutine is part of the SNMP Application Programming Interface in the TCP/IP facility.

Parameters

```
event Specifies a **type_SNMP_GetResponse__PDU** variable that contains the response PDU to send to the SNMP agent.
```

Return Values
If the subroutine is successful, **OK** is returned. Otherwise, **NOTOK** is returned.

Error Codes
If the subroutine is unsuccessful, the **smux_errno** global variable is set to one of the following values:

<table>
<thead>
<tr>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>parameterMissing</td>
<td>Indicates the parameter was null. When the parameter is fixed, the subroutine can be reissued.</td>
</tr>
<tr>
<td>invalidOperation</td>
<td>Indicates the subroutine was attempted before the <strong>smux_init</strong> subroutine successfully completed. Start over with the <strong>smux_init</strong> subroutine.</td>
</tr>
<tr>
<td>youLoseBig</td>
<td>Indicates a SNMP code problem has occurred and the TCP connection is closed. Start over with the <strong>smux_init</strong> subroutine.</td>
</tr>
</tbody>
</table>

Related Information
The **smux_close** subroutine, **smux_error** subroutine, **smux_init** subroutine, **smux_register** subroutine, **smux_simple_open** subroutine, **smux_trap** subroutine, **smux_wait** subroutine.

RFC 1227, SNMP MUX Protocol and MIB.

SNMP Overview for Programmers in AIX 5L Version 5.3 Communications Programming Concepts.

---

**smux_simple_open Subroutine**

**Purpose**
Sends the open protocol data unit (PDU) to the Simple Network Management Protocol (SNMP) daemon.

**Library**
SNMP Library (libsnmp.a)

**Syntax**

```
#include <isode/snmp/smux.h>

int smux_simple_open (identity, description, commname, commlen)
```

18  Technical Reference: Communications, Volume 2
**Description**

Following the `smux_init` command, the `smux_simple_open` subroutine alerts the SNMP daemon that incoming messages are expected. Communication with the SNMP daemon is accomplished by sending an open PDU to the SNMP daemon. The `smux_simple_open` subroutine uses the `identity` object-identifier parameter to identify the SNMP multiplexing (SMUX) peer that is starting to communicate. The `description` parameter describes the SMUX peer. The `commname` and the `commlen` parameters supply the password portion of the open PDU. The `commname` parameter is the password used to authenticate the SMUX peer. The SNMP daemon finds the password in the `/etc/snmpd.conf` file. The SMUX peer can store the password in the `/etc/snmpd.peers` file. The `commlen` parameter specifies the length of the `commname` parameter value.

This subroutine is part of the SNMP Application Programming Interface in the TCP/IP facility.

**Parameters**

- **identity**
  Specifies an object identifier that describes the SMUX peer.

- **description**
  Contains a string of characters that describes the SMUX peer. The `description` parameter value cannot be longer than 254 characters.

- **commname**
  Contains the password to be sent to the SNMP agent. Can be a null value.

- **commlen**
  Indicates the length of the community name (`commname` parameter) to be sent to the SNMP agent. The value for this parameter must be at least 0.

**Return Values**

The subroutine returns an integer value of **OK** on success or **NOTOK** on failure.

**Error Codes**

If the subroutine is unsuccessful, the `smux_errno` global variable is set one of the following values:

<table>
<thead>
<tr>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>parameterMissing</td>
<td>Indicates that a parameter was null. The <code>commname</code> parameter can be null, but the <code>commlen</code> parameter value should be at least 0.</td>
</tr>
<tr>
<td>invalidOperation</td>
<td>Indicates that the <code>smux_init</code> subroutine did not complete successfully before the <code>smux_simple_open</code> subroutine was attempted. Correct the parameters and reissue the <code>smux_simple_open</code> subroutine.</td>
</tr>
<tr>
<td>inProgress</td>
<td>Indicates that the <code>smux_init</code> call has not completed the TCP connection. The <code>smux_simple_open</code> can be reissued.</td>
</tr>
<tr>
<td>systemError</td>
<td>Indicates the TCP connection was not completed. Do not reissue this subroutine without restarting the process with a <code>smux_init</code> subroutine call.</td>
</tr>
<tr>
<td>congestion</td>
<td>Indicates a lack of available memory space. Do not reissue this subroutine without restarting the process with a <code>smux_init</code> subroutine call.</td>
</tr>
<tr>
<td>youLoseBig</td>
<td>The SNMP code is having problems. Do not reissue this subroutine without restarting the process with a <code>smux_init</code> subroutine call.</td>
</tr>
</tbody>
</table>

**Related Information**

- The `smux_close` subroutine, `smux_error` subroutine, `smux_init` subroutine, `smux_register` subroutine, `smux_response` subroutine, `smux_trap` subroutine, `smux_wait` subroutine.

**List of Network Manager Programming References**

- RFC 1227, SNMP MUX Protocol and MIB.
- SNMP Overview for Programmers in AIX 5L Version 5.3 Communications Programming Concepts.
smux_trap Subroutine

Purpose
Sends SNMP multiplexing (SMUX) peer traps to the Simple Network Management Protocol (SNMP) agent.

Library
SNMP Library (libsnmp.a)

Syntax
#include <isode/snmp/smux.h>

int smux_trap (generic, specific, bindings)
int generic;
int specific;
struct type_SNMP_VarBindList *bindings;

Description
The smux_trap subroutine allows the SMUX peer to generate traps and send them to the SNMP agent. The subroutine sets the generic and specific fields in the trap packet to values specified by the parameters. The subroutine also allows the SMUX peer to send a list of variable bindings to the SNMP agent. The variable bindings are values associated with specific variables. If the trap is to return a set of variables, the variables are sent in the variable binding list.

This subroutine is part of the SNMP Application Programming Interface in the TCP/IP facility.

Parameters

generic Contains an integer specifying the generic trap type. The value must be one of the following:
0 Specifies a cold start.
1 Specifies a warm start.
2 Specifies a link down.
3 Specifies a link up.
4 Specifies an authentication failure.
5 Specifies an EGP neighbor loss.
6 Specifies an enterprise-specific trap type.

specific Contains an integer that uniquely identifies the trap. The unique identity is typically assigned by the registration authority for the enterprise owning the SMUX peer.

bindings Indicates the variable bindings to assign to the trap protocol data unit (PDU).

Return Values
The subroutine returns NOTOK on failure and OK on success.

Error Codes
If the subroutine is unsuccessful, the smux_errno global variable is set to one of the following values:

<table>
<thead>
<tr>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>invalidOperation</td>
<td>Indicates the Transmission Control Protocol (TCP) connection was not completed.</td>
</tr>
<tr>
<td>congestion</td>
<td>Indicates memory is not available. The TCP connection was closed.</td>
</tr>
<tr>
<td>youLoseBig</td>
<td>Indicates an error occurred in the SNMP code. The TCP connection was closed.</td>
</tr>
</tbody>
</table>
Related Information

The `smux_close` subroutine, `smux_error` subroutine, `smux_init` subroutine, `smux_register` subroutine, `smux_response` subroutine, `smux_simple_open` subroutine, `smux_wait` subroutine.

RFC 1227, SNMP MUX Protocol and MIB.

SNMP Overview for Programmers in AIX 5L Version 5.3 Communications Programming Concepts.

### smux_wait Subroutine

#### Purpose

Waits for a message from the Simple Network Management Protocol (SNMP) agent.

#### Library

SNMP Library (`libsnmp.a`)

#### Syntax

```
#include <isode/snmp/smux.h>

int smux_wait (struct type_SMUX_PDUs **event, int isecs);
```

#### Description

The `smux_wait` subroutine waits for a period of seconds, designated by the value of the `isecs` parameter, and returns the protocol data unit (PDU) received. The `smux_wait` subroutine waits on the socket descriptor that is initialized in a `smux_init` subroutine and maintained in the SMUX subroutines. The `smux_wait` subroutine waits up to `isecs` seconds. If the value of the `isecs` parameter is 0, the `smux_wait` subroutine returns only the first packet received. If the value of the `isecs` parameter is less than 0, the `smux_wait` subroutine waits indefinitely for the next message or returns a message already received. If no data is received, the `smux_wait` subroutine returns an error message of `NOTOK` and sets the `smux_errno` variable to the `inProgress` value. If the `smux_wait` subroutine is successful, it returns the first PDU waiting to be received. If a close PDU is received, the subroutine will automatically close the TCP connection and return `OK`.

This subroutine is part of the SNMP Application Programming Interface in the TCP/IP facility.

#### Parameters

- `event` Points to a pointer of `type_SMUX_PDUs`. This holds the PDUs received by the `smux_wait` subroutine.
- `isecs` Specifies an integer value equal to the number of seconds to wait for a message.

#### Return Values

If the subroutine is successful, the value `OK` is returned. Otherwise, the return value is `NOTOK`. 

Chapter 1. Simple Network Management Protocol (SNMP) 21
Error Codes
If the subroutine is unsuccessful, the `smux_errno` global variable is set to one of the following values:

<table>
<thead>
<tr>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>parameterMissing</td>
<td>Indicates that the <code>event</code> parameter value was null.</td>
</tr>
<tr>
<td>inProgress</td>
<td>Indicates that there was nothing for the subroutine to receive.</td>
</tr>
<tr>
<td>invalidOperation</td>
<td>Indicates that the <code>smux_init</code> subroutine was not called or failed to operate.</td>
</tr>
<tr>
<td>youLoseBig</td>
<td>Indicates an error occurred in the SNMP code. The TCP connection was closed.</td>
</tr>
</tbody>
</table>

Related Information
The `smux_close` subroutine, `smux_error` subroutine, `smux_init` subroutine, `smux_register` subroutine, `smux_response` subroutine, `smux_simple_open` subroutine, `smux_trap` subroutine.

RFC1227, SNMP MUX Protocol and MIB.

SNMP Overview for Programmers in AIX 5L Version 5.3 Communications Programming Concepts.

text2inst, name2inst, next2inst, or nextot2inst Subroutine

Purpose
Retrieves instances of variables from various forms of data.

Library
SNMP Library (`libsnmp.a`)

Syntax
```
#include <isode/snmp/objects.h>

OI text2inst ( text )
char *text;

OI name2inst ( oid )
OID oid;
OI next2inst ( oid )
OID oid;

OI nextot2inst ( oid, ot )
OID oid;
OT ot;
```

Description
These subroutines return pointers to the actual objects in the database. When supplied with a way to identify the object, the subroutines return the corresponding object.

The `text2inst` subroutine takes a character string object identifier from the `text` parameter. The object’s database is then examined for the specified object. If the specific object is not found, the `NULLOI` value is returned.

The `name2inst` subroutine uses an object identifier structure specified in the `oid` parameter to specify which object is desired. If the object cannot be found, a `NULLOI` value is returned.
The `next2inst` and `nextot2inst` subroutines find the next object in the database given an object identifier. The `next2inst` subroutine starts at the root of the tree, while the `nextot2inst` subroutine starts at the object given in the `ot` parameter. If another object cannot be found, the `NULLOI` value will be returned.

These subroutines are part of the SNMP Application Programming Interface in the TCP/IP facility.

**Parameters**

- `text` Specifies the character string used to identify the object wanted in the `text2inst` subroutine.
- `oid` Specifies the object identifier structure used to identify the object wanted in the `name2inst`, `next2inst`, and `nextot2inst` subroutines.
- `ot` Specifies an object in the database used as a starting point for the `nextot2inst` subroutine.

**Return Values**

If the subroutine is successful, an `OI` value is returned. `OI` is a pointer to an object in the database. On a failure, a `NULLOI` value is returned.

**Related Information**

The `text2oid` subroutine, `text2obj` subroutine.

*SNMP Overview for Programmers* in AIX 5L Version 5.3 Communications Programming Concepts.

---

**text2oid or text2obj Subroutine**

**Purpose**

Converts a text string into some other value.

**Library**

SNMP Library (`libsnmp.a`)

**Syntax**

```
#include <isode/snmp/objects.h>

OID text2oid (text)
char *text;

OT text2obj (text)
char *text;
```

**Description**

The `text2oid` subroutine takes a character string and returns an object identifier. The string can be a name, a name.numbers, or dot notation. The returned object identifier is in memory-allocation storage and should be freed when the operation is completed with the `oid_free` subroutine.

The `text2obj` subroutine takes a character string and returns an object. The string needs to be the name of a specific object. The subroutine returns a pointer to the object.

These subroutines are part of the SNMP Application Programming Interface in the TCP/IP facility.

**Parameters**

- `text` Contains a text string used to specify the object identifier or object to be returned.
Return Values
On a successful execution, these subroutines return completed data structures. If a failure occurs, the `text2oid` subroutine returns a `NULLOID` value and the `text2obj` returns a `NULLOT` value.

Related Information
The `malloc` subroutine, `oid_free` subroutine, `text2inst` subroutine.

SNMP Overview for Programmers in AIX 5L Version 5.3 Communications Programming Concepts.
Chapter 2. Sockets

_getlong Subroutine

Purpose
Retrieves long byte quantities.

Library
Standard C Library (libc.a)

Syntax
#include <sys/types.h>
#include <netinet/in.h>
#include <arpa/nameser.h>
#include <resolv.h>

unsigned long _getlong (MessagePtr)
u_char *MessagePtr;

Description
The _getlong subroutine gets long quantities from the byte stream or arbitrary byte boundaries.

The _getlong subroutine is one of a set of subroutines that form the resolver, a set of functions that resolves domain names. Global information used by the resolver subroutines is kept in the _res data structure. The /usr/include/resolv.h file contains the _res structure definition.

All applications containing the _getlong subroutine must be compiled with the _BSD macro set to a specific value. Acceptable values are 43 and 44. In addition, all socket applications must include the BSD libbsd.a library.

Parameters
MessagePtr Specifies a pointer into the byte stream.

Return Values
The _getlong subroutine returns an unsigned long (32-bit) value.

Files
/etc/resolv.conf Lists name server and domain names.

Related Information
The dn_comp subroutine, dn_expand subroutine, _getshort subroutine, putlong subroutine, putshort subroutine, res_init subroutine, res_mkquery subroutine, “res_ninit Subroutine” on page 165, res_query subroutine, res_search subroutine, res_send subroutine.

Sockets Overview and Understanding Domain Name Resolution in AIX 5L Version 5.3 Communications Programming Concepts.
_getshort Subroutine

Purpose
Retrieves short byte quantities.

Library
Standard C Library (libc.a)

Syntax
#include <sys/types.h>
#include <netinet/in.h>
#include <arpa/nameser.h>
#include <resolv.h>

unsigned short getshort (MessagePtr)
u_char *MessagePtr;

Description
The _getshort subroutine gets quantities from the byte stream or arbitrary byte boundaries.

The _getshort subroutine is one of a set of subroutines that form the resolver, a set of functions that resolve domain names. Global information used by the resolver subroutines is kept in the _res data structure. The /usr/include/resolv.h file contains the _res structure definition.

All applications containing the _getshort subroutine must be compiled with the _BSD macro set to a specific value. Acceptable values are 43 and 44. In addition, all socket applications must include the BSD libbsd.a library.

Parameters
MessagePtr Specifies a pointer into the byte stream.

Return Values
The _getshort subroutine returns an unsigned short (16-bit) value.

Files
/etc/resolv.conf Defines name server and domain names.

Related Information
The dn_comp subroutine, dn_expand subroutine, _getlong subroutine, putlong subroutine, putshort subroutine, res_init subroutine, res_mkquery subroutine, “res_ninit Subroutine” on page 165, res_send subroutine.

Sockets Overview and Understanding Domain Name Resolution in AIX 5L Version 5.3 Communications Programming Concepts.
_putlong Subroutine

Purpose
Places long byte quantities into the byte stream.

Library
Standard C Library (libc.a)

Syntax
#include <sys/types.h>
#include <netinet/in.h>
#include <arpa/nameser.h>
#include <resolv.h>

void _putlong (Long, MessagePtr)
unsigned long Long;
u_char *MessagePtr;

Description
The _putlong subroutine places long byte quantities into the byte stream or arbitrary byte boundaries.

The _putlong subroutine is one of a set of subroutines that form the resolver a set of functions that resolve domain names. Global information used by the resolver subroutines is kept in the _res data structure. The /usr/include/resolv.h file contains the _res structure definition.

All applications containing the _putlong subroutine must be compiled with the _BSD macro set to a specific value. Acceptable values are 43 and 44. In addition, all socket applications must include the BSD libbsd.a library.

Parameters
Long Represents a 32-bit integer.
MessagePtr Represents a pointer into the byte stream.

Files
/etc/resolv.conf Lists the name server and domain name.

Related Information

Sockets Overview and Understanding Domain Name Resolution in AIX 5L Version 5.3 Communications Programming Concepts.
_putshort Subroutine

Purpose
Places short byte quantities into the byte stream.

Library
Standard C Library (libc.a)

Syntax
#include <sys/types.h>
#include <netinet/in.h>
#include <arpa/nameser.h>
#include <resolv.h>

void _putshort ( Short, MessagePtr)
unsigned short Short;
u_char *MessagePtr;

Description
The _putshort subroutine puts short byte quantities into the byte stream or arbitrary byte boundaries.

The _putshort subroutine is one of a set of subroutines that form the resolver a set of functions that resolve domain names. Global information used by the resolver subroutines is kept in the _res data structure. The /usr/include/resolv.h file contains the _res structure definition.

All applications containing the _putshort subroutine must be compiled with the _BSD macro set to a specific value. Acceptable values are 43 and 44. In addition, all socket applications must include the BSD libbsd.a library.

Parameters
Short Represents a 16-bit integer.
MessagePtr Represents a pointer into the byte stream.

Files
/etc/resolv.conf Lists the name server and domain name.

Related Information

Sockets Overview and Understanding Domain Name Resolution in AIX 5L Version 5.3 Communications Programming Concepts.
accept Subroutine

Purpose
Accepts a connection on a socket to create a new socket.

Library
Standard C Library (libc.a)

Syntax
#include <sys/socket.h>

int accept (Socket, Address, AddressLength);

Parameters

Socket Specifies a socket created with the socket subroutine that is bound to an address with the bind subroutine and has issued a successful call to the listen subroutine.

Address Specifies a result parameter that is filled in with the address of the connecting entity as known to the communications layer. The exact format of the Address parameter is determined by the domain in which the communication occurs.

AddressLength Specifies a parameter that initially contains the amount of space pointed to by the Address parameter. Upon return, the parameter contains the actual length (in bytes) of the address returned. The accept subroutine is used with SOCK_STREAM socket types.

Return Values
Upon successful completion, the accept subroutine returns the nonnegative socket descriptor of the accepted socket.
If the `accept` subroutine is unsuccessful, the subroutine handler performs the following functions:

- Returns a value of -1 to the calling program.
- Moves an error code, indicating the specific error, into the `errno` global variable.

### Error Codes

The `accept` subroutine is unsuccessful if one or more of the following is true:

<table>
<thead>
<tr>
<th>Error Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EBADF</td>
<td>The <code>Socket</code> parameter is not valid.</td>
</tr>
<tr>
<td>EINTR</td>
<td>The <code>accept</code> function was interrupted by a signal that was caught before a valid connection arrived.</td>
</tr>
<tr>
<td>EINVAL</td>
<td>The socket referenced by <code>s</code> is not currently a listen socket or has been shutdown with <code>shutdown</code>. A <code>listen</code> must be done before an <code>accept</code> is allowed.</td>
</tr>
<tr>
<td>EMFILE</td>
<td>The system limit for open file descriptors per process has already been reached (<code>OPEN_MAX</code>).</td>
</tr>
<tr>
<td>ENFILE</td>
<td>The maximum number of files allowed are currently open.</td>
</tr>
<tr>
<td>ENOTSOCK</td>
<td>The <code>Socket</code> parameter refers to a file, not a socket.</td>
</tr>
<tr>
<td>EOPNOTSUPP</td>
<td>The referenced socket is not of type <code>SOCK_STREAM</code>.</td>
</tr>
<tr>
<td>EFAULT</td>
<td>The <code>Address</code> parameter is not in a writable part of the user address space.</td>
</tr>
<tr>
<td>EWOULDBLOCK</td>
<td>The socket is marked as nonblocking, and no connections are present to be accepted.</td>
</tr>
<tr>
<td>ENETDOWN</td>
<td>The network with which the socket is associated is down.</td>
</tr>
<tr>
<td>ENOTCONN</td>
<td>The socket is not in the connected state.</td>
</tr>
<tr>
<td>ECONNABORTED</td>
<td>The client aborted the connection.</td>
</tr>
</tbody>
</table>

### Examples

As illustrated in this program fragment, once a socket is marked as listening, a server process can accept a connection:

```c
struct sockaddr_in from;
.
.
.
fromlen = sizeof(from);
newsock = accept(socket, (struct sockaddr*)&from, &fromlen);
```

### Related Information

The `connect` subroutine, `getsockname` subroutine, `listen` subroutine, `socket` subroutine.

[Accepting UNIX® Stream Connections Example Program](#), [Binding Names to Sockets](#), [Sockets Overview](#), [Understanding Socket Connections](#), and [Understanding Socket Creation](#) in AIX 5L Version 5.3 Communications Programming Concepts.

---

### arprecsole_common Subroutine

**Purpose**

Reads or creates new arp entries so that hardware addresses can be resolved.

**Syntax**

```c
int arprecsole_common (ac, arpwhohas, dst, hwaddr, swhaddr, extra, if_dependent)
```

```c
register struct arpcom *ac;
struct mbuf *m;
int (*arpwhohas)(register struct arpcom *, struct in_addr *addr, int skipbestif, void *extra),
struct sockaddr_in *dst;
```
The `arpresolve_common` subroutine reads or creates new arp entries so that hardware addresses can be resolved. It is called by `arpresolve` from the IF layer of the interface. If the arp entry is complete, then `arpresolve_common` returns the address pointed to by `hwaddr` and the data pointed to by `if_dependent` if `if_dependent` is true. If the arp entry is not complete, then this subroutine adds the memory buffer pointed to by `mbuf` to `at_hold`. `at_hold` holds one or more packets that are waiting for the arp entry to complete so they can be transmitted.

If an arp entry does not exist, `arpresolve_common` creates a new entry by calling `arptnew` and then adds the memory buffer pointed to by `mbuf` to `at_hold`. This subroutine calls `arpwhohas` when it creates a new arp entry or when the timer for the incomplete arp entry (with the IP address that is pointed to by `dst`) has expired.

### Parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ac</td>
<td>Points to the <code>arpcom</code> structure.</td>
</tr>
<tr>
<td>m</td>
<td>Points to the memory buffer (<code>mbuf</code>), which will be added to the list awaiting completion of the arp table entry.</td>
</tr>
<tr>
<td>arpwhohas</td>
<td>Points to the <code>arpwhohas</code> subroutine.</td>
</tr>
<tr>
<td>addr</td>
<td>Points to the <code>in_addr</code> structure’s address.</td>
</tr>
<tr>
<td>extra</td>
<td>A void pointer that can be used in the future so that IF layers can pass extra structures to <code>arpwhohas</code>.</td>
</tr>
<tr>
<td>dst</td>
<td>Points to the <code>sockaddr_in</code> structure. This structure has the destination IP address.</td>
</tr>
<tr>
<td>hwaddr</td>
<td>Points to the buffer. This buffer contains the hardware address if it finds a completed entry.</td>
</tr>
<tr>
<td>szhwaddr</td>
<td>Size of the buffer pointed to by <code>hwaddr</code>.</td>
</tr>
<tr>
<td>if_dependent</td>
<td>Pointer to the <code>if_dependent</code> structure. <code>arpresolve_common</code> uses this to pass the <code>if_dependent</code> data, which is part of the <code>arptab</code> entry, to the calling function.</td>
</tr>
</tbody>
</table>

### Return Values

- **ARP_MBUF**: The arp entry is not complete.
- **ARP_HWADDR**: The `hwaddr` buffer is filled with the hardware address.
- **ARP_FLG_NOARP**: The arp entry does not exist, and the `IFF_NOARP` flag is set only if the value of `if_type` is `IFT_ETHER`.

### Related Information

"arpupdate Subroutine"

### arpuupdate Subroutine

**Purpose**

Updates arp entries for a given IP address.
Syntax

```c
int arpupdate (ac, m, hp, action, prm);
register struct arpcom *ac;
struct mbuf *m;
caddr_t hp;
int action;
struct arpupdate_parm *prm;
```

Description

The `arpupdate` subroutine updates arp entries for a given IP address. It is called by `arpinput` from the IF layer of the interface. This subroutine searches the arp table for an entry that matches the IP address. It then updates the arp entry for the given IP address. The `arpupdate` subroutine also performs reverse arp lookups.

The `arpupdate` subroutine enters a new address in `arptab`, pushing out the oldest entry from the bucket if there is no room. This subroutine always succeeds because no bucket can be completely filled with permanent entries (except when `arpioctl` tests whether another permanent entry can fit).

Depending on the action specified, the `prm` IP addresses `isaddr`, `itaddr`, and `myaddr` are used by the `arpupdate` subroutine.

Parameters

- `ac` Points to the `arpcom` structure.
- `m` Points to the memory buffer (`mbuf`), that contains the arp response packet received by the interface.
- `hp` Points to the buffer that is passed by the interrupt handler.
- `action` Returns a value that indicates which action is taken:
  - **LOOK** Looks for the `isaddr` IP address in the arp table and returns the hardware address and `if_dependent` structure.
  - **LK PUB** Looks for the `isaddr` IP address in the arp table and returns the hardware address and `if_dependent` structure only if the `ATF_PUBL` is set.
  - **UPD T** Updates the arp entry for an IP address (`isaddr`). If no arp entry is there, creates a new one and updates the `if_dependent` structure using the `ptr` function passed in the `prm` structure.
  - **RE VARP** Reverses the arp request. `hwaddr` contains the hardware address, `szhwaddr` indicates its size, and `sad dr` returns the IP address if an entry is found.
- `prm` Points to the `arpupdate_parm` structure. The values are:
  - **LOOK** or **LKP UB** `itaddr` and `myaddr` are ignored. `isaddr` is used for arp table lookup.
  - **UPD T** `isaddr` points to the sender protocol address. `itaddr` points to the target protocol address. `myaddr` points to the protocol address of the interface that received the packet.

Return Values

- **ARP_OK** Lookup or update was successful.
- **ARP_FAIL** Lookup or update failed.
- **ARP_NEWF** New arp entry could not be created.
bind Subroutine

Purpose
Binds a name to a socket.

Library
Standard C Library (libc.a)

Syntax
#include <sys/socket.h>

int bind (Socket, Name, NameLength)
int Socket;
const struct sockaddr *Name;
socklen_t NameLength;

Description
The bind subroutine assigns a Name parameter to an unnamed socket. Sockets created by the socket subroutine are unnamed; they are identified only by their address family. Subroutines that connect sockets either assign names or use unnamed sockets.

In the case of a UNIX domain socket, a connect call only succeeds if the process that calls connect has read and write permissions on the socket file created by the bind call. Permissions are determined by the umask value of the process that created the file.

An application program can retrieve the assigned socket name with the getsockname subroutine.

The socket applications can be compiled with COMPAT_43 defined. This will make the sockaddr structure BSD 4.3 compatible. For more details refer to socket.h.

Binding a name in the UNIX domain creates a socket in the file system that must be deleted by the caller when it is no longer needed.

Parameters

Socket Specifies the socket descriptor (an integer) of the socket to be bound.
Name Points to an address structure that specifies the address to which the socket should be bound. The /usr/include/sys/socket.h file defines the sockaddr address structure. The sockaddr structure contains an identifier specific to the address format and protocol provided in the socket subroutine.
NameLength Specifies the length of the socket address structure.

Return Values
Upon successful completion, the bind subroutine returns a value of 0.

If the bind subroutine is unsuccessful, the subroutine handler performs the following actions:
• Returns a value of -1 to the calling program.
Moves an error code, indicating the specific error, into the `errno` global variable. For further explanation of the `errno` variable see “Error Notification Object Class” in AIX 5L Version 5.3 Communications Programming Concepts.

**Error Codes**

The `bind` subroutine is unsuccessful if any of the following errors occurs:

<table>
<thead>
<tr>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EACCES</td>
<td>The requested address is protected, and the current user does not have permission to access it.</td>
</tr>
<tr>
<td>EADDRINUSE</td>
<td>The specified address is already in use.</td>
</tr>
<tr>
<td>EADDRNOTAVAIL</td>
<td>The specified address is not available from the local machine.</td>
</tr>
<tr>
<td>EAFNOSUPPORT</td>
<td>The specified address is not a valid address for the address family of the specified socket.</td>
</tr>
<tr>
<td>EBADF</td>
<td>The <code>Socket</code> parameter is not valid.</td>
</tr>
<tr>
<td>EDESTADDRREQ</td>
<td>The <code>address</code> argument is a null pointer.</td>
</tr>
<tr>
<td>EFAULT</td>
<td>The <code>Address</code> parameter is not in a writable part of the <code>UserAddress</code> space.</td>
</tr>
<tr>
<td>EINVAL</td>
<td>The socket is already bound to an address.</td>
</tr>
<tr>
<td>ENOBUF</td>
<td>Insufficient buffer space available.</td>
</tr>
<tr>
<td>ENODEV</td>
<td>The specified device does not exist.</td>
</tr>
<tr>
<td>ENOTSOCK</td>
<td>The <code>Socket</code> parameter refers to a file, not a socket.</td>
</tr>
<tr>
<td>EOPNOTSUPP</td>
<td>The socket referenced by <code>Socket</code> parameter does not support address binding.</td>
</tr>
</tbody>
</table>

**Examples**

The following program fragment illustrates the use of the `bind` subroutine to bind the name “/tmp/zan/” to a UNIX domain socket.

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

struct sockaddr_un addr;

strcpy(addr.sun_path, "/tmp/zan/");
addr.sun_len = strlen(addr.sun_path);
addr.sun_family = AF_UNIX;
bind(s,(struct sockaddr*)&addr, SUN_LEN(&addr));
```

**Related Information**

The `connect` subroutine, `getsockname` subroutine, `listen` subroutine, `socket` subroutine.

**connect Subroutine**

**Purpose**

Connects two sockets.

**Library**

Standard C Library (`libc.a`)
Syntax

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

int connect (Socket, Name, NameLength);
int Socket;
const struct sockaddr *Name;
socklen_t NameLength;
```

Description

The `connect` subroutine requests a connection between two sockets. The kernel sets up the communication link between the sockets; both sockets must use the same address format and protocol.

If a `connect` subroutine is issued on an unbound socket, the system automatically binds the socket. The `connect` subroutine can be used to connect a socket to itself. This can be done, for example, by binding a socket to a local port (using `bind`) and then connecting it to the same port with a local IP address (using `connect`).

The `connect` subroutine performs a different action for each of the following two types of initiating sockets:

- If the initiating socket is `SOCK_DGRAM`, the `connect` subroutine establishes the peer address. The peer address identifies the socket where all datagrams are sent on subsequent `send` subroutines. No connections are made by this `connect` subroutine.
- If the initiating socket is `SOCK_STREAM` or `SOCK_CONN_DGRAM`, the `connect` subroutine attempts to make a connection to the socket specified by the `Name` parameter. Each communication space interprets the `Name` parameter differently. For `SOCK_CONN_DGRAM` socket type and ATM protocol, some of the ATM parameters may have been modified by the remote station, applications may query new values of ATM parameters using the appropriate socket options.
- In the case of a UNIX domain socket, a `connect` call only succeeds if the process that calls `connect` has read and write permissions on the socket file created by the `bind` call. Permissions are determined by the `umask` value of the process that created the file.

Implementation Specifics

Parameters

- **Socket**: Specifies the unique name of the socket.
- **Name**: Specifies the address of target socket that will form the other end of the communication line.
- **NameLength**: Specifies the length of the address structure.

Return Values

Upon successful completion, the `connect` subroutine returns a value of 0.

If the `connect` subroutine is unsuccessful, the system handler performs the following functions:

- Returns a value of -1 to the calling program.
- Moves an error code, indicating the specific error, into the `errno` global variable.

Error Codes

The `connect` subroutine is unsuccessful if any of the following errors occurs:

<table>
<thead>
<tr>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EADDRINUSE</td>
<td>The specified address is already in use. This error will also occur if the <code>SO_REUSEADDR</code> socket option was set and the local address (whether specified or selected by the system) is already in use.</td>
</tr>
</tbody>
</table>
### Value | Description
--- | ---
EADDRNOTAVAIL | The specified address is not available from the local machine.
EAFNOSUPPORT | The addresses in the specified address family cannot be used with this socket.
EALREADY | The socket is specified with `O_NONBLOCK` or `O_NDELAY`, and a previous connection attempt has not yet completed.
EINTR | The attempt to establish a connection was interrupted by delivery of a signal that was caught; the connection will be established asynchronously.
EACCESS | Search permission is denied on a component of the path prefix or write access to the named socket is denied.
ENOBUFFS | The system ran out of memory for an internal data structure.
EOPNOTSUPP | The socket referenced by `Socket` parameter does not support `connect`.
EWOULDBLOCK | The range allocated for TCP/UDP ephemeral ports has been exhausted.
EBADF | The `Socket` parameter is not valid.
ECONNREFUSED | The attempt to connect was rejected.
EFAULT | The `Address` parameter is not in a writable part of the user address space.
EPROGRESS | The socket is marked as nonblocking. The connection cannot be immediately completed.
EPROTOTYPE | The application program can select the socket for writing during the connection process.
EINVAL | The specified path name contains a character with the high-order bit set.
EISCONN | The socket is already connected.
ENETDOWN | The specified physical network is down.
ENETUNREACH | No route to the network or host is present.
ENOSPC | There is no space left on a device or system table.
ENOTCONN | The socket could not be connected.
ENOTSOCK | The `Socket` parameter refers to a file, not a socket.
ETIMEDOUT | The establishment of a connection timed out out before a connection was made.

### Examples
The following program fragment illustrates the use of the `connect` subroutine by a client to initiate a connection to a server's socket.

```c
struct sockaddr_un server;
.
.
connect(s,(struct sockaddr *)&server, sun_len(&server));
```

### Related Information
- The `accept` subroutine, `bind` subroutine, `getsockname` subroutine, `send` subroutine, `socket` subroutine, `socks5tcp_connect` subroutine.
- [Initiating UNIX Stream Connections Example Program](/LC/MID/1330068866), [Sockets Overview](/LC/MID/1330068868) and [Understanding Socket Connections](/LC/MID/1330068869) in *AIX 5L Version 5.3 Communications Programming Concepts*.

### CreateIoCompletionPort Subroutine

#### Purpose
Creates an I/O completion port with no associated file descriptor or associates an opened socket or file with an existing or newly created I/O completion port.

#### Syntax
```c
#include <iocp.h>
int CreateIoCompletionPort (FileDescriptor, CompletionPort, CompletionKey, ConcurrentThreads)
HANDLE FileDescriptor, CompletionPort;
DWORD CompletionKey, ConcurrentThreads;
```
Description
The `CreateIoCompletionPort` subroutine creates an I/O completion port or associates an open file
descriptor with an existing or newly created I/O completion port. When creating a new I/O completion port,
the `CompletionPort` parameter is set to NULL, the `FileDescriptor` parameter is set to
`INVALID_HANDLE_VALUE` (-1), and the `CompletionKey` parameter is ignored.

The `CreateIoCompletionPort` subroutine returns a descriptor (an integer) to the I/O completion port
created or modified.

The `CreateIoCompletionPort` subroutine is part of the I/O Completion Port (IOCP) kernel extension.

Note: This subroutine only works with file descriptors of sockets, or regular files for use with the
Asynchronous I/O (AIO) subsystem. It does not work with file descriptors of other types.

Parameters
FileDescriptor Specifies a valid file descriptor obtained from a call to the `socket` or `accept` subroutines.

CompletionPort Specifies a valid I/O completion port descriptor. Specifying a `CompletionPort` parameter value of NULL causes the
`CreateIoCompletionPort` subroutine to create a new I/O completion port.

CompletionKey Specifies an integer to serve as the identifier for completion packets generated from a particular
file-completion port set.

ConcurrentThreads This parameter is not implemented and is present only for compatibility purposes.

Return Values
Upon successful completion, the `CreateIoCompletionPort` subroutine returns an integer (the I/O
completion port descriptor).

If the `CreateIoCompletionPort` is unsuccessful, the subroutine handler performs the following functions:
• Returns a value of NULL to the calling program.
• Moves an error code, indicating the specific error, into the `errno` global variable. For further explanation
of the `errno` variable, see the link in the Related Information section of this document.

Error Codes
The `CreateIoCompletionPort` subroutine is unsuccessful if either of the following errors occur:

EBADF The I/O completion port descriptor is invalid.
EINVAL The file descriptor is invalid.

Examples
The following program fragment illustrates the use of the `CreateIoCompletionPort` subroutine to create a
new I/O completion port with no associated file descriptor:
c = CreateIoCompletionPort (INVALID_HANDLE_VALUE, NULL, 0, 0);

The following program fragment illustrates the use of the `CreateIoCompletionPort` subroutine to associate
file descriptor 34 (which has a newly created I/O completion port) with completion key 25:
c = CreateIoCompletionPort (34, NULL, 25, 0);
The following program fragment illustrates the use of the `CreateIoCompletionPort` subroutine to associate file descriptor 54 (which has an existing I/O completion port) with completion key 15:

```
c = CreateIoCompletionPort (54, 12, 15, 0);
```

**Related Information**


For further explanation of the `errno` variable, see Error Notification Object Class in AIX 5L Version 5.3 General Programming Concepts: Writing and Debugging Programs.

---

**dn_comp Subroutine**

**Purpose**

Compresses a domain name.

**Library**

Standard C Library (libc.a)

**Syntax**

```c
#include <sys/types.h>
#include <netinet/in.h>
#include <arpa/nameser.h>
#include <resolv.h>

int dn_comp (ExpDomNam, CompDomNam, Length, DomNamPtr, LastDomNamPtr)
  u_char * ExpDomNam, * CompDomNam;
  int Length;
  u_char ** DomNamPtr, ** LastDomNamPtr;
```

**Description**

The `dn_comp` subroutine compresses a domain name to conserve space. When compressing names, the client process must keep a record of suffixes that have appeared previously. The `dn_comp` subroutine compresses a full domain name by comparing suffixes to a list of previously used suffixes and removing the longest possible suffix.

The `dn_comp` subroutine compresses the domain name pointed to by the `ExpDomNam` parameter and stores it in the area pointed to by the `CompDomNam` parameter. The `dn_comp` subroutine inserts labels into the message as the name is compressed. The `dn_comp` subroutine also maintains a list of pointers to the message labels and updates the list of label pointers.

- If the value of the `DomNamPtr` parameter is null, the `dn_comp` subroutine does not compress any names. The `dn_comp` subroutine translates a domain name from ASCII to internal format without removing suffixes (compressing). Otherwise, the `DomNamPtr` parameter is the address of pointers to previously compressed suffixes.

- If the `LastDomNamPtr` parameter is null, the `dn_comp` subroutine does not update the list of label pointers.

The `dn_comp` subroutine is one of a set of subroutines that form the resolver. The resolver is a set of functions that perform a translation between domain names and network addresses. Global information used by the resolver subroutines resides in the `res` data structure. The `/usr/include/resolv.h` file contains the `res` data structure definition.
All applications containing the **dn_comp** subroutine must be compiled with the **_BSD** macro set to a specific value. Acceptable values are 43 and 44. In addition, all socket applications must include the BSD **libbsd.a** library.

**Parameters**

- **ExpDomNam**: Specifies the address of an expanded domain name.
- **CompDomNam**: Points to an array containing the compressed domain name.
- **Length**: Specifies the size of the array pointed to by the **CompDomNam** parameter.
- **DomNamPtr**: Specifies a list of pointers to previously compressed names in the current message.
- **LastDomNamPtr**: Points to the end of the array specified to by the **CompDomNam** parameter.

**Return Values**

Upon successful completion, the **dn_comp** subroutine returns the size of the compressed domain name.

If unsuccessful, the **dn_comp** subroutine returns a value of -1 to the calling program.

**Files**

/usr/include/resolv.h Contains global information used by the resolver subroutines.

**Related Information**

The **named** daemon.

The **dn_expand**, **getlong**, **getshort**, **putlong**, **putshort** subroutine, **res_init**, **res_mkquery**, **res_query**, **res_send** subroutine, "**res_ninit Subroutine**" on page 165, **res_query** subroutine.

TCP/IP name resolution in *Networks and communication management*.

*Sockets Overview* and *Understanding Domain Name Resolution* in *AIX 5L Version 5.3 Communications Programming Concepts*

---

**dn_expand Subroutine**

**Purpose**

Expands a compressed domain name.

**Library**

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

**Syntax**

```c
#include <sys/types.h>
#include <netinet/in.h>
#include <arpa/nameser.h>
#include <resolv.h>

int dn_expand (MessagePtr, EndOfMesOrig, CompDomNam, ExpandDomNam, Length)
int u_char * MessagePtr, * EndOfMesOrig;
int u_char * CompDomNam, * ExpandDomNam;
int Length;
```
Description

The `dn_expand` subroutine expands a compressed domain name to a full domain name, converting the expanded names to all uppercase letters. A client process compresses domain names to conserve space. Compression consists of removing the longest possible previously occurring suffixes. The `dn_expand` subroutine restores a domain name compressed by the `dn_comp` subroutine to its full size.

The `dn_expand` subroutine is one of a set of subroutines that form the resolver. The resolver is a set of functions that perform a translation between domain names and network addresses. Global information used by the resolver subroutines resides in the `_res` data structure. The `/usr/include/resolv.h` file contains the `_res` data structure definition.

All applications containing the `dn_expand` subroutine must be compiled with the `_BSD` macro set to a specific value. Acceptable values are 43 and 44. In addition, all socket applications must include the BSD `libbsd.a` library.

Parameters

- `MessagePtr` Specifies a pointer to the beginning of a message.
- `EndOfMesOrig` Points to the end of the original message that contains the compressed domain name.
- `CompDomNam` Specifies a pointer to a compressed domain name.
- `ExpandDomNam` Specifies a pointer to a buffer that holds the resulting expanded domain name.
- `Length` Specifies the size of the buffer pointed to by the `ExpandDomNam` parameter.

Return Values

Upon successful completion, the `dn_expand` subroutine returns the size of the expanded domain name. If unsuccessful, the `dn_expand` subroutine returns a value of -1 to the calling program.

Files

`/etc/resolv.conf` Defines name server and domain name constants, structures, and values.

Related Information

The `dn_comp`, `getlong`, `getshort`, `putlong`, `putshort`, `res_init`, `res_mkquery`, `res_search`, `res_send`, `res_query` subroutines, "res_ninit Subroutine" on page 165, `res_send` subroutine.

TCP/IP name resolution in Networks and communication management.

Sockets Overview and Understanding Domain Name Resolution in AIX 5L Version 5.3 Communications Programming Concepts.

endhostent Subroutine

Purpose

Closes the `/etc/hosts` file.
Library

Standard C Library (libc.a)
(libbind)
(libnis)
(liblocal)

Syntax

```c
#include <netdb.h>
endhostent ()
```

Description

When using the `endhostent` subroutine in DNS/BIND name service resolution, `endhostent` closes the TCP connection which the `sethostent` subroutine set up.

When using the `endhostent` subroutine in NIS name resolution or to search the `/etc/hosts` file, `endhostent` closes the `/etc/hosts` file.

Note: If a previous `sethostent` subroutine is performed and the `StayOpen` parameter does not equal 0, the `endhostent` subroutine closes the `/etc/hosts` file. Run a second `sethostent` subroutine with the `StayOpen` value equal to 0 in order for a following `endhostent` subroutine to succeed. Otherwise, the `/etc/hosts` file closes on an `exit` subroutine call.

Files

- `/etc/hosts`
  Contains the host name database.
- `/etc/netsvc.conf`
  Contains the name service ordering.
- `/usr/include/netdb.h`
  Contains the network database structure.

Related Information

The `gethostbyaddr` subroutine, `gethostbyname` subroutine, `sethostent` subroutine, `gethostent` subroutine.

[Sockets Overview](#) and [Network Address Translation](#) in *AIX 5L Version 5.3 Communications Programming Concepts*.

---

**endhostent_r Subroutine**

**Purpose**

Closes the `/etc/hosts` file.

**Library**

Standard C Library (libc.a)
(libbind)
(libnis)
(liblocal)

**Syntax**

```c
#include <netdb.h>

void endhostent_r (struct hostent_data *ht_data);
```
Description
When using the endhostent_r subroutine in DNS/BIND name service resolution, endhostent_r closes the TCP connection which the sethostent_r subroutine set up.

When using the endhostent_r subroutine in NIS name resolution or to search the /etc/hosts file, endhostent_r closes the /etc/hosts file.

Note: If a previous sethostent_r subroutine is performed and the StayOpen parameter does not equal 0, then the endhostent_r subroutine closes the /etc/hosts file. Run a second sethostent_r subroutine with the StayOpen value equal to 0 in order for a following endhostent_r subroutine to succeed. Otherwise, the /etc/hosts file closes on an exit subroutine call.

Parameters
ht_data Points to the hostent_data structure

Files
/etc/hosts Contains the host name database.
/etc/netsvc.conf Contains the name service ordering.
/usr/include/netdb.h Contains the network database structure.

Related Information
“gethostbyaddr_r Subroutine” on page 67, “gethostbyname_r Subroutine” on page 70, “sethostent_r Subroutine” on page 196, and “gethostent_r Subroutine” on page 72.

endnetent Subroutine

Purpose
Closes the /etc/networks file.

Library
Standard C Library (libc.a)

Syntax
#include <netdb.h>
void endnetent ( )

Description
The endnetent subroutine closes the /etc/networks file. Calls made to the getnetent, getnetbyaddr, or getnetbyname subroutine open the /etc/networks file.

All applications containing the endnetent subroutine must be compiled with the _BSD macro set to a specific value. Acceptable values are 43 and 44. In addition, all socket applications must include the BSD libbsd.a library.

Return Values
If a previous setnetent subroutine has been performed and the StayOpen parameter does not equal 0, then the endnetent subroutine will not close the /etc/networks file. Also, the setnetent subroutine does not indicate that it closed the file. A second setnetent subroutine has to be issued with the StayOpen
parameter equal to 0 in order for a following endnetent subroutine to succeed. If this is not done, the /etc/networks file must be closed with the exit subroutine.

Examples
To close the /etc/networks file, type:
endnetent();

Files
/etc/networks Contains official network names.

Related Information
The exit subroutine, getbyaddr subroutine, getbyname subroutine, getent subroutine, setent subroutine.

Sockets Overview, Understanding Network Address Translation, and List of Socket Programming References in AIX 5L Version 5.3 Communications Programming Concepts.

endnetent_r Subroutine

Purpose
Closes the /etc/networks file.

Library
Standard C Library (libc.a)

Syntax
#include <netdb.h>

void endnetent_r (net_data)
struct netent_data *net_data;

Description
The endnetent_r subroutine closes the /etc/networks file. Calls made to the getnetent_r, getbyaddr_r, or getbyname_r subroutine open the /etc/networks file.

Parameters
net_data Points to the netent_data structure.

Files
/etc/networks Contains official network names.

Related Information
“getbyaddr_r Subroutine” on page 77, “getbyname_r Subroutine” on page 79, “getnetent_r Subroutine” on page 81, and “setnetent_r Subroutine” on page 200.
endnetgrent_r Subroutine

Purpose
Handles the group network entries.

Library
Standard C Library (libc.a)

Syntax
```c
#include <netdb.h>
void endnetgrent_r (void **ptr)
```

Description
The setnetgrent_r subroutine establishes the network group from which the getnetgrent_r subroutine will obtain members, and also restarts calls to the getnetgrent_r subroutine from the beginning of the list. If the previous setnetgrent_r call was to a different network group, an endnetgrent_r call is implied.

The endnetgrent_r subroutine frees the space allocated during the getnetgrent_r calls.

Parameters

`ptr` Keeps the function threadsafe.

Files

/etc/netgroup Contains network groups recognized by the system.
/usr/include/netdb.h Contains the network database structures.

Related Information
*“getnetgrent_r Subroutine” on page 82*, and *“setnetgrent_r Subroutine” on page 200.*

endprotoent Subroutine

Purpose
Closes the /etc/protocols file.

Library
Standard C Library (libc.a)

Syntax
```c
void endprotoent (void)
```

Description
The endprotoent subroutine closes the /etc/protocols file. Calls made to the getprotoent subroutine, getprotobyname subroutine, or getprotobynumber subroutine open the /etc/protocols file. An application program can use the endprotoent subroutine to close the /etc/protocols file.
All applications containing the `endprotoent` subroutine must be compiled with the `_BSD` macro set to a specific value. Acceptable values are 43 and 44. In addition, all socket applications must include the BSD `libbsd.a` library.

**Return Values**

If a previous `setprotoent` subroutine has been performed and the `StayOpen` parameter does not equal 0, the `endprotoent` subroutine will not close the `/etc/protocols` file. Also, the `setprotoent` subroutine does not indicate that it closed the file. A second `setprotoent` subroutine has to be issued with the `StayOpen` parameter equal to 0 in order for a following `endprotoent` subroutine to succeed. If this is not done, the `/etc/protocols` file closes on an `exit` subroutine.

**Examples**

To close the `/etc/protocols` file, type:

```c
endprotoent();
```

**Files**

- `/etc/protocols` Contains protocol names.

**Related Information**

The `exit`, `getprotobynumber`, `getprotobyname`, `getprotoent` subroutine, `setprotoent` subroutine.

[Sockets Overview](#) and [Understanding Network Address Translation](#) in *AIX 5L Version 5.3 Communications Programming Concepts*.

---

### endprotoent_r Subroutine

**Purpose**

Closes the `/etc/protocols` file.

**Library**

Standard C Library (`libc.a`)

**Syntax**

```c
void endprotoent_r(proto_data);
struct protoent_data *proto_data;
```

**Description**

The `endprotoent_r` subroutine closes the `/etc/protocols` file, which is opened by the calls made to the `getprotoent_r` subroutine, `getprotobyname_r` subroutine, or `getprotobynumber_r` subroutine.

**Parameters**

- `proto_data` Points to the `protoent_data` structure

**Files**

- `/etc/protocols` Contains protocol names.
Related Information

“getprotobynumber_r Subroutine” on page 88, “getprotobynumber Subroutine” on page 86, “getprotoent_r Subroutine” on page 90, and “setprotoent_r Subroutine” on page 202.

endservent Subroutine

Purpose
Closes the /etc/services file.

Library
Standard C Library (libc.a)

Syntax

```c
#include <netdb.h>
void endservent ( )
```

Description
The endservent subroutine closes the /etc/services file. A call made to the getservent subroutine, getservbyname subroutine, or getservbyport subroutine opens the /etc/services file. An application program can use the endservent subroutine to close the /etc/services file.

All applications containing the endservent subroutine must be compiled with the _BSD macro set to a specific value. Acceptable values are 43 and 44. In addition, all socket applications must include the BSD libbsd.a library.

Return Values
If a previous setservent subroutine has been performed and the StayOpen parameter does not equal 0, then the endservent subroutine will not close the /etc/services file. Also, the setservent subroutine does not indicate that it closed the file. A second setservent subroutine has to be issued with the StayOpen parameter equal to 0 in order for a following endservent subroutine to succeed. If this is not done, the /etc/services file closes on an exit subroutine.

Examples
To close the /etc/services file, type:

```c
endservent ( );
```

Files

/etc/services Contains service names.

Related Information

The endprotoent subroutine, exit subroutine, getprotoent subroutine, getprotobynumber subroutine, getservbyname subroutine, getservbyport subroutine, getprotobynumber subroutine, getprotoent subroutine, setservent subroutine.
endservent_r Subroutine

Purpose
Closes the /etc/services file.

Library
Standard C Library (libc.a)

Syntax
#include <netdb.h>
v o i d  e n d s e r v e n t_r(s e r v _ d a t a)
#include <netdb.h>

Description
The endservent_r subroutine closes the /etc/services file, which is opened by a call made to the
getservent_r subroutine, getservbyname_r subroutine, or getservbyport_r subroutine opens the
/etc/services file.

Parameters
serv_data
Points to the servent_data structure

Examples
To close the /etc/services file, type:
endservent_r(serv_data);

Files
/etc/services
Contains service names.

Related Information
"setservent_r Subroutine" on page 204, "getservent_r Subroutine" on page 98, "getservbyport Subroutine"
on page 94, and "getservbyname_r Subroutine" on page 93.

ether_ntoa, ether_aton, ether_ntohost, ether_hostton, or ether_line
Subroutine

Purpose
Maps 48-bit Ethernet numbers.

Library
Standard C Library (libc.a)

Syntax
#include <sys/types.h>
#include <sys/socket.h>
#include <net/if.h>
#include <netinet/in.h>
#include <netinet/if_ether.h>
#include <netinet/in.h>
#include <netinet/if_ether.h>
char *ether_ntoa (EthernetNumber)
struct ether_addr *EthernetNumber;

struct ether_addr *other_aton(String);
char *string

int *ether_ntohost (HostName, EthernetNumber)
char *HostName;
struct ether_addr *EthernetNumber;

int *ether_hostton (HostName, EthernetNumber)
char *HostName;
struct ether_addr *EthernetNumber;

int *ether_line (Line, EthernetNumber, HostName)
char *Line, *HostName;
struct ether_addr *EthernetNumber;

Description

Attention: Do not use the ether_ntoa or ether_aton subroutine in a multithreaded environment.

The ether_ntoa subroutine maps a 48-bit Ethernet number pointed to by the EthernetNumber parameter to its standard ASCII representation. The subroutine returns a pointer to the ASCII string. The representation is in the form x:x:x:x:x:x where x is a hexadecimal number between 0 and ff. The ether_aton subroutine converts the ASCII string pointed to by the String parameter to a 48-bit Ethernet number. This subroutine returns a null value if the string cannot be scanned correctly.

The ether_ntohost subroutine maps a 48-bit Ethernet number pointed to by the EthernetNumber parameter to its associated host name. The string pointed to by the HostName parameter must be long enough to hold the host name and a null character. The ether_hostton subroutine maps the host name string pointed to by the HostName parameter to its corresponding 48-bit Ethernet number. This subroutine modifies the Ethernet number pointed to by the EthernetNumber parameter.

The ether_line subroutine scans the line pointed to by line and sets the hostname pointed to by the HostName parameter and the Ethernet number pointed to by the EthernetNumber parameter to the information parsed from LINE.

Parameters

EthernetNumber Points to an Ethernet number.
String Points to an ASCII string.
HostName Points to a host name.
Line Points to a line.

Return Values

0 Indicates that the subroutine was successful.
non-zero Indicates that the subroutine was not successful.

Files

/etc/ethers Contains information about the known (48-bit) Ethernet addresses of hosts on the Internet.
FrcaCacheCreate Subroutine

Purpose
Creates a cache instance within the scope of a Fast Response Cache Accelerator (FRCA) instance.

Library
FRCA Library (libfrca.a)

Syntax
```c
#include <frca.h>
int32_t FrcaCacheCreate (CacheHandle, FrcaHandle, CacheSpec);
```

Description
The FrcaCacheCreate subroutine creates a cache instance for an FRCA instance that has already been configured. Multiple caches can be created for an FRCA instance. Cache handles are unique only within the scope of the FRCA instance.

Parameters
- **CacheHandle**: Returns a handle that is required by the other cache-related subroutines of the FRCA API to refer to the newly created FRCA cache instance.
- **FrcaHandle**: Identifies the FRCA instance for which the cache is created.
- **CacheSpec**: Points to a frca_ctrl_create_t structure, which specifies the characteristics of the cache to be created. The structure contains the following members:
  - **cacheType**: Specifies the type of the cache instance. This field must be set to FCTRL_SERVERTYPE_HTTP.
  - **nMaxEntries**: Specifies the maximum number of entries allowed for the cache instance.

Note: Structure members do not necessarily appear in this order.

Return Values
- **0**: The subroutine completed successfully.
- **-1**: The subroutine failed. The global variable `errno` is set to indicate the specific type of error.
Error Codes

EINVAL
The CacheHandle or the CacheSpec parameter is zero or the CacheSpec parameter is not of the correct type FCTRL_CACHETYPE_HTTP.

EFAULT
The CacheHandle or the CacheSpec point to an invalid address.

ENOENT
The FrcaHandle parameter is invalid.

Related Information
The FrcaCacheDelete subroutine, the FrcaCacheLoadFile subroutine, the FrcaCacheUnloadFile subroutine, the FrcaCtrlCreate subroutine, the FrcaCtrlDelete subroutine, the FrcaCtrlLog subroutine, the FrcaCtrlStart subroutine, the FrcaCtrlStop subroutine.

FrcaCacheDelete Subroutine

Purpose
Deletes a cache instance within the scope of a Fast Response Cache Accelerator (FRCA) instance.

Library
FRCA Library (libfrca.a)

Syntax
#include <frca.h>
int32_t FrcaCacheDelete (CacheHandle, FrcaHandle);

Description
The FrcaCacheDelete subroutine deletes a cache instance and releases any associated resources.

Parameters
CacheHandle
Identifies the cache instance that is to be deleted.
FrcaHandle
Identifies the FRCA instance to which the cache instance belongs.

Return Values
0
The subroutine completed successfully.
-1
The subroutine failed. The global variable errno is set to indicate the specific type of error.

Error Codes

ENOENT
The CacheHandle or the FrcaHandle parameter is invalid.
Related Information

The FrcaCacheCreate subroutine, the FrcaCacheLoadFile subroutine, the FrcaCacheUnloadFile subroutine, the FrcaCtrlCreate subroutine, the FrcaCtrlDelete subroutine, the FrcaCtrlLog subroutine, the FrcaCtrlStart subroutine, and the FrcaCtrlStop subroutine.

**FrcaCacheLoadFile Subroutine**

**Purpose**

Loads a file into a cache associated with a Fast Response Cache Accelerator (FRCA) instance.

**Library**

FRCA Library (libfrca.a)

**Syntax**

```c
#include <frca.h>

int32_t FrcaCacheLoadFile (CacheHandle, FrcaHandle, FileSpec, AssocData);
```

**Description**

The `FrcaCacheLoadFile` subroutine loads a file into an existing cache instance for an previously configured FRCA instance.

**Parameters**

- **CacheHandle** Identifies the cache instance to which the new entry should be added.
- **FrcaHandle** Identifies the FRCA instance to which the cache instance belongs.
FileSpec

Points to a `frca_loadfile_t` structure, which specifies characteristics used to identify the cache entry that is to be loaded into the given cache. The structure contains the following members:

```c
uint32_t cacheEntryType;
char *fileName;
char *virtualHost;
char *searchKey;
```

**Note:** Structure members do not necessarily appear in this order.

- **cacheEntryType**
  Specifies the type of the cache entry. This field must be set to `FCTRL_CET_HTTPFILE`.

- **fileName**
  Specifies the absolute path to the file that is providing the contents for the new cache entry.

- **virtualHost**
  Specifies a virtual host name that is being served by the FRCA instance.

- **searchKey**
  Specifies the key that the cache entry can be found under by the FRCA instance when it processes an intercepted request. For the HTTP GET engine, the search key is identical to the `abs_path` part of the HTTP URL according to section 3.2.2 of RFC 2616. For example, the search key corresponding to the URL `http://www.mydomain/welcome.html` is `/welcome.html`.

**Note:** If a cache entry with the same type, file name, virtual host, and search key already exists and the file has not been modified since the existing entry was created, the load request succeeds without any effect. If the entry exists and the file’s contents have been modified since being loaded into the cache, the cache entry is updated. If the entry exists and the file’s contents have not changed, but any of the settings of the HTTP header fields change, the existing entry must be unloaded first.

AssocData

Points to a `frca_assocdata_t` structure, which specifies additional information to be associated with the contents of the given cache entry. The structure contains the following members:

```c
uint32_t assocDataType;
char *cacheControl;
char *contentType;
char *contentEncoding;
char *contentLanguage;
char *contentCharset;
```

**Note:** Structure members do not necessarily appear in this order.

- **assocDataType**
  Specifies the type of data that is associated with the given cache entry.

- **cacheControl**
  Specifies the settings of the corresponding HTTP header field according to RFC 2616.

- **contentType**
  Specifies the settings of the corresponding HTTP header field according to RFC 2616.

- **contentEncoding**
  Specifies the settings of the corresponding HTTP header field according to RFC 2616.

- **contentLanguage**
  Specifies the settings of the corresponding HTTP header field according to RFC 2616.

- **contentCharset**
  Specifies the settings of the corresponding HTTP header field according to RFC 2616.
Return Values

0  The subroutine completed successfully.
-1  The subroutine failed. The global variable *errno* is set to indicate the specific type of error.

Error Codes

**EINVAL**  The *FileSpec* or the *AssocData* parameter is zero or are not of the correct type or any of the *fileName* or the *searchKey* components are zero or the size of the file is zero.

**EFAULT**  The *FileSpec* or the *AssocData* parameter or one of their components points to an invalid address.

**ENOMEM**  The FRCA or NBC subsystem is out of memory.

**EFBIG**  The content of the cache entry failed to load into the NBC. Check network options *nbc_limit*, *nbc_min_cache*, and *nbc_max_cache*.

**ENOTREADY**  The kernel extension is currently being loaded or unloaded.

**ENOENT**  The *CacheHandle* or the *FrcaHandle* parameter is invalid.

Related Information

The **FrcaCacheCreate** subroutine, **FrcaCacheDelete** subroutine, **FrcaCacheUnloadFile** subroutine, **FrcaCtrlCreate** subroutine, **FrcaCtrlDelete** subroutine, **FrcaCtrlLog** subroutine, **FrcaCtrlStart** subroutine, **FrcaCtrlStop** subroutine.

---

**FrcaCacheUnoadFile Subroutine**

**Purpose**

Removes a cache entry from a cache that is associated with a Fast Response Cache Accelerator (FRCA) instance.

**Library**

FRCA Library (*libfrca.a*)

**Syntax**

```c
#include <frca.h>
int32_t FrcaCacheUnloadFile (CacheHandle, FrcaHandle, FileSpec);
```

**Parameters**

- **CacheHandle**  Identifies the cache instance from which the entry should be removed.
- **FrcaHandle**  Identifies the FRCA instance to which the cache instance belongs.
FileSpec

Points to a `frca_loadfile_t` structure, which specifies characteristics used to identify the cache entry that is to be removed from the given cache. The structure contains the following members:

- `uint32_t cacheEntryType`:
  Specifies the type of the cache entry. This field must be set to `FCTRL_CET_HTTPFILE`.
- `char * fileName`:
  Specifies the absolute path to the file that is to be removed from the cache.
- `char * virtualHost`:
  Specifies a virtual host name that is being served by the FRCA instance.
- `char * searchKey`:
  Specifies the key under which the cache entry can be found.

Note: Structure members do not necessarily appear in this order.

Note: The FrcaCacheUnloadFile subroutine succeeds if a cache entry with the same type, file name, virtual host, and search key does not exist. This subroutine fails if the file associated with `fileName` does not exist or if the calling process does not have sufficient access permissions.

Return Values

- `0` The subroutine completed successfully.
- `-1` The subroutine failed. The global variable `errno` is set to indicate the specific type of error.

Error Codes

- **EINVAL**
  The FileSpec parameter is zero or the cacheEntryType component is not set to `FCTRL_CET_HTTPFILE` or the searchKey component is zero or the `fileName` is `/` or the `fileName` is not an absolute path.

- **EFAULT**
  The FileSpec parameter or one of the components points to an invalid address.

- **EACCES**
  Access permission is denied on the `fileName`.

Related Information

The FrcaCacheCreate subroutine, the FrcaCacheDelete subroutine, the FrcaCacheLoadFile subroutine, the FrcaCtrlCreate subroutine, the FrcaCtrlDelete subroutine, the FrcaCtrlLog subroutine, the FrcaCtrlStart subroutine, the FrcaCtrlStop subroutine.

FrcaCtrlCreate Subroutine

**Purpose**

Creates a Fast Response Cache Accelerator (FRCA) control instance.
Library
FRCA Library (libfrca.a)

Syntax
#include <frca.h>
int32_t FrcaCtrlCreate ( FrcaHandle, InstanceSpec );
int32_t *FrcaHandle;
frca_ctrl_create_t *InstanceSpec;

Description
The FrcaCtrlCreate subroutine creates and configures an FRCA instance that is associated with a previously configured TCP listen socket. TCP connections derived from the TCP listen socket are intercepted by the FRCA instance and, if applicable, adequate responses are generated by the in-kernel code on behalf of the user-level application.

The only FRCA instance type that is currently supported handles static GET requests as part of the Hypertext Transfer Protocol (HTTP).

Parameters
FrcaHandle Returns a handle that is required by the other FRCA API subroutines to refer to the newly configured FRCA instance.
Points to a `frca_ctrl_create_t` structure, which specifies the parameters used to configure the newly created FRCA instance. The structure contains the following members:

```c
uint32_t serverType;
char * serverName;
uint32_t * listenSockets;
uint32_t nListenSockets;
uint32_t flags;
uint32_t nMaxConnections;
uint32_t nLogBufs;
char * logFile;
```

**Note:** Structure members do not necessarily appear in this order.

- **serverType**
  Specifies the type for the FRCA instance. This field must be set to `FCTRL_SERVERTYPE_HTTP`.

- **serverName**
  Specifies the value to which the HTTP header field is set.

- **nListenSocket**
  Specifies the number of listen socket descriptors pointed to by `listenSockets`.

- **listenSocket**
  Specifies the TCP listen socket that the FRCA instance should be configured to intercept. The TCP listen socket must exist and the `SO_KERNACCEPT` socket option must be set at the time of calling the `FrcaCtrlCreate` subroutine.

- **flags**
  Specifies the logging format, the initial state of the logging subsystem, and whether responses generated by the FRCA instance should include the `Server: HTTP` header field. The valid flags are as follows:
  - `FCTRL_KEEPALIVE`
  - `FCTRL_LOGFORMAT`
  - `FCTRL_LOGFORMAT_ECLF`
  - `FCTRL_LOGFORMAT_VHOST`
  - `FCTRL_LOGMODE`
  - `FCTRL_LOGMODE_ON`
  - `FCTRL_SENDSERVERHEADER`

- **nMaxConnections**
  Specifies the maximum number of intercepted connections that are allowed at any given point in time.

- **nLogBufs**
  Specifies the number of preallocated logging buffers used for logging information about HTTP GET requests that have been served successfully.

- **logFile**
  Specifies the absolute path to a file used for appending logging information. The HTTP GET engine uses `logFile` as a base name and appends a sequence number to it to generate the actual file name. Whenever the size of the current log file exceeds the threshold of approximately 1 gigabyte, the sequence number is incremented by 1 and the logging subsystem starts appending to the new log file. **Note:** The FRCA instance creates the log file, but not the path to it. If the path does not exist or is not accessible, the FRCA instance reverts to the default log file `/tmp/frca.log`.

**Return Values**

- **0** The subroutine completed successfully.
The subroutine failed. The global variable *errno* is set to indicate the specific type of error.

### Error Codes

**EINVAL**

The `FrcaHandle` or the `InstanceSpec` parameter is zero or is not of the correct type or the `listenSockets` components do not specify any socket descriptors.

**EFAULT**

The `FrcaHandle` or the `InstanceSpec` or a component of the `InstanceSpec` points to an invalid address.

**ENOTREADY**

The kernel extension is currently being loaded or unloaded.

**ENOTSOCK**

A TCP listen socket does not exist.

### Related Information

The `FrcaCacheCreate` subroutine, the `FrcaCacheDelete` subroutine, the `FrcaCacheLoadFile` subroutine, the `FrcaCacheUnloadFile` subroutine, the `FrcaCtrlDelete` subroutine, the `FrcaCtrlLog` subroutine, the `FrcaCtrlStart` subroutine, the `FrcaCtrlStop` subroutine.

---

**FrcaCtrlDelete Subroutine**

### Purpose

Deletes a Fast Response Cache Accelerator (FRCA) control instance.

### Library

FRCA Library (libfrca.a)

### Syntax

```c
#include <frca.h>
int32_t FrcaCtrlDelete (FrcaHandle);
int32_t *FrcaHandle;
```

### Description

The `FrcaCtrlDelete` subroutine deletes an FRCA instance and releases any associated resources.

The only FRCA instance type that is currently supported handles static GET requests as part of the Hypertext Transfer Protocol (HTTP).

### Parameters

`FrcaHandle` Identifies the FRCA instance on which this operation is performed.

### Return Values

<table>
<thead>
<tr>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>The subroutine completed successfully.</td>
</tr>
<tr>
<td>-1</td>
<td>The subroutine failed. The global variable <em>errno</em> is set to indicate the specific type of error.</td>
</tr>
</tbody>
</table>
Error Codes

ENOENT    The FrcaHandle parameter is invalid.
ENOTREADY The FRCA control instance is in an undefined state.

Related Information

The FrcaCacheCreate subroutine, the FrcaCacheDelete subroutine, the FrcaCacheLoadFile subroutine, the FrcaCacheUnloadFile subroutine, the FrcaCtrlCreate subroutine, the FrcaCtrlLog subroutine, the FrcaCtrlStart subroutine, the FrcaCtrlStop subroutine.

FrcaCtrlLog Subroutine

Purpose

Modifies the behavior of the logging subsystem.

Library

FRCA Library (libfrca.a)

Syntax

```
#include <frca.h>
int32_t FrcaCtrlLog ( FrcaHandle, Flags
int32_t FrcaHandle;
uint32_t Flags;
```

Description

The FrcaCtrlLog subroutine modifies the behavior of the logging subsystem for the Fast Response Cache Accelerator (FRCA) instance specified. Modifiable attributes are the logging mode, which can be turned on or off, and the logging format, which defaults to the HTTP Common Log Format (CLF). The logging format can be changed to Extended Common Log Format (ECLF) and can be set to include virtual host information.

The only FRCA instance type that is currently supported handles static GET requests as part of the Hypertext Transfer Protocol (HTTP).

Parameters

- FrcaHandle: Returns a handle that is required by the other FRCA API subroutines to refer to the newly configured FRCA instance.
- Flags: Specifies the behavior of the logging subsystem. The parameter value is constructed by logically ORing single flags. The valid flags are as follows:
  - FCTRL_LOGFORMAT
  - FCTRL_LOGFORMAT_ECLF
  - FCTRL_LOGFORMAT_VHOST
  - FCTRL_LOGMODE
  - FCTRL_LOGMODE_ON
Return Values

<table>
<thead>
<tr>
<th>Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>The subroutine completed successfully.</td>
</tr>
<tr>
<td>-1</td>
<td>The subroutine failed. The global variable <code>errno</code> is set to indicate the specific type of error.</td>
</tr>
</tbody>
</table>

Error Codes

ENOTREADY  The kernel extension is currently being loaded or unloaded.

Related Information

The `FrcaCacheCreate` subroutine, the `FrcaCacheDelete` subroutine, the `FrcaCacheLoadFile` subroutine, the `FrcaCacheUnloadFile` subroutine, the `FrcaCtrlCreate` subroutine, the `FrcaCtrlDelete` subroutine, the `FrcaCtrlStart` subroutine, the `FrcaCtrlStop` subroutine.

FrcaCtrlStart Subroutine

Purpose

Starts the interception of TCP data connections for a previously configured Fast Response Cache Accelerator (FRCA) instance.

Library

FRCA Library (`libfrca.a`)

Syntax

```
#include <frca.h>
int32_t FrcaCtrlStart (FrcaHandle);
int32_t *FrcaHandle;
```

Description

The `FrcaCtrlStart` subroutine starts the interception of TCP data connections for an FRCA instance. If the FRCA instance cannot handle the data on that connection, it passes the data to the user-level application that has established the listen socket.

The only FRCA instance type that is currently supported handles static GET requests as part of the Hypertext Transfer Protocol (HTTP).

Parameters

`FrcaHandle` Identifies the FRCA instance on which this operation is performed.

Return Values

<table>
<thead>
<tr>
<th>Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>The subroutine completed successfully.</td>
</tr>
<tr>
<td>-1</td>
<td>The subroutine failed. The global variable <code>errno</code> is set to indicate the specific type of error.</td>
</tr>
</tbody>
</table>
Error Codes

ENOENT The FrcaHandle parameter is invalid.
ENOTREADY The FRCA control instance is in an undefined state.
ENOTSOCK A TCP listen socket that was passed in with the FrcaCtrlCreate subroutine cannot be intercepted because it does not exist.

Related Information

The FrcaCacheCreate subroutine, the FrcaCacheDelete subroutine, the FrcaCacheLoadFile subroutine, the FrcaCacheUnloadFile subroutine, the FrcaCtrlCreate subroutine, the FrcaCtrlDelete subroutine, the FrcaCtrlLog subroutine, the FrcaCtrlStop subroutine.

FrcaCtrlStop Subroutine

Purpose

Stops the interception of TCP data connections for a Fast Response Cache Accelerator (FRCA) instance.

Library

FRCA Library (libfrca.a)

Syntax

#include <frca.h>
int32_t FrcaCtrlStop (FrcaHandle);
int32_t *FrcaHandle;

Description

The FrcaCtrlStop subroutine stops the interception of newly arriving TCP data connections for a previously configured FRCA instance. Connection requests are passed to the user-level application that has established the listen socket.

The only FRCA instance type that is currently supported handles static GET requests as part of the Hypertext Transfer Protocol (HTTP).

Parameters

FrcaHandle Identifies the FRCA instance on which this operation is performed.

Return Values

0 The subroutine completed successfully.
-1 The subroutine failed. The global variable errno is set to indicate the specific type of error.

Error Codes

ENOENT The FrcaHandle parameter is invalid.
ENOTREADY The FRCA control instance has not been started yet.
Related Information

The FrcaCacheCreate subroutine, the FrcaCacheDelete subroutine, the FrcaCacheLoadFile subroutine, the FrcaCacheUnloadFile subroutine, the FrcaCtrlCreate subroutine, the FrcaCtrlDelete subroutine, the FrcaCtrlLog subroutine, the FrcaCtrlStart subroutine.

freeaddrinfo Subroutine

**Purpose**
Frees memory allocated by the “getaddrinfo Subroutine.”

**Library**
The Standard C Library (<libc.a>)

**Syntax**
```c
#include <sys/socket.h>
#include <netdb.h>
void freeaddrinfo (struct addrinfo *ai)
```

**Description**
The freeaddrinfo subroutine frees one or more addrinfo structures returned by the getaddrinfo subroutine, along with any additional storage associated with those structures. If the ai_next field of the structure is not NULL, the entire list of structures is freed.

**Parameters**

- `ai` Points to dynamic storage allocated by the getaddrinfo subroutine

Related Information

“getaddrinfo Subroutine,” and “getnameinfo Subroutine” on page 75.


getaddrinfo Subroutine

**Purpose**
Protocol-independent hostname-to-address translation.

**Library**
Library (libc.a)

**Syntax**
```c
#include <sys/socket.h>
#include <netdb.h>
int getaddrinfo (hostname, servname, hints, res)
const char *hostname;
const char *servname;
const struct addrinfo *hints;
struct addrinfo **res;
```
Description

The `hostname` and `servname` parameters describe the hostname and/or service name to be referenced. Zero or one of these arguments may be NULL. A non-NULL hostname may be either a hostname or a numeric host address string (a dotted-decimal for IPv4 or hex for IPv6). A non-NULL servname may be either a service name or a decimal port number.

The `hints` parameter specifies hints concerning the desired return information. The `hostname` and `servname` parameters are pointers to null-terminated strings or NULL. One or both of these arguments must be a non-NULL pointer. In a normal client scenario, both the `hostname` and `servname` parameters are specified. In the normal server scenario, only the `servname` parameter is specified. A non-NULL hostname string can be either a host name or a numeric host address string (for example, a dotted-decimal IPv4 address or an IPv6 hex address). A non-NULL `servname` string can be either a service name or a decimal port number.

The caller can optionally pass an `addrinfo` structure, pointed to by the `hints` parameter, to provide hints concerning the type of socket that the caller supports. In this `hints` structure, all members other than `ai_flags`, `ai_family`, `ai_socktype`, and `ai_protocol` must be zero or a NULL pointer. A value of `PF_UNSPEC` for `ai_family` means the caller will accept any protocol family. A value of zero for `ai_socktype` means the caller will accept any socket type. A value of zero for `ai_protocol` means the caller will accept any protocol. For example, if the caller handles only TCP and not UDP, the `ai_socktype` member of the `hints` structure should be set to `SOCK_STREAM` when the `getaddrinfo` subroutine is called. If the caller handles only IPv4 and not IPv6, the `ai_family` member of the `hints` structure should be set to `PF_INET` when `getaddrinfo` is called. If the `hints` parameter in `getaddrinfo` is a NULL pointer, it is the same as if the caller fills in an `addrinfo` structure initialized to zero with `ai_family` set to `PF_UNSPEC`.

Upon successful return, a pointer to a linked list of one or more `addrinfo` structures is returned through the `res` parameter. The caller can process each `addrinfo` structure in this list by following the `ai_next` pointer, until a NULL pointer is encountered. In each returned `addrinfo` structure the three members `ai_family`, `ai_socktype`, and `ai_protocol` are the corresponding arguments for a call to the `socket` subroutine. In each `addrinfo` structure, the `ai_addr` member points to a filled-in socket address structure whose length is specified by the `ai_addrlen` member.

If the `AI_PASSIVE` bit is set in the `ai_flags` member of the `hints` structure, the caller plans to use the returned socket address structure in a call to the `bind` subroutine. If the `hostname` parameter is a NULL pointer, the IP address portion of the socket address structure will be set to `INADDR_ANY` for an IPv4 address or `IN6ADDR_ANY_INIT` for an IPv6 address.

If the `AI_PASSIVE` bit is not set in the `ai_flags` member of the `hints` structure, the returned socket address structure will be ready for a call to the `connect` subroutine (for a connection-oriented protocol) or the `connect`, `sendto`, or `sendmsg` subroutine (for a connectionless protocol). If the `hostname` parameter is a NULL pointer, the IP address portion of the socket address structure will be set to the loopback address.

If the `AI_CANONNAME` bit is set in the `ai_flags` member of the `hints` structure, upon successful return the `ai_canonname` member of the first `addrinfo` structure in the linked list will point to a NULL-terminated string containing the canonical name of the specified hostname.

If the `AI_NUMERICHOST` flag is specified, a non-NULL nodename string supplied is a numeric host address string. Otherwise, an (EAI_NONAME) error is returned. This flag prevents any type of name resolution service (such as, DNS) from being invoked.

If the `AI_NUMERICSERV` flag is specified, a non-NULL `servname` string supplied is a numeric port string. Otherwise, an (EAI_NONAME) error is returned. This flag prevents any type of name resolution service (such as, NIS+) from being invoked.

62 Technical Reference: Communications, Volume 2
If the AI_V4MAPPED flag is specified along with an ai_family value of AF_INET6, the getaddrinfo subroutine returns IPv4-mapped IPv6 addresses when no matching IPv6 addresses (ai_addrlen is 16) are found. For example, when using DNS, if no AAAA or A6 records are found, a query is made for A records. Any found are returned as IPv4-mapped IPv6 addresses. The AI_V4MAPPED flag is ignored unless ai_family equals AF_INET6.

If the AI_ALL flag is used with the AI_V4MAPPED flag, the getaddrinfo subroutine returns all matching IPv6 and IPv4 addresses. For example, when using DNS, a query is first made for AAAA/A6 records. If successful, those IPv6 addresses are returned. Another query is made for A records, and any IPv4 addresses found are returned as IPv4-mapped IPv6 addresses. The AI_ALL flag without the AI_V4MAPPED flag is ignored.

Note: When ai_family is not specified (AF_UNSPEC), AI_V4MAPPED and AI_ALL flags will only be used if AF_INET6 is supported.

If the AI_ADDRCONFIG flag is specified, a query for AAAA or A6 records should occur only if the node has at least one IPv6 source address configured. A query for A records should occur only if the node has at least one IPv4 source address configured. The loopback address is not considered valid as a configured source address.

All of the information returned by the getaddrinfo subroutine is dynamically allocated: the addrinfo structures, the socket address structures, and canonical host name strings pointed to by the addrinfo structures. To return this information to the system, "freeaddrinfo Subroutine" on page 61 is called.

The addrinfo structure is defined as:

```c
struct addrinfo {
    int ai_flags;         /* AI_PASSIVE, AI_CANONNAME */
    int ai_family;        /* PF_xxx */
    int ai_socktype;      /* SOCK_xxx */
    int ai_protocol;      /* 0 or IP=PROTO_xxx for IPv4 and IPv6 */
    size_t ai_addrlen;    /* length of ai_addr */
    char *ai_canonname;   /* canonical name for hostname */
    struct sockaddr *ai_addr;   /* binary address */
    struct addrinfo *ai_next; /* next structure in linked list */
};
```

Return Values
If the query is successful, a pointer to a linked list of one or more addrinfo structures is returned via the res parameter. A zero return value indicates success. If the query fails, a non-zero error code will be returned.

Error Codes
The following names are the non-zero error codes. See netdb.h for further definition.

- **EAI_ADDRFAMILY**: Address family for hostname not supported
- **EAI_AGAIN**: Temporary failure in name resolution
- **EAI_BADFLAGS**: Invalid value for ai_flags
- **EAI_FAIL**: Non-recoverable failure in name resolution
- **EAI_FAMILY**: ai_family not supported
- **EAI_MEMORY**: Memory allocation failure
- **EAI_NODATA**: No address associated with hostname
- **EAI_NONAME**: No hostname or servname provided, or not known
- **EAI_SERVICE**: servname not supported for ai_socktype
- **EAI_SOCKTYPE**: ai_socktype not supported
- **EAI_SYSTEM**: System error returned in errno
get_auth_method Subroutine

Purpose
Returns the list of authentication methods for the secure rcmds.

Library
Authentication Methods Library (libauthm.a)

Syntax

Description
This method returns the authentication methods currently configured in the order in which they should be attempted in the unsigned integer pointer the user passed in.

The list in the unsigned integer pointer is either NULL (on an error) or is an array of unsigned integers terminated by a zero. Each integer identifies an authentication method. The order that a client should attempt to authenticate is defined by the order of the list.

Note: The calling routine is responsible for freeing the memory in which the list is contained.

The flags identifying the authentication methods are defined in the /usr/include/authm.h file.

Parameter

authm Points to an array of unsigned integers. The list of authentication methods is returned in the zero terminated list.

Return Values
Upon successful completion, the get_auth_method subroutine returns a zero.

Upon unsuccessful completion, the get_auth_method subroutine returns an errno.

Related Information
The chauthent command, ftp command, lsauthent command, rcp command, rlogin command, rsh command, telnet, tn, or tn3270 command.
The set_auth_method subroutine.

Communications and networks in Networks and communication management.

Authentication and the secure rcmds in Networks and communication management.
getdomainname Subroutine

Purpose
Gets the name of the current domain.

Library
Standard C Library (libc.a)

Syntax
```c
int getdomainname (Name, Namelen)
char *Name;
int Namelen;
```

Description
The `getdomainname` subroutine returns the name of the domain for the current processor as previously set by the `setdomainname` subroutine. The returned name is null-terminated unless insufficient space is provided.

The purpose of domains is to enable two distinct networks that may have host names in common to merge. Each network would be distinguished by having a different domain name. Only the Network Information Service (NIS) and the `sendmail` command make use of domains.

All applications containing the `getdomainname` subroutine must be compiled with the `_BSD` macro set to a specific value. Acceptable values are 43 and 44. In addition, all socket applications must include the BSD `libbsd.a` library.

Note: Domain names are restricted to 256 characters.

Parameters

- **Name**: Specifies the domain name to be returned.
- **Namelen**: Specifies the size of the array pointed to by the `Name` parameter.

Return Values
If the call succeeds, a value of 0 is returned. If the call is unsuccessful, a value of -1 is returned and an error code is placed in the `errno` global variable.

Error Codes
The following error may be returned by this subroutine:

<table>
<thead>
<tr>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EFAULT</td>
<td>The <code>Name</code> parameter gave an invalid address.</td>
</tr>
</tbody>
</table>

Related Information
The `gethostname` subroutine, `setdomainname` subroutine, `sethostname` subroutine.

Sockets Overview in *AIX 5L Version 5.3 Communications Programming Concepts.*
gethostbyaddr Subroutine

Purpose
Gets network host entry by address.

Library
Standard C Library (libc.a)
(libbind)
(libnis)
(liblocal)

Syntax
#include <netdb.h>

struct hostent *gethostbyaddr (Address, Length, Type)
const void *Address, size_t Length, int Type;

Description
The gethostbyaddr subroutine is threadsafe in AIX 4.3 and later. However, the return value points to static data that is overwritten by subsequent calls. This data must be copied to be saved for use by subsequent calls.

The gethostbyaddr subroutine retrieves information about a host using the host address as a search key. Unless specified, the gethostbyaddr subroutine uses the default name services ordering, that is, it will query DNS/BIND, NIS, then the local /etc/hosts file.

When using DNS/BIND name service resolution, if the file /etc/resolv.conf exists, the gethostbyaddr subroutine queries the domain name server. The gethostbyaddr subroutine recognizes domain name servers as described in RFC 883.

When using NIS for name resolution, if the getdomainname subroutine is successful and yp_bind indicates NIS is running, then the gethostbyaddr subroutine queries NIS.

The gethostbyaddr subroutine also searches the local /etc/hosts file when indicated to do so.

The gethostbyaddr subroutine returns a pointer to a hostent structure, which contains information obtained from one of the name resolutions services. The hostent structure is defined in the netdb.h file.

The environment variable, NSORDER can be set to override the default name services ordering and the order specified in the /etc/netsvc.conf file.

Parameters
Address Specifies a host address. The host address is passed as a pointer to the binary format address.
Length Specifies the length of host address.
Type Specifies the domain type of the host address. It can be either AF_INET or AF_INET6.

Return Values
The gethostbyaddr subroutine returns a pointer to a hostent structure upon success.

If an error occurs or if the end of the file is reached, the gethostbyaddr subroutine returns a NULL pointer and sets h_errno to indicate the error.
Error Codes

The `gethostbyaddr` subroutine is unsuccessful if any of the following errors occur:

<table>
<thead>
<tr>
<th>Error</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>HOST_NOT_FOUND</td>
<td>The host specified by the Name parameter is not found.</td>
</tr>
<tr>
<td>TRY_AGAIN</td>
<td>The local server does 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>
<tr>
<td>NO_ADDRESS</td>
<td>The requested Address parameter is valid but does not have a name at the name server.</td>
</tr>
<tr>
<td>SERVICE_UNAVAILABLE</td>
<td>None of the name services specified are running or available.</td>
</tr>
</tbody>
</table>

Files

- `/etc/hosts` Contains the host-name database.
- `/etc/resolv.conf` Contains the name server and domain name information.
- `/etc/netsvc.conf` Contains the name of the services ordering.
- `/usr/include/netdb.h` Contains the network database structure.

Related Information

The `endhostent` subroutine, `gethostbyname` subroutine, `sethostent` subroutine, `gethostent` subroutine, `inet_addr` subroutine.

Sockets Overview and Network Address Translation in AIX 5L Version 5.3 Communications Programming Concepts.

gethostbyaddr_r Subroutine

Purpose

Gets network host entry by address.

Library

Standard C Library (`libc.a`)
(`libbind`)
(`libnis`)
(`liblocal`)

Syntax

```c
#include <netdb.h>
int gethostbyadd_r(Addr, Len, Type, Htent, Ht_data)
const char *Addr, size_t Len, int Type, struct hostent *Htent, struct hostent_data *Ht_data;
```

Description

This function internally calls the `gethostbyaddr` subroutine and stores the value returned by the `gethostbyaddr` subroutine to the hostent structure.

Parameters

- `Addr` Points to the host address that is a pointer to the binary format address.
- `Len` Specifies the length of the address.
**Type**
Specifies the domain type of the host address. It can be either **AF_INET** or **AF_INET6**.

**Htent**
Points to a hostent structure which is used to store the return value of the *gethostaddr* subroutine.

**Ht_data**
Points to a *hostent_data* structure.

**Return Values**
The function returns a 0 if successful and a -1 if unsuccessful.

**Note:** The return value of the *gethostbyaddr* subroutine points to static data that is overwritten by subsequent calls. This data must be copied at every call to be saved for use by subsequent calls. The *gethostbyaddr_r* subroutine solves this problem.

If the *Name* parameter is a *hostname*, this subroutine searches for a machine with that name as an IP address. Because of this, use the *gethostbyname_r* subroutine.

**Error Codes**
The *gethostbyaddr_r* subroutine is unsuccessful if any of the following errors occur:

- **HOST_NOT_FOUND**
  The host specified by the *Name* parameter was not found.
- **TRY_AGAIN**
  The local server did not receive a response from an authoritative server. Try again later.
- **NO_RECOVERY**
  Indicates an unrecoverable error occurred.
- **NO_ADDRESS**
  The requested *Name* parameter is valid but does not have an Internet address at the name server.
- **SERVICE_UNAVAILABLE**
  None of the name services specified are running or available.
- **EINVAL**
  The hostent pointer is NULL.

**Files**
- **/etc/hosts**
  Contains the host name data base.
- **/etc/resolv.conf**
  Contains the name server and domain name.
- **/etc/netsvc.conf**
  Contains the name services ordering.
- **/usr/include/netdb.h**
  Contains the network database structure.

**Related Information**
- [“endhostent_r Subroutine” on page 41](#)
- [“gethostbyaddr_r Subroutine” on page 67](#)
- [“gethostent_r Subroutine” on page 72](#)
- [“sethostent_r Subroutine” on page 196](#)

**gethostbyname Subroutine**

**Purpose**
Gets network host entry by name.

**Library**

- Standard C Library (*libc.a*)
  - (*libbind*)
  - (*libnisd*)
  - (*liblocal*)


Syntax

```c
#include <netdb.h>

struct hostent *gethostbyname (char *Name);
```

Description

The `gethostbyname` subroutine is threadsafe in AIX 4.3 and later. However, the return value points to static data that is overwritten by subsequent calls. This data must be copied to be saved for use by subsequent calls.

The `gethostbyname` subroutine retrieves host address and name information using a host name as a search key. Unless specified, the `gethostbyname` subroutine uses the default name services ordering, that is, it queries DNS/BIND, NIS or the local `/etc/hosts` file for the name.

When using DNS/BIND name service resolution, if the `/etc/resolv.conf` file exists, the `gethostbyname` subroutine queries the domain name server. The `gethostbyname` subroutine recognizes domain name servers as described in RFC883.

When using NIS for name resolution, if the `getdomainname` subroutine is successful and `yp_bind` indicates yellow pages are running, then the `gethostbyname` subroutine queries NIS for the name.

The `gethostbyname` subroutine also searches the local `/etc/hosts` file for the name when indicated to do so.

The `gethostbyname` subroutine returns a pointer to a `hostent` structure, which contains information obtained from a name resolution services. The `hostent` structure is defined in the `netdb.h` header file.

Parameters

- `Name` Points to the host name.

Return Values

The `gethostbyname` subroutine returns a pointer to a `hostent` structure on success.

If the parameter `Name` passed to `gethostbyname` is actually an IP address, `gethostbyname` will return a non-NULL `hostent` structure with an IP address as the hostname without actually doing a lookup. Remember to call `[inet_addr]` subroutine to make sure `Name` is not an IP address before calling `gethostbyname`. To resolve an IP address call `gethostbyaddr` instead.

If an error occurs or if the end of the file is reached, the `gethostbyname` subroutine returns a null pointer and sets `h_errno` to indicate the error.

The environment variable, `NSORDER` can be set to override the default name services ordering and the order specified in the `/etc/netsvc.conf` file.

By default, resolver routines first attempt to resolve names through the DNS/BIND, then NIS and the `/etc/hosts` file. The `/etc/netsvc.conf` file may specify a different search order. The environment variable `NSORDER` overrides both the `/etc/netsvc.conf` file and the default ordering. Services are ordered as `hosts = value, value, value` in the `/etc/netsvc.conf` file where at least one value must be specified from the list `bind, nis, local`. `NSORDER` specifies a list of values.
Error Codes

The `gethostbyname` subroutine is unsuccessful if any of the following errors occur:

<table>
<thead>
<tr>
<th>Error</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>HOST_NOT_FOUND</td>
<td>The host specified by the <code>Name</code> parameter was not found.</td>
</tr>
<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>
<tr>
<td>NO_ADDRESS</td>
<td>The requested <code>Name</code> is valid but does not have an Internet address at the name server.</td>
</tr>
<tr>
<td>SERVICE_UNAVAILABLE</td>
<td>None of the name services specified are running or available.</td>
</tr>
</tbody>
</table>

Examples

The following program fragment illustrates the use of the `gethostbyname` subroutine to look up a destination host:

```c
hp=gethostbyname(argv[1]);
if(hp == NULL) {
    fprintf(stderr, "rlogin: %s: unknown host\n", argv[1]);
    exit(2);
}
```

Files

<table>
<thead>
<tr>
<th>File</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>/etc/hosts</td>
<td>Contains the host name data base.</td>
</tr>
<tr>
<td>/etc/resolv.conf</td>
<td>Contains the name server and domain name.</td>
</tr>
<tr>
<td>/etc/netsvc.conf</td>
<td>Contains the name services ordering.</td>
</tr>
<tr>
<td>/usr/include/netdb.h</td>
<td>Contains the network database structure.</td>
</tr>
</tbody>
</table>

Related Information

The `endhostent` subroutine, `gethostbyaddr` subroutine, `gethostent` subroutine, `sethostent` subroutine, `inet_addr` subroutine.

Sockets Overview and Network Address Translation in *AIX 5L Version 5.3 Communications Programming Concepts*.

`gethostbyname_r` Subroutine

Purpose

Gets network host entry by name.

Library

Standard C Library (`libc.a`)  
(`libbind`)  
(`libnis`)  
(`liblocal`)

Syntax

```c
#include <netdb.h>
int gethostbyname_r(const char *Name, struct hostent *Htent, struct hostent_data *Ht_data);
```
Description
This function internally calls the `gethostbyname` subroutine and stores the value returned by the `gethostbyname` subroutine to the hostent structure.

Parameters

- **Name**
  - Points to the host name (which is a constant).
- **Htent**
  - Points to a hostent structure in which the return value of the `gethostbyname` subroutine is stored.
- **Ht_data**
  - Points to a hostent_data structure.

Return Values
The function returns a 0 if successful and a -1 if unsuccessful.

Note:
The return value of the `gethostbyname` subroutine points to static data that is overwritten by subsequent calls. This data must be copied at every call to be saved for use by subsequent calls. The `gethostbyname_r` subroutine solves this problem.

If the Name parameter is an IP address, this subroutine searches for a machine with that IP address as a name. Because of this, use the `gethostbyaddr_r` subroutine instead of the `gethostbyname_r` subroutine if the Name parameter is an IP address.

Error Codes
The `gethostbyname_r` subroutine is unsuccessful if any of the following errors occurs:

- **HOST_NOT_FOUND**
  - The host specified by the Name parameter was not found.
- **TRY_AGAIN**
  - The local server did not receive a response from an authoritative server. Try again later.
- **NO_RECOVERY**
  - An unrecoverable error occurred.
- **NO_ADDRESS**
  - The requested Name is valid but does not have an Internet address at the name server.
- **SERVICE_UNAVAILABLE**
  - None of the name services specified are running or available.
- **EINVAL**
  - The hostent pointer is NULL.

Files

- **/etc/hosts**
  - Contains the host name data base.
- **/etc/resolv.conf**
  - Contains the name server and domain name.
- **/etc/netsvc.conf**
  - Contains the name services ordering.
- **/usr/include/netdb.h**
  - Contains the network database structure.

Related Information
- “endhostent_r Subroutine” on page 41, “gethostbyaddr_r Subroutine” on page 67, “gethostent_r Subroutine” on page 72, and “sethostent_r Subroutine” on page 196.
gethostent Subroutine

Purpose
Retrieves a network host entry.

Library
Standard C Library (libc.a)
(libbind)
(libnis)
(liblocal)

Syntax
#include <netdb.h>

struct hostent *gethostent ()

Description
The gethostent subroutine is threadsafe in AIX 4.3 and later. However, the return value points to static data that is overwritten by subsequent calls. This data must be copied to be saved for use by subsequent calls.

When using DNS/BIND name service resolution, the gethostent subroutine is not defined.

When using NIS name service resolution or searching the local /etc/hosts file, the gethostent subroutine reads the next line of the /etc/hosts file, opening the file if necessary.

The gethostent subroutine returns a pointer to a hostent structure, which contains the equivalent fields for a host description line in the /etc/hosts file. The hostent structure is defined in the netdb.h file.

Return Values
Upon successful completion, the gethostent subroutine returns a pointer to a hostent structure.

If an error occurs or the end of the file is reached, the gethostent subroutine returns a null pointer.

Files
/etc/hosts Contains the host name database.
/etc/netsvc.conf Contains the name services ordering.
/usr/include/netdb.h Contains the network database structure.

Related Information
The gethostbyaddr subroutine, gethostbyname subroutine, sethostent subroutine, endhostent subroutine.

Sockets Overview and Network Address Translation in AIX 5L Version 5.3 Communications Programming Concepts.

gethostent_r Subroutine

Purpose
Retrieves a network host entry.
Library
Standard C Library (libc.a)
(libbind)
(libnis)
(liblocal)

Syntax
#include <netdb.h>

int gethostent_r (htent, ht_data)  
struct hostent *htent;  
struct hostent_data *ht_data;

Description
When using DNS/BIND name service resolution, the gethostent_r subroutine is not defined.

When using NIS name service resolution or searching the local /etc/hosts file, the gethostent_r subroutine reads the next line of the /etc/hosts file, and opens the file if necessary.

The gethostent_r subroutine internally calls the gethostent subroutine, and stores the values in the htent and ht_data structures.

The gethostent subroutine overwrites the static data returned in subsequent calls. The gethostent_r subroutine does not.

Parameters
htent Points to the hostent structure
ht_data Points to the hostent_data structure

Return Values
This subroutine returns a 0 if successful, and a -1 if unsuccessful.

Files
/etc/hosts Contains the host name database.
/etc/netsvc.conf Contains the name services ordering.
/usr/include/netdb.h Contains the network database structure.

Related Information
“gethostbyaddr_r Subroutine” on page 67, “gethostbyname_r Subroutine” on page 70, “sethostent_r Subroutine” on page 196, and “endhostent_r Subroutine” on page 41.

gethostid Subroutine

Purpose
Gets the unique identifier of the current host.

Library
Standard C Library (libc.a)
#include <unistd.h>

int gethostid ( )

**Description**
The `gethostid` subroutine allows a process to retrieve the 32-bit identifier for the current host. In most cases, the host ID is stored in network standard byte order and is a DARPA Internet address for a local machine.

All applications containing the `gethostid` subroutine must be compiled with the `_BSD` macro set to a specific value. Acceptable values are 43 and 44. In addition, all socket applications must include the BSD `libbsd.a` library.

**Return Values**
Upon successful completion, the `gethostid` subroutine returns the identifier for the current host.

If the `gethostid` subroutine is unsuccessful, the system handler performs the following functions:
- Returns a value of -1 to the calling program.
- Moves an error code, indicating the specific error, into the `errno` global variable. For further explanation of the `errno` variable see “Error Notification Object Class” in *AIX 5L Version 5.3 Communications Programming Concepts.*

**Related Information**
The `gethostname` subroutine, `sethostid` subroutine, `sethostname` subroutine.

[Sockets Overview](#) in *AIX 5L Version 5.3 Communications Programming Concepts.*

## gethostname Subroutine

**Purpose**
Gets the name of the local host.

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

**Syntax**
```c
#include <unistd.h>
int gethostname ( Name, NameLength )
char *Name;
size_t NameLength;
```

**Description**
The `gethostname` subroutine retrieves the standard host name of the local host. If excess space is provided, the returned `Name` parameter is null-terminated. If insufficient space is provided, the returned name is truncated to fit in the given space. System host names are limited to 256 characters.

The `gethostname` subroutine allows a calling process to determine the internal host name for a machine on a network.
All applications containing the `gethostname` subroutine must be compiled with the `_BSD` macro set to a specific value. Acceptable values are 43 and 44. In addition, all socket applications must include the BSD `libbsd.a` library.

**Parameters**

- **Name**: Specifies the address of an array of bytes where the host name is to be stored.
- **NameLength**: Specifies the length of the `Name` array.

**Return Values**

Upon successful completion, the system returns a value of 0.

If the `gethostname` subroutine is unsuccessful, the subroutine handler performs the following functions:

- Returns a value of -1 to the calling program.
- Moves an error code, indicating the specific error, into the `errno` global variable.

**Error Codes**

The `gethostname` subroutine is unsuccessful if the following is true:

<table>
<thead>
<tr>
<th>Error</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EFAULT</td>
<td>The <code>Name</code> parameter or <code>NameLength</code> parameter gives an invalid address.</td>
</tr>
</tbody>
</table>

**Related Information**

The `gethostid`, `sethostid`, `sethostname` subroutine.

Sockets Overview in *AIX 5L Version 5.3 Communications Programming Concepts*.

### getnameinfo Subroutine

**Purpose**

Address-to-host name translation [given the binary address and port].

**Note**: This is the reverse functionality of the `getaddrinfo Subroutine` on page 61 host-to-address translation.

**Attention**: This is not a POSIX (1003.1g) specified function.

**Library**

Library (*libc.a*)

**Syntax**

```c
#include <sys/socket.h>
#include <netdb.h>
int getnameinfo (sa, salen, host, hostlen, serv, servlen, flags)
const struct sockaddr *sa;
char *host;
size_t hostlen;
char *serv;
size_t servlen;
int flags;
```
Description

The sa parameter points to either a sockaddr_in structure (for IPv4) or a sockaddr_in6 structure (for IPv6) that holds the IP address and port number. The salen parameter gives the length of the sockaddr_in or sockaddr_in6 structure.

Note: A reverse lookup is performed on the IP address and port number provided in sa.

The host parameter is a buffer where the hostname associated with the IP address is copied. The hostlen parameter provides the length of this buffer. The service name associated with the port number is copied into the buffer pointed to by the serv parameter. The servlen parameter provides the length of this buffer.

The flags parameter defines flags that may be used to modify the default actions of this function. By default, the fully-qualified domain name (FQDN) for the host is looked up in DNS and returned.

NI_NOFQDN If set, return only the hostname portion of the FQDN. If cleared, return the FQDN.

NI_NUMERICHOST If set, return the numeric form of the host address. If cleared, return the name.

NI_NAMEREQD If set, return an error if the host’s name cannot be determined. If cleared, return the numeric form of the host’s address (as if NI_NUMERICHOST had been set).

NI_NUMERICSERV If set, return the numeric form of the desired service. If cleared, return the service name.

NI_DGRAM If set, consider the desired service to be a datagram service, (for example, call getservbyport with an argument of udp). If clear, consider the desired service to be a stream service (for example, call getservbyport with an argument of tcp).

Return Values

A zero return value indicates successful completion; a non-zero value indicates failure. If successful, the strings for hostname and service name are copied into the host and serv buffers, respectively. If either the host or service name cannot be located, the numeric form is copied into the host and serv buffers, respectively.

Related Information

“getaddrinfo Subroutine” on page 61, and “freeaddrinfo Subroutine” on page 61.


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

getnetbyaddr Subroutine

Purpose

Gets network entry by address.

Library

Standard C Library (libc.a)
Syntax

```c
#include <netdb.h>

struct netent *getnetbyaddr (Network, Type)
int Network;
int Type;
```

Description

The `getnetbyaddr` subroutine is threadsafe in AIX 4.3 and later. However, the return value points to static data that is overwritten by subsequent calls. This data must be copied to be saved for use by subsequent calls.

The `getnetbyaddr` subroutine retrieves information from the `/etc/networks` file using the network address as a search key. The `getnetbyaddr` subroutine searches the file sequentially from the start of the file until it encounters a matching net number and type or until it reaches the end of the file.

The `getnetbyaddr` subroutine returns a pointer to a `netent` structure, which contains the equivalent fields for a network description line in the `/etc/networks` file. The `netent` structure is defined in the `netdb.h` file.

Use the `endnetent` subroutine to close the `/etc/networks` file.

All applications containing the `getnetbyaddr` subroutine must be compiled with the `_BSD` macro set to a specific value. Acceptable values are 43 and 44. In addition, all socket applications must include the BSD `libbsd.a` library.

Parameters

- `Network` Specifies the number of the network to be located.
- `Type` Specifies the address family for the network. The only supported value is `AF_INET`.

Return Values

Upon successful completion, the `getnetbyaddr` subroutine returns a pointer to a `netent` structure.

If an error occurs or the end of the file is reached, the `getnetbyaddr` subroutine returns a null pointer.

Files

- `/etc/networks` Contains official network names.

Related Information

The `endnetent` subroutine, `getnetbyname` subroutine, `getnetent` subroutine, `setnetent` subroutine. The `Sockets Overview` in AIX 5L Version 5.3 General Programming Concepts: Writing and Debugging Programs.

`getnetbyaddr_r` Subroutine

Purpose

Gets network entry by address.
Library
Standard C Library (libc.a)

Syntax
#include<netdb.h>
int getnetbyaddr_r(net, type, netent, net_data)

register in_addr_t net;
register int type;
struct netent *netent;
struct netent_data *net_data;

Description
The getnetbyaddr_r subroutine retrieves information from the /etc/networks file using the Name parameter as a search key.

The getnetbyaddr_r subroutine internally calls the getnetbyaddr subroutine and stores the information in the structure data.

The getnetbyaddr subroutine overwrites the static data returned in subsequent calls. The getnetbyaddr_r subroutine does not.

Use the endnetent_r subroutine to close the /etc/networks file.

Parameters
Net
Specifies the number of the network to be located.

Type
Specifies the address family for the network. The only supported values are AF_INET, and AF_INET6.

netent
Points to the netent structure.

net_data
Points to the net_data structure.

Return Values
The function returns a 0 if successful and a -1 if unsuccessful.

Files
/etc/networks Contains official network names.

Related Information
”endnetent_r Subroutine” on page 43, ”getnetbyname_r Subroutine” on page 79, ”getnetent_r Subroutine” on page 81, and ”setnetent_r Subroutine” on page 200.

getnetbyname Subroutine

Purpose
Gets network entry by name.

Library
Standard C Library (libc.a)
Syntax
#include <netdb.h>

struct netent *getnetbyname (Name);
char *Name;

Description
The getnetbyname subroutine is threadsafe in AIX 4.3 and later. However, the return value points to static data that is overwritten by subsequent calls. This data must be copied to be saved for use by subsequent calls.

The getnetbyname subroutine retrieves information from the /etc/networks file using the Name parameter as a search key. The getnetbyname subroutine searches the /etc/networks file sequentially from the start of the file until it encounters a matching net name or until it reaches the end of the file.

The getnetbyname subroutine returns a pointer to a netent structure, which contains the equivalent fields for a network description line in the /etc/networks file. The netent structure is defined in the netdb.h file.

Use the endnetent subroutine to close the /etc/networks file.

All applications containing the getnetbyname subroutine must be compiled with the _BSD macro set to a specific value. Acceptable values are 43 and 44. In addition, all socket applications must include the BSD libbsd.a library.

Parameters
Name Points to a string containing the name of the network.

Return Values
Upon successful completion, the getnetbyname subroutine returns a pointer to a netent structure. If an error occurs or the end of the file is reached, the getnetbyname subroutine returns a null pointer.

Files
/etc/networks Contains official network names.

Related Information
The endnetent subroutine, getnetbyaddr subroutine, getnetent subroutine, setnetent subroutine.

Sockets Overview in AIX 5L Version 5.3 Communications Programming Concepts.

getnetbyname_r Subroutine

Purpose
Gets network entry by name.

Library
Standard C Library (libc.a)
# Syntax

```c
#include <netdb.h>
int getnetbyname_r(Name, netent, net_data)
register const char *Name;
struct netent *netent;
struct netent_data *net_data;
```

### Description

The `getnetbyname_r` subroutine retrieves information from the `/etc/networks` file using the `Name` parameter as a search key.

The `getnetbyname_r` subroutine internally calls the `getnetbyname` subroutine and stores the information in the structure data.

The `getnetbyname` subroutine overwrites the static data returned in subsequent calls. The `getnetbyname_r` subroutine does not.

Use the `endnetent_r` subroutine to close the `/etc/networks` file.

### Parameters

- **Name**: Points to a string containing the name of the network.
- **netent**: Points to the `netent` structure.
- **net_data**: Points to the `net_data` structure.

### Return Values

The function returns a 0 if successful and a -1 if unsuccessful.

**Note:** If an error occurs or the end of the file is reached, the `getnetbyname_r` subroutine returns a -1 to indicate error.

### Files

- **/etc/networks**: Contains official network names.

### Related Information

- “endnetent_r Subroutine” on page 43, “getnetbyaddr_r Subroutine” on page 77, “getnetent_r Subroutine” on page 81, and “setnetent_r Subroutine” on page 200.

---

## getnetent Subroutine

### Purpose

Gets network entry.

### Library

Standard C Library (`libc.a`)

### Syntax

```c
#include <netdb.h>
struct netent *getnetent ( )
```
Description
The getnetent subroutine is threadsafe in AIX 4.3 and later. However, the return value points to static data that is overwritten by subsequent calls. This data must be copied to be saved for use by subsequent calls.

The getnetent subroutine retrieves network information by opening and sequentially reading the /etc/networks file.

The getnetent subroutine returns a pointer to a netent structure, which contains the equivalent fields for a network description line in the /etc/networks file. The netent structure is defined in the netdb.h file.

Use the endnetent subroutine to close the /etc/networks file.

All applications containing the getnetent subroutine must be compiled with the _BSD macro set to a specific value. Acceptable values are 43 and 44. In addition, all socket applications must include the BSD libbsd.a library.

Return Values
Upon successful completion, the getnetent subroutine returns a pointer to a netent structure.

If an error occurs or the end of the file is reached, the getnetent subroutine returns a null pointer.

Files
/etc/networks Contains official network names.

Related Information
The endnetent subroutine, getnetbyaddr subroutine, getnetbyname subroutine, setnetent subroutine.

Sockets Overview in AIX 5L Version 5.3 Communications Programming Concepts.

gnetent_r Subroutine

Purpose
Gets network entry.

Library
Standard C Library (libc.a)

Syntax
#include <netdb.h>
int getnetent_r(netent, net_data)

struct netent *netent;
struct netent_data *net_data;

Description
The getnetent_r subroutine retrieves network information by opening and sequentially reading the /etc/networks file. This subroutine internally calls the getnetent subroutine and stores the values in the hostent structure.

The getnetent subroutine overwrites the static data returned in subsequent calls. The getnetent_r subroutine does not. Use the endnetent_r subroutine to close the /etc/networks file.
Parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>netent</td>
<td>Points to the netent structure.</td>
</tr>
<tr>
<td>net_data</td>
<td>Points to the net_data structure.</td>
</tr>
</tbody>
</table>

Return Values
The function returns a 0 if successful and a -1 if unsuccessful.

Note: If an error occurs or the end of the file is reached, the getnetent_r subroutine returns a -1 to indicate error.

Files

<table>
<thead>
<tr>
<th>File</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>/etc/networks</td>
<td>Contains official network names.</td>
</tr>
</tbody>
</table>

Related Information

“endnetent_r Subroutine” on page 43, “getnetbyaddr_r Subroutine” on page 77, “getnetbyname_r Subroutine” on page 79, and “setnetent_r Subroutine” on page 200.

getnetgrent_r Subroutine

Purpose
Handles the group network entries.

Library
Standard C Library (libc.a)

Syntax
```c
#include<netdb.h>
int getnetgrent_r(machinep, namep, domainp, ptr)
    char **machinep, **namep, **domainp;
    void **ptr;
```

Description
The getnetgrent_r subroutine internally calls the getnetgrent subroutine and stores the information in the structure data. This subroutine returns 1 or 0, depending if netgroup contains the machine, user, and domain triple as a member. Any of these three strings can be NULL, in which case it signifies a wild card.

The getnetgrent_r subroutine returns the next member of a network group. After the call, the machinep parameter contains a pointer to a string containing the name of the machine part of the network group member. The namep and domainp parameters contain similar pointers. If machinep, namep, or domainp is returned as a NULL pointer, it signifies a wild card.

The getnetgrent subroutine overwrites the static data returned in subsequent calls. The getnetgrent_r subroutine does not.

Parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>machinep</td>
<td>Points to the string containing the machine part of the network group.</td>
</tr>
</tbody>
</table>
namep
Points to the string containing the user part of the network group.

domainp
Points to the string containing the domain name.

ptr
Keeps the function threadsafe.

**Return Values**
The function returns a 0 if successful and a -1 if unsuccessful.

**Files**

/etc/netgroup
Contains network groups recognized by the system.

/usr/include/netdb.h
Contains the network database structures.

**Related Information**
“endnetgrent_r Subroutine” on page 44, and “setnetgrent_r Subroutine” on page 200.

---

**getpeername Subroutine**

**Purpose**
Gets the name of the peer socket.

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

**Syntax**
```
#include <sys/socket.h>
int getpeername (Socket, Name, NameLength);
```

**Description**
The getpeername subroutine retrieves the Name parameter from the peer socket connected to the specified socket. The Name parameter contains the address of the peer socket upon successful completion.

A process created by another process can inherit open sockets. The created process may need to identify the addresses of the sockets it has inherited. The getpeername subroutine allows a process to retrieve the address of the peer socket at the remote end of the socket connection.

**Note:** The getpeername subroutine operates only on connected sockets.

A process can use the getsockname subroutine to retrieve the local address of a socket.

All applications containing the getpeername subroutine must be compiled with the _BSD macro set to a specific value. Acceptable values are 43 and 44. In addition, all socket applications must include the BSD libbsd.a library.
Parameters

- **Socket**: Specifies the descriptor number of a connected socket.
- **Name**: Points to a `sockaddr` structure that contains the address of the destination socket upon successful completion. The `/usr/include/sys/socket.h` file defines the `sockaddr` structure.
- **NameLength**: Points to the size of the address structure. Initializes the `NameLength` parameter to indicate the amount of space pointed to by the `Name` parameter. Upon successful completion, it returns the actual size of the `Name` parameter returned.

Return Values

Upon successful completion, a value of 0 is returned and the `Name` parameter holds the address of the peer socket.

If the `getpeername` subroutine is unsuccessful, the system handler performs the following functions:

- Returns a value of -1 to the calling program.
- Moves an error code, indicating the specific error, into the `errno` global variable.

Error Codes

The `getpeername` subroutine is unsuccessful if any of the following errors occurs:

<table>
<thead>
<tr>
<th>Error</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EBADF</td>
<td>The <code>Socket</code> parameter is not valid.</td>
</tr>
<tr>
<td>EINVAL</td>
<td>The socket has been shut down.</td>
</tr>
<tr>
<td>ENOTSOCK</td>
<td>The <code>Socket</code> parameter refers to a file, not a socket.</td>
</tr>
<tr>
<td>ENOTCONN</td>
<td>The socket is not connected.</td>
</tr>
<tr>
<td>ENOBUFS</td>
<td>Insufficient resources were available in the system to complete the call.</td>
</tr>
<tr>
<td>EFAULT</td>
<td>The <code>Address</code> parameter is not in a writable part of the user address space.</td>
</tr>
</tbody>
</table>

Examples

The following program fragment illustrates the use of the `getpeername` subroutine to return the address of the peer connected on the other end of the socket:

```c
struct sockaddr_in name;
int namelen = sizeof(name);
...
if(getpeername(0,(struct sockaddr*)&name, &namelen)<0){
    syslog(LOG_ERR,"getpeername: %m");
    exit(1);
} else 
    syslog(LOG_INFO,"Connection from %s",inet_ntoa(name.sin_addr));
...
```

Related Information

The `accept`, `bind`, `getsockname`, `socket` subroutine.

[Sockets Overview](#) in *AIX 5L Version 5.3 Communications Programming Concepts.*
getprotobynamesubroutine

Purpose
Gets protocol entry from the /etc/protocols file by protocol name.

Library
Standard C Library (libc.a)

Syntax
#include <netdb.h>

struct protoent *getprotobyname (Name);
char *Name;

Description
The getprotobynamesubroutine is threadsafe in AIX 4.3 and later. However, the return value points to static data that is overwritten by subsequent calls. This data must be copied to be saved for use by subsequent calls.

The getprotobynamesubroutine retrieves protocol information from the /etc/protocols file by protocol name. An application program can use the getprotobynamesubroutine to access a protocol name, its aliases, and protocol number.

The getprotobynamesubroutine searches the protocols file sequentially from the start of the file until it finds a matching protocol name or until it reaches the end of the file. The subroutine returns a pointer to a protoent structure, which contains fields for a line of information in the /etc/protocols file. The netdb.h file defines the protoent structure.

Use the endprotoent subroutine to close the /etc/protocols file.

All applications containing the getprotobynamesubroutine must be compiled with the _BSD macro set to a specific value. Acceptable values are 43 and 44. In addition, all socket applications must include the BSD libbsd.a library.

Parameters
Name Specifies the protocol name.

Return Values
Upon successful completion, the getprotobynamesubroutine returns a pointer to a protoent structure.

If an error occurs or the end of the file is reached, the getprotobynamesubroutine returns a null pointer.

Related Information
The endprotoent subroutine, getprotobynumber subroutine, getprotoent subroutine, setprotoent subroutine.

Sockets Overview in AIX 5L Version 5.3 Communications Programming Concepts.
getprotobyname_r Subroutine

Purpose
Gets protocol entry from the /etc/protocols file by protocol name.

Library
Standard C Library (libc.a)

Syntax
#include <netdb.h>

int getprotobyname_r(Name, protoent, proto_data)
register const char *Name;
struct protoent *protoent;
struct protoent_data *proto_data;

Description
The getprotobyname_r subroutine retrieves protocol information from the /etc/protocols file by protocol name.

An application program can use the getprotobyname_r subroutine to access a protocol name, aliases, and protocol number.

The getprotobyname_r subroutine searches the protocols file sequentially from the start of the file until it finds a matching protocol name or until it reaches the end of the file. The subroutine writes the protoent structure, which contains fields for a line of information in the /etc/protocols file.

The netdb.h file defines the protoent structure.

The getprotobyname subroutine overwrites any static data returned in subsequent calls. The getprotobyname_r subroutine does not.

Use the endprotoent_r subroutine to close the /etc/protocols file.

Parameters
Name
Specifies the protocol name.
protoent
Points to the protoent structure.
proto_data
Points to the proto_data structure.

Return Values
The function returns a 0 if successful and a -1 if unsuccessful.

Note: If an error occurs or the end of the file is reached, the getprotobyname_r subroutine returns a -1 to indicate error.

Related Information
endprotoent_r Subroutine on page 45, getprotobynumber_r Subroutine on page 88, getprotoent_r Subroutine on page 90, and setprotoent_r Subroutine on page 202.
**getprotobynumber Subroutine**

**Purpose**
Gets a protocol entry from the `/etc/protocols` file by number.

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

**Syntax**
```
#include <netdb.h>

struct protoent *getprotobynumber (int Protocol);
```

**Description**
The `getprotobynumber` subroutine is threadsafe in AIX 4.3 and later. However, the return value points to static data that is overwritten by subsequent calls. This data must be copied to be saved for use by subsequent calls.

The `getprotobynumber` subroutine retrieves protocol information from the `/etc/protocols` file using a specified protocol number as a search key. An application program can use the `getprotobynumber` subroutine to access a protocol name, its aliases, and protocol number.

The `getprotobynumber` subroutine searches the `/etc/protocols` file sequentially from the start of the file until it finds a matching protocol name or protocol number, or until it reaches the end of the file. The subroutine returns a pointer to a `protoent` structure, which contains fields for a line of information in the `/etc/protocols` file. The `netdb.h` file defines the `protoent` structure.

Use the `endprotoent` subroutine to close the `/etc/protocols` file.

All applications containing the `getprotobynumber` subroutine must be compiled with the `_BSD` macro set to a specific value. Acceptable values are 43 and 44. In addition, all socket applications must include the BSD `libbsd.a` library.

**Parameters**

`Protocol` Specifies the protocol number.

**Return Values**
Upon successful completion, the `getprotobynumber` subroutine, returns a pointer to a `protoent` structure.

If an error occurs or the end of the file is reached, the `getprotobynumber` subroutine returns a null pointer.

**Files**

`/etc/protocols` Contains protocol information.
getprotobynumber_r Subroutine

Purpose
Gets a protocol entry from the /etc/protocols file by number.

Library
Standard C Library (libc.a)

Syntax

```
#include <netdb.h>
int getprotobynumber_r(proto, protoent, proto_data)
register int proto;
struct protoent *protoent;
struct protoent_data *proto_data;
```

Description
The getprotobynumber_r subroutine retrieves protocol information from the /etc/protocols file using a specified protocol number as a search key.

An application program can use the getprotobynumber_r subroutine to access a protocol name, aliases, and number.

The getprotobynumber_r subroutine searches the /etc/protocols file sequentially from the start of the file until it finds a matching protocol name, protocol number, or until it reaches the end of the file.

The subroutine writes the protoent structure, which contains fields for a line of information in the /etc/protocols file.

The netdb.h file defines the protoent structure.

The getprotobynumber subroutine overwrites static data returned in subsequent calls. The getprotobynumber_r subroutine does not.

Use the endprotoent_r subroutine to close the /etc/protocols file.

Parameters

- `proto` Specifies the protocol number.
- `protoent` Points to the `protoent` structure.
- `proto_data` Points to the `proto_data` structure.

Return Values
The function returns a 0 if successful and a -1 if unsuccessful.

Note: If an error occurs or the end of the file is reached, the getprotobynumber_r subroutine sets the `protoent` parameter to NULL and returns a -1 to indicate error.
getprotoent Subroutine

Purpose
Gets protocol entry from the /etc/protocols file.

Library
Standard C Library (libc.a)

Syntax
```
#include <netdb.h>
struct protoent *getprotoent ( )
```

Description
The getprotoent subroutine is threadsafe in AIX 4.3 and later. However, the return value points to static data that is overwritten by subsequent calls. This data must be copied to be saved for use by subsequent calls.

The getprotoent subroutine retrieves protocol information from the /etc/protocols file. An application program can use the getprotoent subroutine to access a protocol name, its aliases, and protocol number.

The getprotoent subroutine opens and performs a sequential read of the /etc/protocols file. The getprotoent subroutine returns a pointer to a protoent structure, which contains the fields for a line of information in the /etc/protocols file. The netdb.h file defines the protoent structure.

Use the endprotoent subroutine to close the /etc/protocols file.

All applications containing the getprotoent subroutine must be compiled with the _BSD macro set to a specific value. Acceptable values are 43 and 44. In addition, all socket applications must include the BSD libbsd.a library.

Return Values
Upon successful completion, the getprotoent subroutine returns a pointer to a protoent structure.

If an error occurs or the end of the file is reached, the getprotoent subroutine returns a null pointer.

Files
```
//etc/protocols
```

Contains protocol information.

Related Information
The endprotoent subroutine, getprotobyname subroutine, getprotobynumber subroutine, setprotoent subroutine.
getprotoent_r Subroutine

Purpose
Gets protocol entry from the /etc/protocols file.

Library
Standard C Library (libc.a)

Syntax
#include <netdb.h>

int getprotoent_r(protoent, proto_data)
struct protoent *protoent;
struct protoent_data *proto_data;

Description
The getprotoent_r subroutine retrieves protocol information from the /etc/protocols file. An application program can use the getprotoent_r subroutine to access a protocol name, its aliases, and protocol number. The getprotoent_r subroutine opens and performs a sequential read of the /etc/protocols file. This subroutine writes to the protoent structure, which contains the fields for a line of information in the /etc/protocols file.

The netdb.h file defines the protoent structure.

Use the endprotoent_r subroutine to close the /etc/protocols file. Static data is overwritten in subsequent calls when using the getprotoent subroutine. The getprotoent_r subroutine does not overwrite.

Parameters
protoent Points to the protoent structure.
proto_data Points to the proto_data structure.

Return Values
The function returns a 0 if successful and a -1 if unsuccessful.

Note: If an error occurs or the end of the file is reached, the getprotoent_r subroutine sets the protoent parameter to NULL.

Files
/etc/protocols Contains protocol information.

Related Information
"endprotoent_r Subroutine" on page 45, “setprotoent_r Subroutine” on page 202, "getprotobytename_r Subroutine" on page 86, “getprotobynumber_r Subroutine” on page 88, and “setprotoent_r Subroutine” on page 202.
GetQueuedCompletionStatus Subroutine

Purpose
Dequeues a completion packet from a specified I/O completion port.

Syntax
```c
#include <iocp.h>
boolean_t GetQueuedCompletionStatus(HANDLE CompletionPort, LPDWORD TransferCount, CompletionKey, LPOVERLAPPED Overlapped, DWORD Timeout);
```

Description
The GetQueuedCompletionStatus subroutine attempts to dequeue a completion packet from the CompletionPort parameter. If there is no completion packet to be dequeued, this subroutine waits a predetermined amount of time as indicated by the Timeout parameter for a completion packet to arrive.

The GetQueuedCompletionStatus subroutine returns a boolean indicating whether or not a completion packet has been dequeued.

The GetQueuedCompletionStatus subroutine is part of the I/O Completion Port (IOCP) kernel extension.

Note: This subroutine only works with file descriptors of sockets, or regular files for use with the Asynchronous I/O (AIO) subsystem. It does not work with file descriptors of other types.

Parameters
- **CompletionPort**: Specifies the completion port that this subroutine will attempt to access.
- **TransferCount**: Specifies the number of bytes transferred. This parameter is set by the subroutine from the value received in the completion packet.
- **CompletionKey**: Specifies the completion key associated with the file descriptor used in the transfer request. This parameter is set by the subroutine from the value received in the completion packet.
- **Overlapped**: Specifies the overlapped structure used in the transfer request. This parameter is set by the subroutine from the value received in the completion packet. For regular files, this parameter contains a pointer to the AIOCB for a completed AIO request. If an application uses the same completion port for both socket and AIO to regular files, it must use unique CompletionKey values to differentiate between sockets and regular files in order to properly interpret the Overlapped parameter.
- **Timeout**: Specifies the amount of time in milliseconds the subroutine is to wait for a completion packet. If this parameter is set to INFINITE, the subroutine will never timeout.

Return Values
Upon successful completion, the GetQueuedCompletionStatus subroutine returns a boolean indicating its success.

If the GetQueuedCompletionStatus subroutine is unsuccessful, the subroutine handler performs the following functions:
- Returns a value of 0 to the calling program.
- Moves an error code, indicating the specific error, into the errno global variable. For further explanation of the errno variable, see the link in the Related Information section of this document.
Error Codes
The subroutine is unsuccessful if any of the following errors occur:

**ETIMEDOUT**
No completion packet arrived to be dequeued and the *Timeout* parameter has elapsed.

**EINVAL**
The *CompletionPort* parameter was invalid.

**EAGAIN**
Resource temporarily unavailable. If a sleep is interrupted by a signal, *EAGAIN* may be returned.

**ENOTCONN**
Socket is not connected. The *ENOTCONN* return can happen for two reasons. One is if a request is made, the *fd* is then closed, then the request is returned back to the process. The error will be *ENOTCONN*. The other is if the socket drops while the *fd* is still open, the requests after the socket drops (disconnects) will return *ENOTCONN*.

Examples
The following program fragment illustrates the use of the *GetQueuedCompletionStatus* subroutine to dequeue a completion packet.

```c
int transfer_count, completion_key;
LPOVERLAPPED overlapped;
c = GetQueuedCompletionStatus(34, &transfer_count, &completion_key, &overlapped, 1000);
```

Related Information
The "socket Subroutine" on page 226, "accept Subroutine" on page 29, "ReadFile Subroutine" on page 155, "WriteFile Subroutine" on page 243, "CreateIoCompletionPort Subroutine" on page 36, and "PostQueuedCompletionStatus Subroutine" on page 150.

For further explanation of the *errno* variable, see [Error Notification Object Class] in *AIX 5L Version 5.3 General Programming Concepts: Writing and Debugging Programs*.

### getservbyname Subroutine

**Purpose**
Gets service entry by name.

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

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

struct servent *getservbyname (char *Name, *Protocol);
```

**Description**
The *getservbyname* subroutine is threadsafe in AIX 4.3 and later. However, the return value points to static data that is overwritten by subsequent calls. This data must be copied to be saved for use by subsequent calls.

The *getservbyname* subroutine retrieves an entry from the */etc/services* file using the service name as a search key.
An application program can use the `getservbyname` subroutine to access a service, service aliases, the protocol for the service, and a protocol port number for the service.

The `getservbyname` subroutine searches the `/etc/services` file sequentially from the start of the file until it finds one of the following:
- Matching name and protocol number
- Matching name when the `Protocol` parameter is set to 0
- End of the file

Upon locating a matching name and protocol, the `getservbyname` subroutine returns a pointer to the `servent` structure, which contains fields for a line of information from the `/etc/services` file. The `netdb.h` file defines the `servent` structure and structure fields.

Use the `endservent` subroutine to close the `/etc/services` file.

All applications containing the `getservbyname` subroutine must be compiled with the `_BSD` macro set to a specific value. Acceptable values are 43 and 44. In addition, all socket applications must include the BSD `libbsd.a` library.

**Parameters**

- **Name**: Specifies the name of a service.
- **Protocol**: Specifies a protocol for use with the specified service.

**Return Values**

The `getservbyname` subroutine returns a pointer to a `servent` structure when a successful match occurs. Entries in this structure are in network byte order.

If an error occurs or the end of the file is reached, the `getservbyname` subroutine returns a null pointer.

**Files**

- **/etc/services**: Contains service names.

**Related Information**

The `endprotoent`, `endservent`, `getprotobyname`, `getprotobynumber`, `getprotoent`, `getservbyport`, `getservent`, `setprotoent`, `setservent` subroutines.

[Sockets Overview](#) and [Understanding Network Address Translation](#) in *AIX 5L Version 5.3 Communications Programming Concepts*.

**getservbyname_r Subroutine**

**Purpose**

 Gets service entry by name.

**Library**

Standard C Library (`libc.a`)
Syntax

```c
#include <netdb.h>
int getservbyname_r(name, proto, servent, serv_data)
const char *Name, *proto;
struct servent *servent;
struct servent_data *serv_data;
```

Description

An application program can use the `getservbyname_r` subroutine to access a service, service aliases, the protocol for the service, and a protocol port number for the service.

The `getservbyname_r` subroutine searches the `/etc/services` file sequentially from the start of the file until it finds one of the following:
- Matching name and protocol number.
- Matching name when the `Protocol` parameter is set to 0.
- End of the file.

Upon locating a matching name and protocol, the `getservbyname_r` subroutine stores the values to the `servent` structure. The `getservbyname` subroutine overwrites the static data it returns in subsequent calls. The `getservbyname_r` subroutine does not.

Use the `endservent_r` subroutine to close the `/etc/hosts` file.

Parameters

- `name`: Specifies the name of a service.
- `proto`: Specifies a protocol for use with the specified service.
- `servent`: Points to the `servent` structure.
- `serv_data`: Points to the `servent_data` structure.

Return Values

The function returns a 0 if successful and a -1 if unsuccessful. The `getservbyname` subroutine returns a pointer to a servent structure when a successful match occurs. Entries in this structure are in network byte order.

Note: If an error occurs or the end of the file is reached, the `getservbyname_r` returns a -1.

Files

- `/etc/services`: Contains service names.

Related Information

- "endservent_r Subroutine" on page 47, "setservent_r Subroutine" on page 204, "getservent_r Subroutine" on page 98, and "getservbyport_r Subroutine" on page 96.

getservbyport Subroutine

Purpose

Gets service entry by port.
Library
Standard C Library (libc.a)

Syntax
#include <netdb.h>

struct servent *getservbyport (Port, Protocol);
int Port; char *Protocol;

Description
The getservbyport subroutine is threadsafe in AIX 4.3 and later. However, the return value points to static
data that is overwritten by subsequent calls. This data must be copied to be saved for use by subsequent
calls.

The getservbyport subroutine retrieves an entry from the /etc/services file using a port number as a
search key.

An application program can use the getservbyport subroutine to access a service, service aliases, the
protocol for the service, and a protocol port number for the service.

The getservbyport subroutine searches the services file sequentially from the beginning of the file until it
finds one of the following:
- Matching protocol and port number
- Matching protocol when the Port parameter value equals 0
- End of the file

Upon locating a matching protocol and port number or upon locating a matching protocol only if the Port
parameter value equals 0, the getservbyport subroutine returns a pointer to a servent structure, which
contains fields for a line of information in the /etc/services file. The netdb.h file defines the servent
structure and structure fields.

Use the endservent subroutine to close the /etc/services file.

All applications containing the getservbyport subroutine must be compiled with the _BSD macro set to a
specific value. Acceptable values are 43 and 44. In addition, all socket applications must include the BSD
libbsd.a library.

Parameters
Port Specifies the port where a service resides.
Protocol Specifies a protocol for use with the service.

Return Values
Upon successful completion, the getservbyport subroutine returns a pointer to a servent structure.

If an error occurs or the end of the file is reached, the getservbyport subroutine returns a null pointer.

Files
/etc/services Contains service names.
Related Information

The endprotoent subroutine, endservent subroutine, getprotobynumber subroutine, getprotoent subroutine, getprotobyname subroutine, getservbyname subroutine, getservbyport subroutine, getservbyname subroutine, getservent subroutine, getservbyport subroutine, endservent subroutine.

Sockets Overview in AIX 5L Version 5.3 Communications Programming Concepts.

getservbyport_r Subroutine

Purpose
Gets service entry by port.

Library
Standard C Library (libc.a)

Syntax

```c
#include <netdb.h>

int getservbyport_r(Port, Proto, servent, serv_data)
int Port;
const char *Proto;
struct servent *servent;
struct servent_data *serv_data;
```

Description
The getservbyport_r subroutine retrieves an entry from the /etc/services file using a port number as a search key. An application program can use the getservbyport_r subroutine to access a service, service aliases, the protocol for the service, and a protocol port number for the service.

The getservbyport_r subroutine searches the services file sequentially from the beginning of the file until it finds one of the following:
- Matching protocol and port number
- Matching protocol when the Port parameter value equals 0
- End of the file

Upon locating a matching protocol and port number or upon locating a matching protocol where the Port parameter value equals 0, the getservbyport_r subroutine returns a pointer to a servent structure, which contains fields for a line of information in the /etc/services file. The netdb.h file defines the servent structure, the servent_data structure, and their fields.

The getservbyport routine overwrites static data returned on subsequent calls. The getservbyport_r routine does not.

Use the endservent_r subroutine to close the /etc/services file.

Parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Port</td>
<td>Specifies the port where a service resides.</td>
</tr>
<tr>
<td>Proto</td>
<td>Specifies a protocol for use with the service.</td>
</tr>
<tr>
<td>servent</td>
<td>Points to the servent structure.</td>
</tr>
<tr>
<td>serv_data</td>
<td>Points to the servent_data structure.</td>
</tr>
</tbody>
</table>
Return Values
The function returns a 0 if successful and a -1 if unsuccessful.

Note: If an error occurs or the end of the file is reached, the getservbyport_r subroutine returns a -1 to indicate error.

Files
/etc/services Contains service names.

Related Information
“endservent_r Subroutine” on page 47, “setservent_r Subroutine” on page 204, “getservent_r Subroutine” on page 98, and “getservbyname_r Subroutine” on page 93.

getservent Subroutine

Purpose
Gets services file entry.

Library
Standard C Library (libc.a)

Syntax
#include <netdb.h>
struct servent *getservent()

Description
The getservent subroutine is threadsafe in AIX 4.3 and later. However, the return value points to static data that is overwritten by subsequent calls. This data must be copied to be saved for use by subsequent calls.

The getservent subroutine opens and reads the next line of the /etc/services file.

An application program can use the getservent subroutine to retrieve information about network services and the protocol ports they use.

The getservent subroutine returns a pointer to a servent structure, which contains fields for a line of information from the /etc/services file. The servent structure is defined in the netdb.h file.

The /etc/services file remains open after a call by the getservent subroutine. To close the /etc/services file after each call, use the setservent subroutine. Otherwise, use the endservent subroutine to close the /etc/services file.

All applications containing the getservent subroutine must be compiled with the _BSD macro set to a specific value. Acceptable values are 43 and 44. In addition, all socket applications must include the BSD libbsd.a library.

Return Values
The getservent subroutine returns a pointer to a servent structure when a successful match occurs.

If an error occurs or the end of the file is reached, the getservent subroutine returns a null pointer.
Files
/etc/services

Contains service names.

Related Information
The endprotoent subroutine, endservent subroutine, getprotobynumber subroutine, getprotobynumber subroutine, getservbyname subroutine, getservbyport subroutine, getprotoent subroutine, setprotoent subroutine.

Sockets Overview and Understanding Network Address Translation in AIX 5L Version 5.3 Communications Programming Concepts.

getservent_r Subroutine

Purpose
Gets services file entry.

Library
Standard C Library (libc.a)

Syntax
#include <netdb.h>
int getservent_r(servent, serv_data)
struct servent *servent;
struct servent_data *serv_data;

Description
The getservent_r subroutine opens and reads the next line of the /etc/services file. An application program can use the getservent_r subroutine to retrieve information about network services and the protocol ports they use.

The /etc/services file remains open after a call by the getservent_r subroutine. To close the /etc/services file after each call, use the setservent_r subroutine. Otherwise, use the endservent_r subroutine to close the /etc/services file.

Parameters
servent
Points to the servent structure.
serv_data
Points to the serv_data structure.

Return Values
The getservent_r fails when a successful match occurs. The getservent subroutine overwrites static data returned on subsequent calls. The getservent_r subroutine does not.

Files
/etc/services

Contains service names.
getsockname Subroutine

Purpose
Gets the socket name.

Library
Standard C Library (libc.a)

Syntax

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

int getsockname (Socket, Name, NameLength)
int Socket;
struct sockaddr *Name;
socklen_t *NameLength;
```

Description

The `getsockname` subroutine retrieves the locally bound address of the specified socket. The socket address represents a port number in the Internet domain and is stored in the `sockaddr` structure pointed to by the `Name` parameter. The `sys/socket.h` file defines the `sockaddr` data structure.

**Note:** The `getsockname` subroutine does not perform operations on UNIX domain sockets.

A process created by another process can inherit open sockets. To use the inherited socket, the created process needs to identify their addresses. The `getsockname` subroutine allows a process to retrieve the local address bound to the specified socket.

A process can use the `getpeername` subroutine to determine the address of a destination socket in a socket connection.

All applications containing the `getsockname` subroutine must be compiled with the `_BSD` macro set to a specific value. Acceptable values are 43 and 44. In addition, all socket applications must include the BSD `libbsd.a` library.

Parameters

- **Socket**
  Specifies the socket for which the local address is desired.

- **Name**
  Points to the structure containing the local address of the specified socket.

- **NameLength**
  Specifies the size of the local address in bytes. Initializes the value pointed to by the `NameLength` parameter to indicate the amount of space pointed to by the `Name` parameter.

Return Values

Upon successful completion, a value of 0 is returned, and the `NameLength` parameter points to the size of the socket address.

If the `getsockname` subroutine is unsuccessful, the subroutine handler performs the following functions:

- Returns a value of -1 to the calling program.
Moves an error code, indicating the specific error, into the `errno` global variable.

**Error Codes**
The `getsockopt` subroutine is unsuccessful if any of the following errors occurs:

<table>
<thead>
<tr>
<th>Error</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EBADF</td>
<td>The <code>Socket</code> parameter is not valid.</td>
</tr>
<tr>
<td>ENOTSOCK</td>
<td>The <code>Socket</code> parameter refers to a file, not a socket.</td>
</tr>
<tr>
<td>ENOBUFS</td>
<td>Insufficient resources are available in the system to complete the call.</td>
</tr>
<tr>
<td>EFAULT</td>
<td>The <code>Address</code> parameter is not in a writable part of the user address space.</td>
</tr>
</tbody>
</table>

**Related Information**
The `accept` subroutine, `bind` subroutine, `getpeername` subroutine, `socket` subroutine.

Checking for Pending Connections Example Program, Reading Internet Datagrams Example Program, and Sockets Overview in AIX 5L Version 5.3 Communications Programming Concepts.

**getsockopt Subroutine**

**Purpose**
Gets options on sockets.

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

**Syntax**
```
#include <sys/socket.h>

int getsockopt (Socket, Level, OptionName, OptionValue, OptionLength)

int Socket,
Level,
OptionName,
OptionValue,
OptionLength;

void * OptionValue;
socklen_t * OptionLength;
```

**Description**
The `getsockopt` subroutine allows an application program to query socket options. The calling program specifies the name of the socket, the name of the option, and a place to store the requested information. The operating system gets the socket option information from its internal data structures and passes the requested information back to the calling program.

Options can exist at multiple protocol levels. They are always present at the uppermost socket level. When retrieving socket options, specify the level where the option resides and the name of the option.

All applications containing the `getsockopt` subroutine must be compiled with the `_BSD` macro set to a specific value. Acceptable values are 43 and 44. In addition, all socket applications must include the BSD `libbsd.a` library.

**Parameters**

- `Socket`: Specifies the unique socket name.
**Level**

Specifies the protocol level where the option resides. Options can be retrieved at the following levels:

**Socket level**

Specifies the Level parameter as the SOL_SOCKET option.

**Other levels**

Supplies the appropriate protocol number for the protocol controlling the option. For example, to indicate that an option will be interpreted by the TCP protocol, set the Level parameter to the protocol number of TCP, as defined in the netinet/in.h file.

**OptionName**

Specifies a single option. The OptionName parameter and any specified options are passed uninterpreted to the appropriate protocol module for interpretation. The sys/socket.h file contains definitions for socket level options. The netinet/tcp.h file contains definitions for TCP protocol level options. Socket-level options can be enabled or disabled; they operate in a toggle fashion. The sys/atmsock.h file contains definitions for ATM protocol level options.

The following list defines socket protocol level options found in the sys/socket.h file:

**SO_DEBUG**

Specifies the recording of debugging information. This option enables or disables debugging in the underlying protocol modules.

**SO_BROADCAST**

Specifies whether transmission of broadcast messages is supported. The option enables or disables broadcast support.

**SO_CKSUMREV**

Enables performance enhancements in the protocol layers. If the protocol supports this option, enabling causes the protocol to defer checksum verification until the user's data is moved into the user's buffer (on recv, recvfrom, read, or recvmsg thread). This can cause applications to be awakened when no data is available, in the case of a checksum error. In this case, EAGAIN is returned. Applications that set this option must handle the EAGAIN error code returned from a receive call.

**SO_REUSEADDR**

Specifies that the rules used in validating addresses supplied by a bind subroutine should allow reuse of a local port. A particular IP address can only be bound once to the same port. This option enables or disables reuse of local ports.

**SO_REUSEADDR** allows an application to explicitly deny subsequent bind subroutine to the port/address of the socket with SO_REUSEADDR set. This allows an application to block other applications from binding with the bind subroutine.

**SO_REUSEPORT**

Specifies that the rules used in validating addresses supplied by a bind subroutine should allow reuse of a local port/address combination. Each binding of the port/address combination must specify the SO_REUSEPORT socket option. This option enables or disables the reuse of local port/address combinations.

**SO_KEEPALIVE**

Monitors the activity of a connection by enabling or disabling the periodic transmission of ACK messages on a connected socket. The idle interval time can be designated using the TCP/IP no command. Broken connections are discussed in "Understanding Socket Types and Protocols" in AIX 5L Version 5.3 Communications Programming Concepts.
SO_DONTROUTE
Indicates outgoing messages should bypass the standard routing facilities. Does not apply routing on outgoing messages. Directs messages to the appropriate network interface according to the network portion of the destination address. This option enables or disables routing of outgoing messages.

SO_LINGER
Lingers on a close subroutine if data is present. This option controls the action taken when an unsent messages queue exists for a socket, and a process performs a close subroutine on the socket.

If the SO_LINGER option is set, the system blocks the process during the close subroutine until it can transmit the data or until the time expires. If the SO_LINGER option is not specified, and a close subroutine is issued, the system handles the call in a way that allows the process to continue as quickly as possible.

The sys/socket.h file defines the linger structure that contains the l_linger member for specifying linger time interval. If linger time is set to anything but 0, the system tries to send any messages queued on the socket. The maximum value that the l_linger member can be set to is 65535. If the application has requested SPEC1170 compliant behavior by exporting the XPG_SUS_ENV environment variable, the linger time is n seconds; otherwise, the linger time is n/100 seconds (ticks), where n is the value of the l_linger member.

SO_OOBINLINE
Leaves received out-of-band data (data marked urgent) in line. This option enables or disables the receipt of out-of-band data.

SO_SNDBUF
Retrieves buffer size information.

SO_RCVBUF
Retrieves buffer size information.

SO_SNDLOWAT
Retrieves send buffer low-water mark information.

SO_RCVLOWAT
Retrieves receive buffer low-water mark information.
SO_SNDBTIMEO
Retrieves time-out information. This option is settable, but currently not used.

SO_RCVTIMEO
Retrieves time-out information. This option is settable, but currently not used.

SO_PEERID
Retrieves the credential information of the process associated with a peer UNIX domain socket. This information includes the process ID, effective user ID, and effective group ID. The peercred_struct structure must be used in order to get the credential information. This structure is defined in the sys/socket.h file.

SO_ERROR
Retrieves information about error status and clears.

The following list defines TCP protocol level options found in the netinet/tcp.h file:

TCP_RFC1323
Indicates whether RFC 1323 is enabled or disabled on the specified socket. A non-zero OptionValue returned by the getsockopt subroutine indicates the RFC is enabled.

TCP_NODELAY
Specifies whether TCP should follow the Nagle algorithm for deciding when to send data. By default TCP will follow the Nagle algorithm. To disable this behavior, applications can enable TCP_NODELAY to force TCP to always send data immediately. A non-zero OptionValue returned by the getsockopt subroutine indicates TCP_NODELAY is enabled. For example, TCP_NODELAY should be used when there is an application using TCP for a request/response.

TCP_NODELAYACK
Specifies if TCP needs to send immediate acknowledgement packets to the sender. If this option is not set, TCP delays sending the acknowledgement packets by up to 200 ms. This allows the acknowledgements to be sent along with the data on a response and minimizes system overhead. Setting this TCP option might cause a slight increase in system overhead, but can result in higher performance for network transfers if the sender is waiting on the receiver's acknowledgements.
The following list defines ATM protocol level options found in the `sys/atmsock.h` file:

**SO_ATM_PARM**
Retrieves all ATM parameters. This socket option can be used instead of using individual sockets options described below. It uses the `connect_ie` structure defined in `sys/call_ie.h` file.

**SO_ATM_AAL_PARM**
Retrieves ATM AAL (Adaptation Layer) parameters. It uses the `aal_parm` structure defined in `sys/call_ie.h` file.

**SO_ATM_TRAFFIC_DES**
Retrieves ATM Traffic Descriptor values. It uses the `traffic_desc` structure defined in `sys/call_ie.h` file.

**SO_ATM_BEARER**
Retrieves ATM Bearer capability information. It uses the `bearer` structure defined in `sys/call_ie.h` file.

**SO_ATM_BHLI**
Retrieves ATM Broadband High Layer Information. It uses the `bhli` structure defined in `sys/call_ie.h` file.

**SO_ATM_BLLI**
Retrieves ATM Broadband Low Layer Information. It uses the `blli` structure defined in `sys/call_ie.h` file.

**SO_ATM_QoS**
Retrieves ATM Quality Of Service values. It uses the `qos_parm` structure defined in `sys/call_ie.h` file.

**SO_ATM_TRANSIT_SEL**
Retrieves ATM Transit Selector Carrier. It uses the `transit_sel` structure defined in `sys/call_ie.h` file.

**SO_ATM_MAX_PEND**
Retrieves the number of outstanding transmit buffers that are permitted before an error indication is returned to applications as a result of a transmit operation. This option is only valid for non best effort types of virtual circuits.

**SO_ATM_CAUSE**
Retrieves cause for the connection failure. It uses the `cause_t` structure defined in the `sys/call_ie.h` file.

*OptionValue* Specifies a pointer to the address of a buffer. The *OptionValue* parameter takes an integer parameter. The *OptionValue* parameter should be set to a nonzero value to enable a Boolean option or to a value of 0 to disable the option. The following options enable and disable in the same manner:

- SO_DEBUG
- SO_REUSEADDR
- SO_KEEPALIVE
- SO_DONTROUTE
- SO_BROADCAST
- SO_OOBINLINE
- TCP_RFC1323

*OptionLength* Specifies the length of the *OptionValue* parameter. The *OptionLength* parameter initially contains the size of the buffer pointed to by the *OptionValue* parameter. On return, the *OptionLength* parameter is modified to indicate the actual size of the value returned. If no option value is supplied or returned, the *OptionValue* parameter can be 0.

Options at other protocol levels vary in format and name.
IP level (IPPROTO_IP level) options are defined as follows:

**IP_DONTFRAG**  Get current IP_DONTFRAG option value.
**IP_PICKDND**  Get current PMTU value.
**IP_PMTUAGE**  Get current PMTU time out value.

In the case of TCP protocol sockets:

**IP_DONTFRAG**  Not supported.
**IP_PICKDND**  Get current PMTU value.
**IP_PMTUAGE**  Not supported.

IPV6 level (IPPROTO_IPV6 level) options are defined as follows:

<table>
<thead>
<tr>
<th>Option</th>
<th>Description</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>IPV6_V6ONLY</td>
<td>Determines whether the socket is restricted to IPV6 communications only.</td>
<td>int (boolean interpretation)</td>
</tr>
<tr>
<td>IPV6_UNICAST_HOPS</td>
<td>Allows the user to determine the outgoing hop limit value for unicast IPV6 packets.</td>
<td>int</td>
</tr>
<tr>
<td>IPV6MULTICAST_HOPS</td>
<td>Allows the user to determine the outgoing hop limit value for multicast IPV6 packets.</td>
<td>int</td>
</tr>
<tr>
<td>IPV6_MULTICAST_IF</td>
<td>Allows the user to determine the interface being used for outgoing multicast packets.</td>
<td>unsigned int</td>
</tr>
<tr>
<td>IPV6_MULTICAST_LOOP</td>
<td>If a multicast datagram is sent to a group that the sending host belongs, a copy of the datagram is looped back by the IP layer for local delivery (if the option is set to 1). If the option is set to 0, a copy is not looped back.</td>
<td>unsigned int</td>
</tr>
<tr>
<td>IPV6_RECVPKTINFO</td>
<td>Determines whether the destination IPv6 address and arriving interface index of incoming IPv6 packets are being received as ancillary data on UDP and raw sockets.</td>
<td>int (boolean interpretation)</td>
</tr>
<tr>
<td>IPV6_RECVHOLIMIT</td>
<td>Determines whether the hop limit of incoming IPv6 packets is being received as ancillary data on UDP and raw sockets.</td>
<td>int (boolean interpretation)</td>
</tr>
<tr>
<td>IPV6_RECVTCCLASS</td>
<td>Determines whether the traffic class of incoming IPv6 packets is being received as ancillary data on UDP and raw sockets.</td>
<td>int (boolean interpretation)</td>
</tr>
<tr>
<td>IPV6_RECVRTHDR</td>
<td>Determines whether the routing header of incoming IPv6 packets is being received as ancillary data on UDP and raw sockets.</td>
<td>int (boolean interpretation)</td>
</tr>
<tr>
<td>IPV6_RECVHOPOPTS</td>
<td>Determines whether the hop-by-hop options header of incoming IPv6 packets is being received as ancillary data on UDP and raw sockets.</td>
<td>int (boolean interpretation)</td>
</tr>
<tr>
<td>IPV6_RECVDSTOPTS</td>
<td>Determines whether the destination options header of incoming IPv6 packets is being received as ancillary data on UDP and raw sockets.</td>
<td>int (boolean interpretation)</td>
</tr>
<tr>
<td>IPV6_PKTINFO</td>
<td>Determines the source IPv6 address and outgoing interface index for all IPv6 packets being sent on this socket.</td>
<td>struct in6_pktinfo defined in the netinet/in.h file.</td>
</tr>
<tr>
<td>Option</td>
<td>Description</td>
<td>Type</td>
</tr>
<tr>
<td>---------------------</td>
<td>----------------------------------------------------------------------------------------------------------------------------------------------</td>
<td>-----------------</td>
</tr>
<tr>
<td>IPV6_NEXTHOP</td>
<td>Determines the next hop being used for outgoing IPv6 datagrams on this socket.</td>
<td>struct sockaddr_in6 defined in the netinet/in.h file.</td>
</tr>
<tr>
<td>IPV6_TCLASS</td>
<td>Determines the traffic class for outgoing IPv6 datagrams on this socket.</td>
<td>int</td>
</tr>
<tr>
<td>IPV6_RTHDR</td>
<td>Determines the routing header to be used for outgoing IPv6 datagrams on this socket.</td>
<td>struct ip6_rthdr defined in the netinet/ip6.h file.</td>
</tr>
<tr>
<td>IPV6_HOPOPTS</td>
<td>Determines the hop-by-hop options header to be used for outgoing IPv6 datagrams on this socket.</td>
<td>struct ip6_hbh defined in the netinet/ip6.h file.</td>
</tr>
<tr>
<td>IPV6_DSTOPTS</td>
<td>Determines the destination options header to be used for outgoing IPv6 datagrams on this socket. This header will follow a routing header (if present) and will also be used when there is no routing header specified.</td>
<td>struct ip6_dest defined in the netinet/ip6.h file.</td>
</tr>
<tr>
<td>IPV6_RTHDRDSTOPTS</td>
<td>Determines the destination options header to be used for outgoing IPv6 datagrams on this socket. This header will precede a routing header (if present). If no routing header is specified, this option will be silently ignored.</td>
<td>struct ip6_dest defined in the netinet/ip6.h file.</td>
</tr>
<tr>
<td>IPV6_USE_MIN_MTU</td>
<td>Determines how IPv6 path MTU discovery is being controlled for this socket.</td>
<td>int</td>
</tr>
<tr>
<td>IPV6_DONTFRAG</td>
<td>Determines whether fragmentation of outgoing IPv6 packets has been disabled on this socket.</td>
<td>int (boolean interpretation)</td>
</tr>
<tr>
<td>IPV6_RECVPATHMTU</td>
<td>Determines whether IPv6_PATHMTU messages are being received as ancillary data on this socket.</td>
<td>int (boolean interpretation)</td>
</tr>
<tr>
<td>IPV6_PATHMTU</td>
<td>Determines the current Path MTU for a connected socket.</td>
<td>struct ip6_mtuinfo defined in the netinet/in.h file.</td>
</tr>
</tbody>
</table>

ICMPV6 level (IPPROTO_ICMPV6 level) options are defined as follows:

<table>
<thead>
<tr>
<th>Option</th>
<th>Description</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>ICMP6_FILTER</td>
<td>Allows the user to filter ICMPV6 messages by the ICMPV6 type field. If no filter was set, the default kernel filter will be returned.</td>
<td>The icmp6_filter structure defined in the netinet/icmp6.h file.</td>
</tr>
</tbody>
</table>

Return Values

Upon successful completion, the `getsockopt` subroutine returns a value of 0.

If the `getsockopt` subroutine is unsuccessful, the subroutine handler performs the following actions:
- Returns a value of -1 to the calling program.
- Moves an error code, indicating the specific error, into the `errno` global variable.

Upon successful completion of the IPPROTO_IP option IP_PMTUAGE the returns are:

Prior to AIX 5.3:
- OptionValue 0 if PMTU discovery is not enabled.
- OptionValue -1 if PMTU discovery is not complete.
Positive non-zero OptionValue if PMTU is available.

Beginning with AIX 5.3:
• Positive non-zero OptionValue.

Upon successful completion of TCP protocol sockets option IP_FNDPMTU the returns are:

Prior to AIX 5.3:
• OptionValue 0 if PMTU discovery (tcp_pmtu_discover) is not enabled.
• OptionValue -1 if PMTU discovery is not complete/not available.
• Positive non-zero OptionValue if PMTU is available.

Beginning with AIX 5.3:
• OptionValue 0 if PMTU discovery (tcp_pmtu_discover) is not enabled/not available.
• Positive non-zero OptionValue if PMTU is available.

Error Codes
The getsockopt subroutine is unsuccessful if any of the following errors occurs:

EBADF  The Socket parameter is not valid.
EFAULT  The address pointed to by the OptionValue parameter is not in a valid (writable) part of the process space, or the OptionLength parameter is not in a valid part of the process address space.
EINVAL  The Level, OptionName, or OptionLength is invalid.
ENOBUFF  Insufficient resources are available in the system to complete the call.
ENOTSOCK  The Socket parameter refers to a file, not a socket.
ENOPROTOOPT  The option is unknown.
EOPNOTSUPP  The option is not supported by the socket family or socket type.

Examples
The following program fragment illustrates the use of the getsockopt subroutine to determine an existing socket type:

```c
#include <sys/types.h>
#include <sys/socket.h>
int type, size;
size = sizeof(int);
if(getsockopt(s, SOL_SOCKET, SO_TYPE, (char*)&type,&size)<0){
    //
    //
}
```

Related Information
The no command.

The bind subroutine, close subroutine, endprotoent subroutine, getprotobynumber subroutine, getprotoent subroutine, setprotoent subroutine, setsockopt subroutine, socket subroutine.

Sockets Overview, Understanding Socket Options, and Understanding Socket Types and Protocols in AIX 5L Version 5.3 Communications Programming Concepts.
htonl Subroutine

Purpose
Converts an unsigned long integer from host byte order to Internet network byte order.

Library
ISODE Library (libisode.a)

Syntax
```
#include <sys/types.h>
#include <netinet/in.h>

uint32_t htonl (HostLong)
uint32_t HostLong;
```

Description
The htonl subroutine converts an unsigned long (32-bit) integer from host byte order to Internet network byte order.

The Internet network requires addresses and ports in network standard byte order. Use the htonl subroutine to convert the host integer representation of addresses and ports to Internet network byte order.

The htonl subroutine is defined in the net/nh.h file as a null macro if the host byte order is the same as the network byte order.

The htonl subroutine is declared in the net/nh.h file as a function if the host byte order is not the same as the network byte order.

All applications containing the htonl subroutine must be compiled with the __BSD macro set to a specific value. Acceptable values are 43 and 44. In addition, all socket applications must include the BSD libbsd.a library.

Parameters
HostLong Specifies a 32-bit integer in host byte order.

Return Values
The htonl subroutine returns a 32-bit integer in Internet network byte order (most significant byte first).

Related Information
The htons subroutine, ntohs subroutine, ntohs subroutine, htonl subroutine, ntohll subroutine.

Sockets Overview in AIX 5L Version 5.3 Communications Programming Concepts.

htonll Subroutine

Purpose
Converts an unsigned long integer from host byte order to Internet network byte order.
Library
ISODE Library (libisode.a)

Syntax
#include <sys/types.h>
#include <netinet/in.h>

uint64_t htonl1 (HostLong)
uint64_t HostLong;

Description
The htonl1 subroutine converts an unsigned long (64-bit) integer from host byte order to Internet network byte order.

The Internet network requires addresses and ports in network standard byte order. Use the htonl1 subroutine to convert the host integer representation of addresses and ports to Internet network byte order.

The htonl1 subroutine is defined in the net/nh.h file as a null macro if the host byte order is the same as the network byte order.

The htonl1 subroutine is declared in the net/nh.h file as a function if the host byte order is not the same as the network byte order.

All applications containing the htonl1 subroutine must be compiled with the _BSD macro set to a specific value. Acceptable values are 43 and 44. In addition, all socket applications must include the BSD libbsd.a library.

Parameters
HostLong    Specifies a 64-bit integer in host byte order.

Return Values
The htonl1 subroutine returns a 64-bit integer in Internet network byte order (most significant byte first).

Related Information
The htons subroutine, ntohl subroutine, ntohs subroutine, htonl1 subroutine, ntohl1 subroutine.

Sockets Overview in AIX 5L Version 5.3 Communications Programming Concepts.

htons Subroutine

Purpose
Converts an unsigned short integer from host byte order to Internet network byte order.

Library
ISODE Library (libisode.a)

Syntax
#include <sys/types.h>
#include <netinet/in.h>
uint16_t htons (HostShort)
uint16_t HostShort;

Description
The htons subroutine converts an unsigned short (16-bit) integer from host byte order to Internet network byte order.

The Internet network requires ports and addresses in network standard byte order. Use the htons subroutine to convert addresses and ports from their host integer representation to network standard byte order.

The htons subroutine is defined in the net/nh.h file as a null macro if the host byte order is the same as the network byte order.

The htons subroutine is declared in the net/nh.h file as a function if the host byte order is not the same as the network byte order.

All applications containing the htons subroutine must be compiled with the _BSD macro set to a specific value. Acceptable values are 43 and 44. In addition, all socket applications must include the BSD libbsd.a library.

Parameters
HostShort Specifies a 16-bit integer in host byte order that is a host address or port.

Return Values
The htons subroutine returns a 16-bit integer in Internet network byte order (most significant byte first).

Related Information
The htonl subroutine, ntohl subroutine, ntohs subroutine, htonl subroutine, ntohl subroutine.
Sockets Overview in AIX 5L Version 5.3 Communications Programming Concepts.

if_freenameindex Subroutine

Purpose
Frees the dynamic memory that was allocated by the if_nameindex Subroutine on page 112.

Library
Library (libc.a)

Syntax
#include <net/if.h>

void if_freenameindex (struct if_nameindex *ptr);

Description
The ptr parameter is a pointer returned by the if_nameindex subroutine. After the if_freenameindex subroutine has been called, the application must not use the array of which ptr is the address.
Parameters

ptr

Pointer returned by the if_nameindex subroutine

Related Information

"if_nameindex Subroutine" on page 112, "if_indextoname Subroutine," and "if_nameindex Subroutine" on page 112.

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

if_indextoname Subroutine

Purpose
Maps an interface index into its corresponding name.

Library
Standard C Library <libc.a>

Syntax
#include <net/if.h>
char *if_indextoname(unsigned int ifindex, char *ifname);

Description
When the if_indextoname subroutine is called, the ifname parameter points to a buffer of at least IF_NAMESIZE bytes. The if_indextoname subroutine places the name of the interface in this buffer with the ifindex index.

Note: IF_NAMESIZE is also defined in <net/if.h> and its value includes a terminating null byte at the end of the interface name.

If ifindex is an interface index, the if_indextoname Subroutine returns the ifname value, which points to a buffer containing the interface name. Otherwise, it returns a NULL pointer and sets the errno global value to indicate the error.

If there is no interface corresponding to the specified index, the errno global value is set to ENXIO. If a system error occurs (such as insufficient memory), the errno global value is set to the proper value (such as, ENOMEM).

Parameters

ifindex
Possible interface index

ifname
Possible name of an interface

Error Codes

ENXIO 
There is no interface corresponding to the specified index

ENOMEM 
Insufficient memory
if_nameindex Subroutine

Purpose
Retrieves index and name information for all interfaces.

Library
The Standard C Library (<libc.a>)

Syntax
#include <net/if.h>

struct if_nameindex *if_nameindex(void)
struct if_nameindex {
  unsigned int if_index; /* 1, 2, ... */
  char *if_name; /* null terminated name: "le0", ... */
};

Description
The if_nameindex subroutine returns an array of if_nameindex structures (one per interface).

The memory used for this array of structures is obtained dynamically. The interface names pointed to by the if_name members are obtained dynamically as well. This memory is freed by the if_freeindex subroutine.

The function returns a NULL pointer upon error, and sets the errno global value to the appropriate value. If successful, the function returns an array of structures. The end of an array of structures is indicated by a structure with an if_index value of 0 and an if_name value of NULL.

Related Information
"if_nametoindex Subroutine," "if_indextoname Subroutine" on page 111, and "if_freenameindex Subroutine" on page 110.

if_nametoindex Subroutine

Purpose
Maps an interface name to its corresponding index.

Library
Standard C Library (libc.a)
Syntax

```c
#include <net/if.h>
unsigned int if_nametoindex(const char *ifname);
```

Description

If the `ifname` parameter is the name of an interface, the `if_nametoindex` subroutine returns the interface index corresponding to the `ifname` name. If the `ifname` parameter is not the name of an interface, the `if_nametoindex` subroutine returns a 0 and the `errno` global variable is set to the appropriate value.

Parameters

`ifname` Possible name of an interface.

Related Information

“if_indextoname Subroutine” on page 111, “if_nameindex Subroutine” on page 112, and “if_frenameindex Subroutine” on page 110.

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

**inet6_opt_append Subroutine**

**Purpose**

Returns the updated total length of the extension header.

**Syntax**

```c
int inet6_opt_append(void *extbuf, socklen_t extlen, int offset,
                     uint8_t type, socklen_t len, uint_t align,
                     void **databufp);
```

**Description**

The `inet6_opt_append` subroutine returns the updated total length of the extension header, taking into account adding an option with length `len` and alignment `align`. If `extbuf` is not NULL, then, in addition to returning the length, the subroutine inserts any needed pad option, initializes the option (setting the type and length fields), and returns a pointer to the location for the option content in `databufp`. After `inet6_opt_append()` has been called, the application can use the `databuf` directly, or use `inet6_opt_set_val()` to specify the content of the option.

**Parameters**

- `extbuf` If NULL, `inet6_opt_append` will return only the updated length. If `extbuf` is not NULL, in addition to returning the length, the function inserts any needed pad option, initializes the option (setting the `type` and `len` fields) and returns a pointer to the location for the option content in `databufp`.
- `extlen` Size of the buffer pointed to by `extbuf`.
- `offset` The length returned by `inet6_opt_init()` or a previous `inet6_opt_append()`.
- `type` 8-bit option type. Must have a value from 2 to 255, inclusive. (0 and 1 are reserved for the `Pad1` and `PadN` options, respectively.)
**Return Values**

-1  Option content does not fit in the extension header buffer.
integer value  Updated total length of the extension header.

**Related Information**

"inet6_opt_find Subroutine," "inet6_opt_finish Subroutine" on page 115, "inet6_opt_get_val Subroutine" on page 115, "inet6_opt_init Subroutine" on page 116, "inet6_opt_next Subroutine" on page 117, "inet6_opt_set_val Subroutine" on page 118, "inet6_rth_add Subroutine" on page 118, "inet6_rth_getaddr Subroutine" on page 119, "inet6_rth_init Subroutine" on page 119, "inet6_rth_reverse Subroutine" on page 120, "inet6_rth_segments Subroutine" on page 121, "inet6_rth_space Subroutine" on page 122

**inet6_opt_find Subroutine**

**Purpose**

Looks for a specified option in the extension header.

**Syntax**

```c
int inet6_opt_find(void *extbuf, socklen_t extlen, int offset, uint8_t *typep, socklen_t *lenp, void **databufp);
```

**Description**

The `inet6_opt_find` subroutine is similar to the `inet6_opt_next()` function, except this subroutine lets the caller specify the option type to be searched for, instead of always returning the next option in the extension header.

**Parameters**

- `extbuf`  Specifies the extension header.
- `extlen`  Size of the buffer pointed to by `extbuf`.
- `offset`  Specifies the position where scanning of the extension buffer can continue. Should either be 0 (for the first option) or the length returned by a previous call to `inet6_opt_next()` or `inet6_opt_find()`.
- `typep`  Stores the option type.
- `lenp`  Stores the length of the option data (excluding the option type and option length fields).
- `databufp`  Points to the data field of the option.

**Return Values**

The `inet6_opt_find` subroutine returns the updated “previous” total length computed by advancing past the option that was returned and past any options that did not match the type. This returned “previous”
length can then be passed to subsequent calls to `inet6_opt_find()` for finding the next occurrence of the same option type.

−1 The option cannot be located, there are no more options, or the option extension header is malformed.

**Related Information**

- “inet6_opt_append Subroutine” on page 113
- “inet6_opt_finish Subroutine”
- “inet6_opt_get_val Subroutine”
- “inet6_opt_init Subroutine” on page 116
- “inet6_opt_next Subroutine” on page 117
- “inet6_opt_set_val Subroutine” on page 118
- “inet6_rth_add Subroutine” on page 118
- “inet6_rth_getaddr Subroutine” on page 119
- “inet6_rth_init Subroutine” on page 119
- “inet6_rth_reverse Subroutine” on page 120
- “inet6_rth_segments Subroutine” on page 121
- “inet6_rth_space Subroutine” on page 122

---

### inet6_opt_finish Subroutine

**Purpose**

Returns the final length of an extension header.

**Syntax**

```c
int inet6_opt_finish(void *extbuf, socklen_t extlen, int offset);
```

**Description**

The `inet6_opt_finish` subroutine returns the final length of an extension header, taking into account the final padding of the extension header to make it a multiple of 8 bytes.

**Parameters**

- `extbuf`
  - If NULL, `inet6_opt_finish` will only return the final length.
  - If `extbuf` is not NULL, in addition to returning the length, the function initializes the option by inserting a `Pad1` or `PadN` option of the proper length.

- `extlen`
  - Size of the buffer pointed to by `extbuf`.

- `offset`
  - The length returned by `inet6_opt_init()` or a previous `inet6_opt_append()`.

**Return Values**

- -1 The necessary pad does not fit in the extension header buffer.

- integer value Final length of the extension header.

**Related Information**

- “inet6_opt_append Subroutine” on page 113
- “inet6_opt_find Subroutine” on page 114
- “inet6_opt_get_val Subroutine”
- “inet6_opt_init Subroutine” on page 116
- “inet6_opt_next Subroutine” on page 117
- “inet6_opt_set_val Subroutine” on page 118
- “inet6_rth_add Subroutine” on page 118
- “inet6_rth_getaddr Subroutine” on page 119
- “inet6_rth_init Subroutine” on page 119
- “inet6_rth_reverse Subroutine” on page 120
- “inet6_rth_segments Subroutine” on page 121
- “inet6_rth_space Subroutine” on page 122

---

### inet6_opt_get_val Subroutine

**Purpose**

Extracts data items of various sizes in the data portion of the option.
Syntax

```c
int inet6_opt_get_val(void *databuf, int offset, void *val,
                     socklen_t vallen);
```

Description

The `inet6_opt_get_val` subroutine extracts data items of various sizes in the data portion of the option. It is expected that each field is aligned on its natural boundaries, but the subroutine will not rely on the alignment.

Parameters

databuf  
Pointer to the data content returned by `inet6_opt_next()` or `inet6_opt_find()`.

offset  
Specifies where in the data portion of the option the value should be extracted. The first byte after the option type and length is accessed by specifying an offset of 0.

val  
Pointer to the destination for the extracted data.

vallen  
Specifies the size of the data content to be extracted.

Return Values

The `inet6_opt_get_val` subroutine returns the offset for the next field (that is, `offset + vallen`), which can be used when extracting option content with multiple fields.

Related Information

- `inet6_opt_append Subroutine` on page 113, `inet6_opt_find Subroutine` on page 114, `inet6_opt_finish Subroutine` on page 115, `inet6_opt_init Subroutine`, `inet6_opt_next Subroutine` on page 117, `inet6_opt_set_val Subroutine` on page 118, `inet6_rth_add Subroutine` on page 118, `inet6_rth_getaddr Subroutine` on page 119, `inet6_rth_init Subroutine` on page 119, `inet6_rth_reverse Subroutine` on page 120, `inet6_rth_segments Subroutine` on page 121, `inet6_rth_space Subroutine` on page 122

`inet6_opt_init` Subroutine

Purpose

Returns the number of bytes needed for an empty extension header.

Syntax

```c
int inet6_opt_init(void *extbuf, socklen_t extlen);
```

Description

The `inet6_opt_init` subroutine returns the number of bytes needed for the empty extension header (that is, a header without any options).

Parameters

extbuf  
Specifies NULL for an empty header. If `extbuf` is not NULL, it initializes the extension header to have the correct length field.

extlen  
 Specifies the size of the extension header. The value of `extlen` must be a positive value that is a multiple of 8.

Return Values

-1  
The value of `extlen` is not a positive (non-zero) multiple of 8.
integer value Number of bytes needed for an empty extension header.

## Related Information

- “inet6_opt_append Subroutine” on page 113
- “inet6_opt_find Subroutine” on page 114
- “inet6_opt_finish Subroutine” on page 115
- “inet6_opt_get_val Subroutine” on page 115
- “inet6_opt_next Subroutine” on page 115
- “inet6_opt_set_val Subroutine” on page 118
- “inet6_rth_add Subroutine” on page 118
- “inet6_rth_getaddr Subroutine” on page 119
- “inet6_rth_init Subroutine” on page 119
- “inet6_rth_reverse Subroutine” on page 120
- “inet6_rth_segments Subroutine” on page 121
- “inet6_rth_space Subroutine” on page 122

## inet6_opt_next Subroutine

### Purpose

Parses received option extension headers returning the next option.

### Syntax

```c
int inet6_opt_next(void *extbuf, socklen_t extlen, int offset,
    uint8_t *typep, socklen_t *lenp, void **databufp);
```

### Description

The `inet6_opt_next` subroutine parses received option extension headers, returning the next option. The next option is returned by updating the `typep`, `lenp`, and `databufp` parameters.

### Parameters

- **extbuf**
  - Specifies the extension header.
- **extlen**
  - Size of the buffer pointed to by `extbuf`.
- **offset**
  - Specifies the position where scanning of the extension buffer can continue. Should either be 0 (for the first option) or the length returned by a previous call to `inet6_opt_next()` or `inet6_opt_find()`.
- **typep**
  - Stores the option type.
- **lenp**
  - Stores the length of the option data (excluding the option type and option length fields).
- **databufp**
  - Points to the data field of the option.

### Return Values

The `inet6_opt_next` subroutine returns the updated "previous" length computed by advancing past the option that was returned. This returned "previous" length can then be passed to subsequent calls to `inet6_opt_next()`. This function does not return any `PAD1` or `PADN` options.

-1 There are no more options or the option extension header is malformed.

### Related Information

- “inet6_opt_append Subroutine” on page 113
- “inet6_opt_find Subroutine” on page 114
- “inet6_opt_finish Subroutine” on page 115
- “inet6_opt_get_val Subroutine” on page 115
- “inet6_opt_init Subroutine” on page 116
- “inet6_opt_next Subroutine” on page 115
- “inet6_opt_set_val Subroutine” on page 118
- “inet6_rth_add Subroutine” on page 118
- “inet6_rth_getaddr Subroutine” on page 119
- “inet6_rth_init Subroutine” on page 119
- “inet6_rth_reverse Subroutine” on page 120
- “inet6_rth_segments Subroutine” on page 121
- “inet6_rth_space Subroutine” on page 122
inet6_opt_set_val Subroutine

Purpose
Inserts data items into the data portion of an option.

Syntax
```c
int inet6_opt_set_val(void *databuf, int offset, void *val);
```

Description
The `inet6_opt_set_val` subroutine inserts data items of various sizes into the data portion of the option. The caller must ensure that each field is aligned on its natural boundaries. However, even when the alignment requirement is not satisfied, `inet6_opt_set_val` will just copy the data as required.

Parameters
- `databuf`: Pointer to the data area returned by `inet6_opt_append()`.
- `offset`: Specifies where in the data portion of the option the value should be inserted; the first byte after the option type and length is accessed by specifying an offset of 0.
- `val`: Pointer to the data content to be inserted.
- `vallen`: Specifies the size of the data content to be inserted.

Return Values
The function returns the offset for the next field (that is, `offset + vallen`), which can be used when composing option content with multiple fields.

Related Information
- "inet6_opt_append Subroutine" on page 113
- "inet6_opt_find Subroutine" on page 114
- "inet6_opt_finish Subroutine" on page 115
- "inet6_opt_get_val Subroutine" on page 115
- "inet6_opt_init Subroutine" on page 116
- "inet6_opt_next Subroutine" on page 117
- "inet6_rth_add Subroutine" on page 119
- "inet6_rth_getaddr Subroutine" on page 119
- "inet6_rth_init Subroutine" on page 119
- "inet6_rth_reverse Subroutine" on page 120
- "inet6_rth_segments Subroutine" on page 121
- "inet6_rth_space Subroutine" on page 122

inet6_rth_add Subroutine

Purpose
Adds an IPv6 address to the end of the Routing header being constructed.

Syntax
```c
int inet6_rth_add(void *bp, const struct in6_addr *addr);
```

Description
The `inet6_rth_add` subroutine adds the IPv6 address pointed to by `addr` to the end of the Routing header being constructed.

Parameters
- `bp`: Points to the buffer of the Routing header.
- `addr`: Specifies which IPv6 address is to be added.
Return Values

0  Success. The segleft member of the Routing Header is updated to account for the new address in the Routing header.
–1  The new address could not be added.

Related Information


inet6_rth_getaddr Subroutine

Purpose
Returns a pointer to a specific IPv6 address in a Routing header.

Syntax

```c
struct in6_addr *inet6_rth_getaddr(const void *bp, int index);
```

Description
The inet6_rth_getaddr subroutine returns a pointer to the IPv6 address specified by index in the Routing header described by bp. An application should first call inet6_rth_segments() to obtain the number of segments in the Routing header.

Parameters

bp  Points to the Routing header.
index  Specifies the index of the IPv6 address that must be returned. The value of index must be between 0 and one less than the value returned by inet6_rth_segments().

Return Values

NULL  The inet6_rth_getaddr subroutine failed.
Valid pointer  Pointer to the address indexed by index.

Related Information


inet6_rth_init Subroutine

Purpose
Initializes a buffer to contain a Routing header.
Syntax

```c
void *inet6_rth_init(void *bp, socklen_t bp_len, int type, int segments);
```

Description

The `inet6_rth_init` subroutine initializes the buffer pointed to by `bp` to contain a Routing header of the specified `type` and sets `ip6r_len` based on the `segments` parameter. `bp_len` is only used to verify that the buffer is large enough. The `ip6r_segleft` field is set to 0; `inet6_rth_add()` increments it.

When the application uses ancillary data, the application must initialize any `cmsghdr` fields. The caller must allocate the buffer, and the size of the buffer can be determined by calling `inet6_rth_space()`.

Parameters

- `bp` Points to the buffer to be initialized.
- `bp_len` Size of the buffer pointed to by `bp`.
- `type` Specifies the type of Routing header to be held.
- `segments` Specifies the number of addresses within the Routing header.

Return Values

Upon success, the return value is the pointer to the buffer (`bp`), and this is then used as the first argument to the `inet6_rth_add()` function.

- `NULL` The buffer could not be initialized.

Related Information

- "inet6_opt_append Subroutine" on page 113
- "inet6_opt_find Subroutine" on page 114
- "inet6_opt_finish Subroutine" on page 115
- "inet6_opt_get_val Subroutine" on page 115
- "inet6_opt_init Subroutine" on page 116
- "inet6_opt_next Subroutine" on page 117
- "inet6_opt_set_val Subroutine" on page 118
- "inet6_rth_add Subroutine" on page 118
- "inet6_rth_getaddr Subroutine" on page 119
- "inet6_rth_reverse Subroutine" on page 119
- "inet6_rth_segments Subroutine" on page 121
- "inet6_rth_space Subroutine" on page 122

**inet6_rth_reverse Subroutine**

Purpose

Writes a new Routing header that sends datagrams along the reverse route of a Routing header extension header.

Syntax

```c
int inet6_rth_reverse(const void *in, void *out);
```

Description

The `inet6_rth_reverse` subroutine takes a Routing header extension header (pointed to by the first argument) and writes a new Routing header that sends datagrams along the reverse of that route. The function reverses the order of the addresses and sets the `segleft` member in the new Routing header to the number of segments. Both arguments are allowed to point to the same buffer (that is, the reversal can occur in place).
Parameters

in  Points to the original Routing header extension header.
out Points to the new Routing header route that reverses the route of in.

Return Values

0   The reverse Routing header was successfully created.
–1  The reverse Routing header could not be created.

Related Information

"inet6_opt_append Subroutine" on page 113, "inet6_opt_find Subroutine" on page 114, "inet6_opt_finish Subroutine" on page 115, "inet6_opt_get Subroutine" on page 115, "inet6_opt_init Subroutine" on page 116, "inet6_opt_next Subroutine" on page 117, "inet6_opt_set Subroutine" on page 118, "inet6_rth_add Subroutine" on page 119, "inet6_rth_getaddr Subroutine" on page 119, "inet6_rth_init Subroutine" on page 119, "inet6_rth_reverse Subroutine" on page 120, "inet6_rth_space Subroutine" on page 122

inet6_rth_segments Subroutine

Purpose

Returns the number of segments (addresses) contained in a Routing header.

Syntax

int inet6_rth_segments(const void *bp);

Description

The inet6_rth_segments subroutine returns the number of segments (addresses) contained in the Routing header described by bp.

Parameters

bp  Points to the Routing header.

Return Values

0 (or greater)   The number of addresses in the Routing header was returned.
–1               The number of addresses of the Routing header could not be returned.

Related Information

"inet6_opt_append Subroutine" on page 113, "inet6_opt_find Subroutine" on page 114, "inet6_opt_finish Subroutine" on page 115, "inet6_opt_get Subroutine" on page 115, "inet6_opt_init Subroutine" on page 116, "inet6_opt_next Subroutine" on page 117, "inet6_opt_set Subroutine" on page 118, "inet6_rth_add Subroutine" on page 119, "inet6_rth_getaddr Subroutine" on page 119, "inet6_rth_init Subroutine" on page 119, "inet6_rth_reverse Subroutine" on page 120, "inet6_rth_space Subroutine" on page 122
inet6_rth_space Subroutine

Purpose
Returns the required number of bytes to hold a Routing header.

Syntax
socklen_t inet6_rth_space(int type, int segments);

Description
The inet6_rth_space subroutine returns the number of bytes required to hold a Routing header of the specified type containing the specified number of segments (addresses). For an IPv6 Type 0 Routing header, the number of segments must be between 0 and 127, inclusive. The return value is simply the space for the Routing header. When the application uses ancillary data, the application must pass the returned length to CMSG_SPACE() in order to determine how much memory is needed for the ancillary data object (including the cmsghdr structure).

Note: Although inet6_rth_space returns the size of the ancillary data, it does not allocate the space required for the ancillary data. This allows an application to allocate a larger buffer, so that other ancillary data objects can be added, because all the ancillary data objects must be specified to sendmsg() as a single msg_control buffer.

Parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>type</td>
<td>Specifies the type of Routing header to be held.</td>
</tr>
<tr>
<td>segments</td>
<td>Specifies the number of addresses within the Routing header.</td>
</tr>
</tbody>
</table>

Return Values

<table>
<thead>
<tr>
<th>Return Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Either the type of the Routing header is not supported by this implementation or the number of segments is invalid for this type of Routing header.</td>
</tr>
<tr>
<td>length</td>
<td>Determines how much memory is needed for the ancillary data object.</td>
</tr>
</tbody>
</table>

Related Information

inet6_opt_append Subroutine on page 113, inet6_opt_find Subroutine on page 114, inet6_opt_finish Subroutine on page 115, inet6_opt_get_val Subroutine on page 115, inet6_opt_init Subroutine on page 116, inet6_opt_next Subroutine on page 117, inet6_opt_set_val Subroutine on page 118, inet6_rth_add Subroutine on page 118, inet6_rth_getaddr Subroutine on page 119, inet6_rth_init Subroutine on page 119, inet6_rth_reverse Subroutine on page 120, inet6_rth_segments Subroutine on page 121

inet_addr Subroutine

Purpose
Converts Internet addresses to Internet numbers.

Library
Standard C Library (libc.a)


```c
#include <sys/socket.h>
#include <sys/socketvar.h>
#include <netinet/in.h>
#include <arpa/inet.h>

in_addr_t inet_addr (const char *CharString);
```

### Description

The **inet_addr** subroutine converts an ASCII string containing a valid Internet address using dot notation into an Internet address number typed as an unsigned integer value. An example of dot notation is 120.121.5.123. The **inet_addr** subroutine returns an error value if the Internet address notation in the ASCII string supplied by the application is not valid.

**Note:** Although they both convert Internet addresses in dot notation to Internet numbers, the **inet_addr** subroutine and **inet_network** process ASCII strings differently. When an application gives the **inet_addr** subroutine a string containing an Internet address value without a delimiter, the subroutine returns the logical product of the value represented by the string and 0xFFFFFFFF. For any other Internet address, if the value of the fields exceeds the previously defined limits, the **inet_addr** subroutine returns an error value of -1.

When an application gives the **inet_network** subroutine a string containing an Internet address value without a delimiter, the **inet_network** subroutine returns the logical product of the value represented by the string and 0xFF. For any other Internet address, the subroutine returns an error value of -1 if the value of the fields exceeds the previously defined limits.

All applications containing the **inet_addr** subroutine must be compiled with the **_BSD** macro set to a specific value. Acceptable values are 43 and 44. In addition, all socket applications must include the BSD **libbsd.a** library.

Sample return values for each subroutine are as follows:

<table>
<thead>
<tr>
<th>Application String</th>
<th>inet_addr Returns</th>
<th>inet_network Returns</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x1234567890abcdef</td>
<td>0x090abcdef 0xFFFFFFFF (= -1)</td>
<td>0x000000ef 0x0000ef00 0x00010203</td>
</tr>
</tbody>
</table>

The ASCII string for the **inet_addr** subroutine must conform to the following format:

```
string::= field | field delimited_field^1-3 | delimited_field^1-3
delimited_field::= delimiter field | delimiter
delimiter::=. 
field::= 0X | 0x | 0Xhexadecimal* | 0x hexadecimal* | decimal* | 0 octal
hexadecimal::= decimal [a|b|c|d|e|f|A|B|C|D|E|F
decimal::= octal | 8 | 9
octal::= 0|1|2|3|4|5|6|7
```

**Notes:**
1. `^n` indicates `n` repetitions of a pattern.
2. `^n-m` indicates `n` to `m` repetitions of a pattern.
3. `*` indicates 0 or more repetitions of a pattern, up to environmental limits.
4. The Backus Naur form (BNF) description states the space character, if one is used. Text indicates text, not a BNF symbol.
The **inet_addr** subroutine requires an application to terminate the string with a null terminator (0x00) or a space (0x30). The string is considered invalid if the application does not end it with a null terminator or a space. The subroutine ignores characters trailing a space.

The following describes the restrictions on the field values for the **inet_addr** subroutine:

<table>
<thead>
<tr>
<th>Format</th>
<th>Field Restrictions (in decimal)</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>Value _a &lt; 4,294,967,296</td>
</tr>
<tr>
<td>a.b</td>
<td>Value _a &lt; 256; Value _b &lt; 16,777,216</td>
</tr>
<tr>
<td>a.b.c</td>
<td>Value _a &lt; 256; Value _b &lt; 256; Value _c &lt; 65536</td>
</tr>
<tr>
<td>a.b.c.d</td>
<td>Value _a &lt; 256; Value _b &lt; 256; Value _c &lt; 256; Value _d &lt; 256</td>
</tr>
</tbody>
</table>

Applications that use the **inet_addr** subroutine can enter field values exceeding these restrictions. The subroutine accepts the least significant bits up to an integer in length, then checks whether the truncated value exceeds the maximum field value. For example, if an application enters a field value of 0x1234567890 and the system uses 16 bits per integer, then the **inet_addr** subroutine uses bits 0 -15. The subroutine returns 0x34567890.

Applications can omit field values between delimiters. The **inet_addr** subroutine interprets empty fields as 0.

**Notes:**
1. The **inet_addr** subroutine does not check the pointer to the ASCII string. The user must ensure the validity of the address in the ASCII string.
2. The application must verify that the network and host IDs for the Internet address conform to either a Class A, B, or C Internet address. The **inet_attr** subroutine processes any other number as a Class C address.

**Parameters**

*CharString* Represents a string of characters in the Internet address form.

**Return Values**

For valid input strings, the **inet_addr** subroutine returns an unsigned integer value comprised of the bit patterns of the input fields concatenated together. The subroutine places the first pattern in the most significant position and appends any subsequent patterns to the next most significant positions.

The **inet_addr** subroutine returns an error value of -1 for invalid strings.

**Note:** An Internet address with a dot notation value of 255.255.255.255 or its equivalent in a different base format causes the **inet_addr** subroutine to return an unsigned integer value of 4294967295. This value is identical to the unsigned representation of the error value. Otherwise, the **inet_addr** subroutine considers 255.255.255.255 a valid Internet address.

**Files**

* /etc/hosts
  Contains host names.

* /etc/networks
  Contains network names.
Related Information

The **endhostent** subroutine, **endnetent** subroutine, **gethostbyaddr** subroutine, **gethostbyname** subroutine, **gethostbyaddr** subroutine, **getnetbyname** subroutine, **getnetent** subroutine, **inet_lnaof** subroutine, **inet_makeaddr** subroutine, **inet_netof** subroutine, **inet_network** subroutine, **inet_ntoa** subroutine, **sethostent** subroutine, **setnetent** subroutine.

[Sockets Overview](#) and [Understanding Network Address Translation](#) in *AIX 5L Version 5.3 Communications Programming Concepts.*

---

**inet_lnaof Subroutine**

**Purpose**

Returns the host ID of an Internet address.

**Library**

Standard C Library (libc.a)

**Syntax**

```c
#include <sys/socket.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>

int inet_lnaof (InternetAddr)
struct in_addr InternetAddr;
```

**Description**

The **inet_lnaof** subroutine masks off the host ID of an Internet address based on the Internet address class. The calling application must enter the Internet address as an unsigned long value.

All applications containing the **inet_lnaof** subroutine must be compiled with the _BSD macro set to a specific value. Acceptable values are 43 and 44. In addition, all socket applications must include the BSD libbsd.a library.

**Note:** The application must verify that the network and host IDs for the Internet address conform to either a Class A, B, or C Internet address. The **inet_lnaof** subroutine processes any other number as a Class C address.

**Parameters**

*InternetAddr* Specifies the Internet address to separate.

**Return Values**

The return values of the **inet_lnaof** subroutine depend on the class of Internet address the application provides:

<table>
<thead>
<tr>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Class A</td>
<td>The logical product of the Internet address and 0xffffffff.</td>
</tr>
<tr>
<td>Class B</td>
<td>The logical product of the Internet address and 0x0000ffff.</td>
</tr>
<tr>
<td>Class C</td>
<td>The logical product of the Internet address and 0x000000ff.</td>
</tr>
</tbody>
</table>
Files

/etc/hosts

Contains host names.

Related Information

The endhostent subroutine, endnetent subroutine, gethostbyaddr subroutine, gethostbyname subroutine, getnetbyaddr subroutine, getnetbyname subroutine, getnetent subroutine, inet_makeaddr subroutine, inet_netof subroutine, inet_network subroutine, inet_ntoa subroutine, sethostent subroutine, setnetent subroutine.

Sockets Overview and Understanding Network Address Translation in AIX 5L Version 5.3 Communications Programming Concepts.

inet_makeaddr Subroutine

Purpose

Returns a structure containing an Internet address based on a network ID and host ID provided by the application.

Library

Standard C Library (libc.a)

Syntax

```c
#include <sys/socket.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>

struct in_addr inet_makeaddr (Net, LocalNetAddr)
int Net, LocalNetAddr;
```

Description

The `inet_makeaddr` subroutine forms an Internet address from the network ID and Host ID provided by the application (as integer types). If the application provides a Class A network ID, the `inet_makeaddr` subroutine forms the Internet address using the net ID in the highest-order byte and the logical product of the host ID and 0x00FFFFFF in the 3 lowest-order bytes. If the application provides a Class B network ID, the `inet_makeaddr` subroutine forms the Internet address using the net ID in the two highest-order bytes and the logical product of the host ID and 0x0000FFFF in the lowest two ordered bytes. If the application does not provide either a Class A or Class B network ID, the `inet_makeaddr` subroutine forms the Internet address using the network ID in the 3 highest-order bytes and the logical product of the host ID and 0x0000FFFF in the lowest-ordered byte.

The `inet_makeaddr` subroutine ensures that the Internet address format conforms to network order, with the first byte representing the high-order byte. The `inet_makeaddr` subroutine stores the Internet address in the structure as an unsigned long value.

The application must verify that the network ID and host ID for the Internet address conform to class A, B, or C. The `inet_makeaddr` subroutine processes any nonconforming number as a Class C address.
The **inet_makeaddr** subroutine expects the **in_addr** structure to contain only the Internet address field. If the application defines the **in_addr** structure otherwise, then the value returned in **in_addr** by the **inet_makeaddr** subroutine is undefined.

All applications containing the **inet_makeaddr** subroutine must be compiled with the _BSD macro set to a specific value. Acceptable values are 43 and 44. In addition, all socket applications must include the BSD libbsd.a library.

**Parameters**

- **Net**
  - Contains an Internet network number.
- **LocalNetAddr**
  - Contains a local network address.

**Return Values**

Upon successful completion, the **inet_makeaddr** subroutine returns a structure containing an Internet address.

If the **inet_makeaddr** subroutine is unsuccessful, the subroutine returns a -1.

**Files**

- **/etc/hosts**
  - Contains host names.

**Related Information**

The **endhostent** subroutine, **endnetent** subroutine, **gethostbyaddr** subroutine, **gethostbyname** subroutine, **net_addr** subroutine, **in_nhaof** subroutine, **net_netof** subroutine, **net_network** subroutine, **net_nfoa** subroutine, **sethostent** subroutine, **setnetent** subroutine.

[Detailed information on related subroutines](AIX 5L Version 5.3 Communications Programming Concepts).

---

**inet_net_ntop Subroutine**

**Purpose**

Converts between binary and text address formats.

**Library**

Library (libc.a)

**Syntax**

```c
char *inet_net_ntop (af, src, bits, dst, size);
```

```c
int af;
const void *src;
int bits;
char *dst;
sizet size;
```
Description
This function converts a network address and the number of bits in the network part of the address into the CIDR format ascii text (for example, 9.3.149.0/24). The \texttt{af} parameter specifies the family of the address. The \texttt{src} parameter points to a buffer holding an IPv4 address if the \texttt{af} parameter is \texttt{AF_INET}. The \texttt{bits} parameter is the size (in bits) of the buffer pointed to by the \texttt{src} parameter. The \texttt{dst} parameter points to a buffer where the function stores the resulting text string. The \texttt{size} parameter is the size (in bytes) of the buffer pointed to by the \texttt{dst} parameter.

Parameters

\begin{description}
\item \texttt{af} \hspace{1cm} Specifies the family of the address. \\
\item \texttt{src} \hspace{1cm} Points to a buffer holding and IPv4 address if the \texttt{af} parameter is \texttt{AF_INET}. \\
\item \texttt{bits} \hspace{1cm} Specifies the size of the buffer pointed to by the \texttt{src} parameter. \\
\item \texttt{dst} \hspace{1cm} Points to a buffer where the resulting text string is stored. \\
\item \texttt{size} \hspace{1cm} Specifies the size of the buffer pointed to by the \texttt{dst} parameter.
\end{description}

Return Values
If successful, a pointer to a buffer containing the text string is returned. If unsuccessful, NULL is returned. Upon failure, \texttt{errno} is set to EAFNOSUPPORT if the \texttt{af} parameter is invalid or ENOSPC if the size of the result buffer is inadequate.

Related Information
"inet_net_pton Subroutine," "inet_ntop Subroutine" on page 133, and "inet_pton Subroutine" on page 134.

\textbf{Subroutines Overview} in \textit{AIX 5L Version 5.3 General Programming Concepts: Writing and Debugging Programs}.

\textbf{inet_net_pton Subroutine}

Purpose
Converts between text and binary address formats.

Library
Library (\texttt{libc.a})

Syntax
\begin{verbatim}
int inet_net_pton (af, src, dst, size);
\end{verbatim}

Description
This function converts a network address in ascii into the binary network address. The ascii representation can be CIDR-based (for example, 9.3.149.0/24) or class-based (for example, 9.3.149.0). The \texttt{af} parameter specifies the family of the address. The \texttt{src} parameter points to the string being passed in. The \texttt{dst} parameter points to a buffer where the function will store the resulting numeric address. The \texttt{size} parameter is the size (in bytes) of the buffer pointed to by the \texttt{dst} parameter.
Parameters

*af* Specifies the family of the address.
*src* Points to the string being passed in.
*dst* Points to a buffer where the resulting numeric address is stored.
*size* Specifies the size (in bytes) of the buffer pointed to by the *dst* parameter.

Return Values
If successful, the number of bits, either inputted classfully or specified with /CIDR, is returned. If unsuccessful, a -1 (negative one) is returned (check errno). ENOENT means it was not a valid network specification.

Related Information
"inet_net_ntop Subroutine" on page 127, "inet_ntop Subroutine" on page 133, and "inet_pton Subroutine" on page 134.
"Subroutines Overview" in AIX 5L Version 5.3 General Programming Concepts: Writing and Debugging Programs.

inet_netof Subroutine

Purpose
Returns the network id of the given Internet address.

Library
Standard C Library (*libc.a*)

Syntax
```c
#include <sys/socket.h>
#include <sys/socketvar.h>
#include <netinet/in.h>
#include <arpa/inet.h>

int inet_netof (InternetAddr)
struct in_addr InternetAddr;
```

Description
The *inet_netof* subroutine returns the network number from the specified Internet address number typed as unsigned long value. The *inet_netof* subroutine masks off the network number and the host number from the Internet address based on the Internet address class.

All applications containing the *inet_netof* subroutine must be compiled with the _BSD macro set to a specific value. Acceptable values are 43 and 44. In addition, all socket applications must include the BSD *libbsd.a* library.

Note: The application assumes responsibility for verifying that the network number and the host number for the Internet address conforms to a class A or B or C Internet address. The *inet_netof* subroutine processes any other number as a class C address.
Parameters

InternetAddr

Specifies the Internet address to separate.

Return Values

Upon successful completion, the `inet_netof` subroutine returns a network number from the specified long value representing the Internet address. If the application gives a class A Internet address, the `inet_lnoaf` subroutine returns the logical product of the Internet address and `0xFF000000`. If the application gives a class B Internet address, the `inet_lnoaf` subroutine returns the logical product of the Internet address and `0xFFFF0000`. If the application does not give a class A or B Internet address, the `inet_lnoaf` subroutine returns the logical product of the Internet address and `0xFFFFFFFF`.

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_makeaddr` subroutine, `inet_network` subroutine, `inet_ntoa` subroutine, `sethostent` subroutine, `setnetent` subroutine.

Sockets Overview and Understanding Network Address Translation in AIX 5L Version 5.3 Communications Programming Concepts.

inet_network Subroutine

Purpose

Converts an ASCII string containing an Internet network addressee in . (dot) notation to an Internet address number.

Library

Standard C Library (libc.a)

Syntax

```c
#include <sys/socket.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>

in_addr_t inet_network (CharString)
register const char *CharString;
```

Description

The `inet_network` subroutine converts an ASCII string containing a valid Internet address using . (dot) notation (such as, 120.121.122.123) to an Internet address number formatted as an unsigned integer.
The **inet_network** subroutine returns an error value if the application does not provide an ASCII string containing a valid Internet address using . notation.

The input ASCII string must represent a valid Internet address number, as described in [TCP/IP addressing](#) in *Networks and communication management*. The input string must be terminated with a null terminator (0x00) or a space (0x30). The **inet_network** subroutine ignores characters that follow the terminating character.

The input string can express an Internet address number in decimal, hexadecimal, or octal format. In hexadecimal format, the string must begin with 0x. The string must begin with 0 to indicate octal format. In decimal format, the string requires no prefix.

Each octet of the input string must be delimited from another by a period. The application can omit values between delimiters. The **inet_network** subroutine interprets missing values as 0.

The following examples show valid strings and their output values in both decimal and hexadecimal notation:

**Examples of valid strings**

<table>
<thead>
<tr>
<th>Input String</th>
<th>Output Value (in decimal)</th>
<th>Output Value (in hex)</th>
</tr>
</thead>
<tbody>
<tr>
<td>...1</td>
<td>1</td>
<td>0x00000001</td>
</tr>
<tr>
<td>.1..</td>
<td>65536</td>
<td>0x00010000</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0x1</td>
</tr>
<tr>
<td>0xFFFFFFFF</td>
<td>255</td>
<td>0x000000FF</td>
</tr>
<tr>
<td>1.</td>
<td>256</td>
<td>0x100</td>
</tr>
<tr>
<td>1.2.3.4</td>
<td>66048</td>
<td>0x010200</td>
</tr>
<tr>
<td>0x01.0X2.03.004</td>
<td>16909060</td>
<td>0x01020304</td>
</tr>
<tr>
<td>1.2. 3.4</td>
<td>16777218</td>
<td>0x01000002</td>
</tr>
<tr>
<td>9999.1.1.1</td>
<td>251724033</td>
<td>0x0F010101</td>
</tr>
</tbody>
</table>

The following examples show invalid input strings and the reasons they are not valid:

**Examples of invalid strings**

<table>
<thead>
<tr>
<th>Input String</th>
<th>Reason</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.2.3.4.5</td>
<td>Excessive fields.</td>
</tr>
<tr>
<td>1.2.3.4.</td>
<td>Excessive delimiters (and therefore fields).</td>
</tr>
<tr>
<td>1.2.</td>
<td>Bad delimiter.</td>
</tr>
<tr>
<td>1p</td>
<td>String not terminated by null terminator nor space.</td>
</tr>
<tr>
<td>(empty string)</td>
<td>No field or delimiter present.</td>
</tr>
</tbody>
</table>

Typically, the value of each octet of an Internet address cannot exceed 246. The **inet_network** subroutine can accept larger values, but it uses only the eight least significant bits for each field value. For example, if an application passes 0x1234567890.0abcdef, the **inet_network** subroutine returns 37103 (0x000090EF).

The application must verify that the network ID and host ID for the Internet address conform to class A, class B, or class C. The **inet_makeaddr** subroutine processes any nonconforming number as a class C address.

The **inet_network** subroutine does not check the pointer to the ASCII input string. The application must verify the validity of the address of the string.
All applications containing the **inet_network** subroutine must be compiled with the **_BSD** macro 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* Represents a string of characters in the Internet address form.

**Return Values**

For valid input strings, the **inet_network** subroutine returns an unsigned integer value that comprises the bit patterns of the input fields concatenated together. The **inet_network** subroutine places the first pattern in the leftmost (most significant) position and appends subsequent patterns if they exist.

For invalid input strings, the **inet_network** subroutine returns a value of -1.

**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_ifaddrs** subroutine, **inet_makeaddr** subroutine, **inet_netof** subroutine, **inet_ntoa** subroutine.

[Sockets Overview] and [Understanding Network Address Translation] in *AIX 5L Version 5.3 Communications Programming Concepts*.

**inet_ntoa Subroutine**

**Purpose**

Converts an Internet address into an ASCII string.

**Library**

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

**Syntax**

```c
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>

char *inet_ntoa (struct in_addr InternetAddr);
```

**Description**

The **inet_ntoa** subroutine takes an Internet address and returns an ASCII string representing the Internet address in dot notation. All Internet addresses are returned in network order, with the first byte being the high-order byte.
Use C language integers when specifying each part of a dot notation.

All applications containing the `inet_ntoa` subroutine must be compiled with the `_BSD` macro set to a specific value. Acceptable values are 43 and 44. In addition, all socket applications must include the BSD `libbsd.a` library.

**Parameters**

- **InternetAddr**: Contains the Internet address to be converted to ASCII.

**Return Values**

Upon successful completion, the `inet_ntoa` subroutine returns an Internet address.

If the `inet_ntoa` subroutine is unsuccessful, the subroutine returns a -1.

**Files**

- `/etc/hosts`: Contains host names.
- `/etc/networks`: Contains network names.

**Related Information**

The `endhostent` subroutine, `endnetent` subroutine, `gethostbyaddr` subroutine, `gethostbyname` subroutine, `inet_addr` subroutine, `inet_lnaof` subroutine, `inet_makeaddr` subroutine, `inet_network` subroutine, `sethostent` subroutine, `setnetent` subroutine.

[Sockets Overview and Understanding Network Address Translation](#) in AIX 5L Version 5.3 Communications Programming Concepts.

**inet_ntop Subroutine**

**Purpose**

Converts a binary address into a text string suitable for presentation.

**Library**

Library (`libc.a`)

**Syntax**

```c
const char *inet_ntop (af, src, dst, size)
int af;
const void *src;
char *dst;
size_t size;
```

**Description**

This function converts from an address in binary format (as specified by the `src` parameter) to standard text format, and places the result in the `dst` parameter (if `size`, which specifies the space available in the `dst` parameter, is sufficient). The `af` parameter specifies the family of the address. This can be `AF_INET` or `AF_INET6`. 
The src parameter points to a buffer holding an IPv4 address if the af parameter is AF_INET, or an IPv6 address if the af parameter is AF_INET6. The dst parameter points to a buffer where the function will store the resulting text string. The size parameter specifies the size of this buffer (in bytes). The application must specify a non-NULL dst parameter. For IPv6 addresses, the buffer must be at least INET6_ADDRSTRLEN bytes. For IPv4 addresses, the buffer must be at least INET_ADDRSTRLEN bytes.

In order to allow applications to easily declare buffers of the proper size to store IPv4 and IPv6 addresses in string form, the following two constants are defined in the `<netinet/in.h>` library:

```c
#define INET_ADDRSTRLEN 16
#define INET6_ADDRSTRLEN 46
```

**Parameters**

- **af**: Specifies the family of the address. This can be AF_INET or AF_INET6.
- **src**: Points to a buffer holding an IPv4 address if the af parameter is set to AF_INET, or an IPv6 address if the af parameter is set to AF_INET6.
- **dst**: Points to a buffer where the resulting text string is stored.
- **size**: Specifies the size (in bytes) of the buffer pointed to by the dst parameter.

**Return Values**

If successful, a pointer to the buffer containing the converted address is returned. If unsuccessful, NULL is returned. Upon failure, the errno global variable is set to EAFNOSUPPORT if the specified address family (af) is unsupported, or to ENOSPC if the size parameter indicates the destination buffer is too small.

**Related Information**

"inet_net_ntop Subroutine" on page 127, "inet_net_pton Subroutine" on page 128, and "inet_pton Subroutine." in AIX 5L Version 5.3 General Programming Concepts: Writing and Debugging Programs.

**inet_pton Subroutine**

**Purpose**

Converts an address in its standard text form into its numeric binary form.

**Library**

Library (libc.a)

**Syntax**

```c
int inet_pton (af, src, dst)
```

**Description**

This function converts an address in its standard text format into its numeric binary form. The af parameter specifies the family of the address.

**Note**: Only the AF_INET and AF_INET6 address families are supported.
The src parameter points to the string being passed in. The dst parameter points to a buffer where the function stores the numeric address. The address is returned in network byte order.

**Parameters**

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>af</td>
<td>Specifies the family of the address. This can be AF_INET or AF_INET6.</td>
</tr>
<tr>
<td>src</td>
<td>Points to a buffer holding an IPv4 address if the af parameter is set to AF_INET, or an IPv6 address if the af parameter is set to AF_INET6.</td>
</tr>
<tr>
<td>dst</td>
<td>Points to a buffer where the resulting text string is stored.</td>
</tr>
</tbody>
</table>

**Return Values**

If successful, one is returned. If unsuccessful, zero is returned if the input is not a valid IPv4 dotted-decimal string or a valid IPv6 address string; or a negative one with the errno global variable set to EAFNOSUPPORT if the af parameter is unknown. The calling application must ensure that the buffer referred to by the dst parameter is large enough to hold the numeric address (4 bytes for AF_INET or 16 bytes for AF_INET6).

If the af parameter is AF_INET, the function accepts a string in the standard IPv4 dotted-decimal form.

```
ddd.ddd.ddd.ddd
```

Where ddd is a one to three digit decimal number between 0 and 255.

**Note:** Many implementations of the existing inet_addr and inet_aton functions accept nonstandard input such as octal numbers, hexadecimal numbers, and fewer than four numbers. inet_pton does not accept these formats.

If the af parameter is AF_INET6, then the function accepts a string in one of the standard IPv6 text forms defined in the addressing architecture specification.

**Related Information**

"inet_net_ntop Subroutine" on page 127, "inet_net_pton Subroutine" on page 128, and "inet_ntop Subroutine" on page 133.

**innetsg, getnetgrent, setnetgrent, or endnetgrent Subroutine**

**Purpose**

Handles the group network entries.

**Library**

Standard C Library (libc.a)

**Syntax**

```c
#include <netdb.h>

innetsg (NetGroup, Machine, User, Domain)


getnetgrent (MachinePointer, UserPointer, DomainPointer)

char ** MachinePointer, ** UserPointer, ** DomainPointer;
```
void setnetgrent (NetGroup)
char *NetGroup
void endnetgrent ()

Description
The innetgr subroutine is threadsafe in AIX 4.3 and later. However, the return value points to static data that is overwritten by subsequent calls. This data must be copied to be saved for use by subsequent calls.

The innetgr subroutine returns 1 or 0, depending on if netgroup contains the machine, user, domain triple as a member. Any of these three strings; machine, user, or domain, can be NULL, in which case it signifies a wild card.

The getnetgrent subroutine returns the next member of a network group. After the call, machinepointer will contain a pointer to a string containing the name of the machine part of the network group member, and similarly for userpointer and domainpointer. If any of machinepointer, userpointer, or domainpointer is returned as a NULL pointer, it signifies a wild card. The getnetgrent subroutine uses malloc to allocate space for the name. This space is released when the endnetgrent subroutine is called. getnetgrent returns 1 if it succeeded in obtaining another member of the network group or 0 when it has reached the end of the group.

The setnetgrent subroutine establishes the network group from which the getnetgrent subroutine will obtain members, and also restarts calls to the getnetgrent subroutine from the beginning of the list. If the previous setnetgrent() call was to a different network group, an endnetgrent() call is implied. endnetgrent() frees the space allocated during the getnetgrent() calls.

Parameters

Domain Specifies the domain.
DomainPointer Points to the string containing Domain part of the network group.
Machine Specifies the machine.
MachinePointer Points to the string containing Machine part of the network group.
NetGroup Points to a network group.
User Specifies a user.
UserPointer Points to the string containing User part of the network group.

Return Values

1 Indicates that the subroutine was successful in obtaining a member.
0 Indicates that the subroutine was not successful in obtaining a member.

Files

/etc/netgroup Contains network groups recognized by the system.
/usr/include/netdb.h Contains the network database structures.

Related Information

Sockets Overview in AIX 5L Version 5.3 Communications Programming Concepts.
ioctl Socket Control Operations

Purpose
Performs network related control operations.

Syntax
```c
#include <sys/ioctl.h>

int ioctl (fd, cmd, .../* arg */)
int fd;
int cmd;
int ... /* arg */
```

Description
The socket ioctl commands perform a variety of network-related control. The `fd` argument is a socket descriptor. For non-socket descriptors, the functions performed by this call are unspecified.

The `cmd` argument and an optional third argument (with varying type) are passed to and interpreted by the socket ioctl function to perform an appropriate control operation specified by the user.

The socket ioctl control operations can be in the following control operations categories:
- **Socket**
- **Routing table**
- **ARP table**
- **Global network parameters**
- **Interface**

Parameters
- **fd** Open file descriptor that refers to a socket created using socket or accept calls.
- **cmd** Selects the control function to be performed.
- .../* arg */ Represents additional information that is needed to perform the requested function. The type of `arg` depends on the particular control request, but it is either an integer or a pointer to a socket-specific data structure.

Socket Control Operations
The following ioctl commands operate on sockets:

<table>
<thead>
<tr>
<th>ioctl command</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>SIOCATMARK</td>
<td>Determines whether the read pointer is currently pointing to the logical mark in the data stream. The logical mark indicates the point at which the out-of-band data is sent.</td>
</tr>
</tbody>
</table>

```c
ioctl(fd, SIOCATMARK,&atmark);
int atmark;
```

If `atmark` is set to 1 on return, the read pointer points to the mark and the next read returns data after the mark. If `atmark` is set to 0 on return, (assuming out-of-band data is present on the data stream), the next read returns data sent prior to the out-of-band mark.

**Note:** The out-of-band data is a logically independent data channel that is delivered to the user independently of normal data; in addition, a signal is also sent because of the immediate attention required. Ctrl-C characters are an example.
<table>
<thead>
<tr>
<th>ioctl command</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>SIOCSPGRP</td>
<td>SIOCSPGRP sets the process group information for a socket. SIOCGPGRP gets the process group ID associated with a socket.</td>
</tr>
<tr>
<td>SIOCGPGRP</td>
<td></td>
</tr>
<tr>
<td>ioctl (fd, cmd, (int)&amp;pgrp);</td>
<td>cmd</td>
</tr>
<tr>
<td>pgrp</td>
<td>Specifies the process group ID for the socket.</td>
</tr>
</tbody>
</table>

**Routing Table Control Operations**
The following ioctl commands operate on the kernel routing table:

<table>
<thead>
<tr>
<th>ioctl command</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>SIOCADDRT</td>
<td>SIOCADDRT adds a route entry in the routing table. SIOCDELRT deletes a route entry from the routing table.</td>
</tr>
<tr>
<td>SIOCDELRT</td>
<td>ioctl(fd, cmd, (caddr_t)&amp;route); struct ortentry route; cmd</td>
</tr>
<tr>
<td>SIOUPDROUTE</td>
<td>Updates the routing table using the information passed in the ifreq structure.</td>
</tr>
<tr>
<td>SIOUPDROUTE</td>
<td>ioctl (fd, SIOUPDROUTE, (caddr_t)&amp;ifr); struct ifreq ifr;</td>
</tr>
<tr>
<td>Note:</td>
<td>SIOUPDROUTE is available beginning with AIX 5100-002 (maintenance level 2) and later.</td>
</tr>
</tbody>
</table>

**ARP Table Control Operations**
The following ioctl commands operate on the kernel ARP table. The net/if_arp.h header file must be included.

<table>
<thead>
<tr>
<th>ioctl command</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>SIOCSARP</td>
<td>SIOCSARP adds or modifies an ARP entry in the ARP table. SIOCARP deletes an ARP entry from the ARP table.</td>
</tr>
<tr>
<td>SIOCARP</td>
<td>ioctl(fd, cmd, (caddr_t)&amp;or); struct arpreq or;</td>
</tr>
<tr>
<td>cmd</td>
<td>Set to SIOCSARP, SIOCARP, or SIOCGARP.</td>
</tr>
<tr>
<td>SIOCGARP</td>
<td>The ARP entry information is passed in the arpreq structure.</td>
</tr>
</tbody>
</table>
### Global Network Parameters Control Operations

The following ioctl commands operate as global network parameters:

<table>
<thead>
<tr>
<th>ioctl command</th>
<th>Description</th>
</tr>
</thead>
</table>
| SIOCSNETOPT      | SIOCSNETOPT sets the value of a network option. SIOCGNETOPT gets the value of a network option. SIOCDNETOPT sets the default values of a network option. ioctl(fd, cmd, (caddr_t)&oreq);  
  struct optreq oreq;  
  cmd Set to SIOCSNETOPT, SIOCGNETOPT, or SIOCDNETOPT. The network option value is stored in the optreq structure. SIOCGNETOPT1 gets the current value, default value, and the range of a network option. ioctl(fd, SIOCGNETOPT1, (caddr_t)&oreq);  
  struct optreq1 oreq1;  
  The network option information is stored in the optreq1 structure upon return The optreq and optreq1 structures are defined in net/netopt.h. Note: SIOCGNETOPT1 is available beginning with AIX 5.2 and later. |
| SIOCGETMTUS      | SIOCGETMTUS gets the number of MTUs maintained in the list of common MTUs. SIOCADDMTU adds an MTU in the list of common MTUs. SIOCDELMTU deletes an MTU from the list of common MTUs. ioctl(fd, cmd, (caddr_t)&nmtus);  
  int nmtus;  
  cmd Set to SIOCGNMTUS, SIOCADDMTU, or SIOCDELMTU. SIOCGGETMTUS gets the MTUs maintained in the list of common MTUs. ioctl(fd, SIOCGETMTUS, (caddr_t)&gm);  
  struct get_mtus gm;  
  The get_mtus structure is defined innetinet/in.h. |

### Interface Control Operations

The following ioctl commands operate on interfaces. The net/if.h header file must be included.

<table>
<thead>
<tr>
<th>ioctl command</th>
<th>Description</th>
</tr>
</thead>
</table>
| SIOCSIFADDR      | SIOCSIFADDR sets an interface address. SIOCAIFADDR adds or changes an interface address. SIOCDIFADDR deletes an interface address. The interface address is specified in the ifr.ifr_addr field. SIOCGIFADDR gets an interface address. The address is returned in the ifr.ifr_addr field. ioctl(fd, cmd, (caddr_t)&ifr, sizeof(struct ifreq));  
  struct ifreq ifr;  
  cmd Set to SIOCSIFADDR, SIOCAIFADDR, SIOCDIFADDR, or SIOCGIFADDR. |
| SIOCGIFADDRS     | Gets the list of addresses associated with an interface. ioctl (fd, SIOCGIFADDRS, (caddr_t)ifaddrsp);  
  struct ifreqaddr* ifaddrsp;  
  The interface name is passed in the ifaddrsp->ifr_name field. The addresses associated with the interface are stored in ifaddrsp->ifrasu[] array on return. Note: The ifreqaddr structure contains space for storing only one sockaddr_in/sockaddr_in6 structure (array of one sockaddr_in/ sockaddr_in6 element). In order to get n addresses associated with an interface, the caller of the ioctl command must allocate space for (sizeof (struct ifreqaddr) + (n * sizeof (struct sockaddr_in))) bytes. Note: SIOCGIFADDRS is available beginning with AIX 5.3 and later. |
<table>
<thead>
<tr>
<th>ioctl command</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>SIOCSIFDSTADDR</td>
<td>Sets the point-to-point address for an interface specified in the <code>ifr.ifr_dstaddr</code> field. SIOCGIFDSTADDR gets the point-to-point address associated with an interface. The address is stored in the <code>ifr.ifr_dstaddr</code> field on return.</td>
</tr>
<tr>
<td>cmd</td>
<td>Set to SIOCSIFDSTADDR or SIOCGIFDSTADDR.</td>
</tr>
<tr>
<td>ioctl(fd, cmd, (caddr_t)&amp;ifr, sizeof(struct ifreq));</td>
<td>struct ifreq ifr;</td>
</tr>
</tbody>
</table>

| SIOCSIFNETMASK      | Sets the interface netmask specified in the `ifr.ifr_addr` field. SIOCGIFNETMASK gets the interface netmask.                                                                            |
| cmd                 | Set to SIOCSIFNETMASK or SIOCGIFNETMASK.                                                                                                                                         |
| ioctl(fd, cmd, (caddr_t)&ifr, sizeof(struct ifreq)); | struct ifreq ifr;                                                                                                                                                                      |

| SIOCSIFBRDADDR      | Sets the interface broadcast address specified in the `ifr.ifr_broadaddr` field. SIOCGIFBRDADDR gets the interface broadcast address. The broadcast address is placed in the `ifr.ifr_broadaddr` field. |
| cmd                 | Set to SIOCSIFBRDADDR or SIOCGIFBRDADDR.                                                                                                                                         |
| ioctl(fd, cmd, (caddr_t)&ifr, sizeof(struct ifreq)); | struct ifreq ifr;                                                                                                                                                                      |

| SIOCGBSIZIFCONF     | Gets the size of memory required to get configuration information for all interfaces returned by SIOCGIFCONF.                                                                    |
| cmd                 |                                                                                                                                                                                  |
| ioctl(fd, cmd, (caddr_t)&ifconfsize); | int ifconfsize;                                                                                                                                                                    |

| SIOCGIFCONF         | Returns configuration information for all the interfaces configured on the system. The configuration information is returned in a list of `ifreq` structures pointed to by the `ifc.ifc_req` field, with one `ifreq` structure per interface.     |
| cmd                 |                                                                                                                                                                                  |
| ioctl(fd, SIOCGIFCONF, (caddr_t)&ifc); | struct ifconf ifc;                                                                                                                                                                   |

The configuration information is returned in a list of `ifreq` structures pointed to by the `ifc.ifc_req` field, with one `ifreq` structure per interface. **Note:** The caller of the ioctl command must allocate sufficient space to store the configuration information, returned as a list of `ifreq` structures for all of the interfaces configured on the system. For example, if `n` interfaces are configured on the system, `ifc.ifc_req` must point to `{n * sizeof (struct ifreq)}` bytes of space allocated. **Note:** Alternatively, the SIOCGBSIZIFCONF ioctl command can be used for this purpose.

| SIOCSIFFLAGS        | Sets the interface flags. SIOCGIFFLAGS gets the interface flags.                                                                                                               |
| cmd                 |                                                                                                                                                                                  |
| ioctl(fd, cmd, (caddr_t)&ifr); | struct ifreq ifr;                                                                                                                                                                   |

Refer to `/usr/include/net/if.h` for the interface flags, denoted by `IFF_XXX`. **Note:** The `IFF_BROADCAST`, `IFF_POINTTOPOINT`, `IFF_SIMPLEX`, `IFF_RUNNING`, `IFF_OACTIVE`, and `IFF_MULTICAST` flags cannot be changed using ioctl.

| SIOCSIFMETRIC       | Sets the interface metric specified in the `ifr.ifr_metric` field. SIOCGIFMETRIC gets the interface metric. The interface metric is placed in the `ifr.ifr_metric` field on return. |
| cmd                 |                                                                                                                                                                                  |
| ioctl(fd, cmd, (caddr_t)&ifr); | struct ifreq ifr;                                                                                                                                                                   |

<p>| SIOCSIFSUBCHAN      | Sets the subchannel address specified in the <code>ifr.ifr_flags</code> field. SIOCGIFSUBCHAN gets the subchannel address in the <code>ifr.ifr_flags</code> field.                                           |
| cmd                 |                                                                                                                                                                                  |
| ioctl(fd, SIOCSIFSUBCHAN, (caddr_t)&amp;ifr); | struct ifreq ifr;                                                                                                                                                                   |</p>
<table>
<thead>
<tr>
<th>ioctl command</th>
<th>Description</th>
</tr>
</thead>
</table>
| SIOCSIFOPTIONS | SIOCSIFOPTIONS sets the interface options. SIOCGIFOPTIONS gets the interface options. ioctl(fd, SIOCSIFOPTIONS, (caddr_t)&ifr); struct ifreq ifr;  
The interface options are stored in the ifr_flags field of the ifreq structure. Refer to /usr/include/net/if.h file for the list of interface options denoted by IFO_xxx. |
| SIOCDELMULTI | SIOCDELMULTI deletes a multicast address from the list of multicast addresses for an interface. ioctl(fd, cmd, (caddr_t)&ifr); struct ifreq ifr;  
  cmd Set to SIOCDELMULTI or SIOCDELMULTI.  
The multicast address information is specified in the ifr_addr structure. |
| SIOCADDMULTI | SIOCADDMULTI adds an address to the list of multicast addresses for an interface. ioctl(fd, cmd, (caddr_t)&ifr); struct ifreq ifr;  
  cmd Set to SIOCADDMULTI or SIOCDELMULTI.  
The multicast address information is specified in the ifr_addr structure. |
| SIOCGIFMTU | SIOCSIFMTU sets the interface maximum transmission unit (MTU). SIOCGIFMTU gets the interface MTU. ioctl(fd, cmd, (caddr_t)&ifr); struct ifreq ifr;  
The MTU value is stored in ifr.ifr_mtu field.  
**Note:** The range of valid values for MTU varies for an interface depending on the interface type. |
<p>| SIOCIFATTACH | SIOCIFATTACH attaches an interface. This initializes and adds an interface in the network interface list. SIOCIFDETACH detaches an interface broadcast address. This removes the interface from the network interface list. The interface name is specified in the ifr.ifr_name field. ioctl(fd, cmd, (caddr_t)&amp;ifr); struct ifreq ifr; |
| SIOCIFDETACH |  |
| SIOCSIFGIDLIST | SIOCSIFGIDLIST adds or deletes the list of group IDs specified in the ifrg.ifrg_gidlist field to the gidlist interface. The interface name is specified in the ifrg.ifrg_name field. An operation code, ADD_GRP/DEL_GRP, specified in the ifrg.ifrg_gidlist field indicates whether the specified list of group IDs must be added to or deleted from the gidlist interface. SIOCGIFGIDLIST gets the list of group IDs associated with an interface. The group IDs are placed in the ifrg.ifrg_gidlist field on return. ioctl(fd, cmd, (caddr_t)&amp;ifr); struct ifgreq ifrg; |</p>
<table>
<thead>
<tr>
<th>ioctl command</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>SIOCIF_ATM_UBR</td>
<td>SIOCIF_ATM_UBR sets the UBR rate for an ATM interface.</td>
</tr>
<tr>
<td>SIOCIF_ATM_SNMPARP</td>
<td>SIOCIF_ATM_SNMPARP gets the SNMP ATM ARP entries.</td>
</tr>
<tr>
<td>SIOCIF_ATM_DUMPARP</td>
<td>SIOCIF_ATM_DUMPARP gets the specified number of ATM ARP entries.</td>
</tr>
<tr>
<td>SIOCIF_ATM_IDLE</td>
<td>SIOCIF_ATM_IDLE sets the idle time limit on the interface.</td>
</tr>
<tr>
<td>SIOCIF_ATM_DARP</td>
<td>SIOCIF_ATM_DARP deletes an ATM ARP entry from the ARP table.</td>
</tr>
<tr>
<td>SIOCIF_ATM_GARP</td>
<td>SIOCIF_ATM_GARP gets an ATM ARP entry to the ARP table.</td>
</tr>
<tr>
<td>SIOCIF_ATM_SARP</td>
<td>SIOCIF_ATM_SARP adds an ATM ARP entry. The ARP information is specified in the atm_arpreq structure.</td>
</tr>
<tr>
<td>SIOCIF_ATM_SVC</td>
<td>SIOCIF_ATM_SVC specifies whether this interface supports Permanent Virtual Circuit (PVC) and Switched Virtual Circuit (SVC) types of virtual connections. It also specifies whether this interface will be an ARP client or an ARP server for this Logical IP Subnetwork (LIS) based on the flag set in the ifatm_svc_arg structure.</td>
</tr>
<tr>
<td>SIOCIF_ATM_IDLE</td>
<td>SIOCIF_ATM_IDLE specifies the idle time limit on the interface.</td>
</tr>
<tr>
<td>SIOCGIFBAUDRATE</td>
<td>Gets the value of the interface baud rate in the ifr_baudrate field.</td>
</tr>
<tr>
<td>SIOCADDIFVIPA</td>
<td>SIOCADDIFVIPA associates the specified list of interfaces pointed to by ifrv.ifrv_ifname with the virtual interface specified by ifrv.ifrv_name. This causes the source address for all outgoing packets on these interfaces to be set to the virtual interface address.</td>
</tr>
<tr>
<td>SIOCDELIFVIPA</td>
<td>SIOCDELIFVIPA removes the list of specified interfaces pointed to by ifrv.ifrv_ifname, that are associated with the virtual interface specified by ifrv.ifrv_name, using SIOCADDIFVIPA.</td>
</tr>
<tr>
<td>SIOCLISTIFVIPA</td>
<td>SIOCLISTIFVIPA lists all the interfaces associated with the virtual interface specified by ifrv.ifrv_name.</td>
</tr>
</tbody>
</table>

Note: These flags operate on a virtual interface only. Also, these flags are available beginning with AIX 5.2 and later.

Return Values
Upon successful completion, ioctl returns 0. Otherwise, it returns -1 and sets errno to indicate the error.

Error Codes
The ioctl commands fail under the following general conditions:

- EBADF  The file descriptor fd is not a valid open socket file descriptor.
- EINTR   A signal was caught during ioctl operation.
- EINVAL  An invalid command or argument was specified.

If the underlying operation specified by the ioctl command cmd failed, ioctl fails with one of the following error codes:
**EACCES**  Permission denied for the specified operation.
**EADDRNOTAVAIL**  Specified address not available for interface.
**EAFNOSUPPORT**  Operation not supported on sockets.
**EBUSY**  Resource is busy.
**EEXIST**  An entry or file already exists.
**EFAULT**  Argument references an inaccessible memory area.
**EIO**  I/O error.
**ENETUNREACH**  Gateway unreachable.
**ENOBUFS**  Routing table overflow.
**ENOCONNECT**  No connection.
**ENOMEM**  Not enough memory available.
**ENOTCONN**  The operation is only defined on a connected socket, but the socket was not connected.
**ENXIO**  Device does not exist.
**ESRCH**  No such process.

**Related Information**

Socket Overview

The **ioctl** subroutine.

"ioctl Streams Device Driver Operations" on page 265.

---

**isinet_addr Subroutine**

**Purpose**
Determines if the given ASCII string contains an Internet address using dot notation.

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

**Syntax**

```c
#include <sys/types.h>
#include <netinet/in.h>

u_long isinet_addr (char *name);
```

**Description**

The **isinet_addr** subroutine determines if the given ASCII string contains an Internet address using dot notation (for example, "120.121.122.123"). The **isaddr_inet** subroutine considers Internet address strings as a valid string, and considers any other string type as an invalid strings.

The **isinet_addr** subroutine expects the ASCII string to conform to the following format:

```
string ::= field | field delimited_field^1-3
delimited_field ::= delimiter field
delimiter ::= 
field ::= 0 X | 0 x | 0 X hexadecimal* | 0 x hexadecimal* | decimal* | 0 octal*
hexadecimal ::= decimal | a | b | c | d | e | f | A | B | C | D | E | F
decimal ::= octal | 8 | 9
octal ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7
```

Chapter 2. Sockets 143
The BNF description explicitly states the space character (" "), if used.

Value | Description
--- | ---
A^n | Indicates \( n \) repetitions of pattern A.
A^n-m | Indicates \( n \) to \( m \) repetitions of pattern A.
A* | Indicates zero or more repetitions of pattern A, up to environmental limits.

The `isinet_addr` subroutine allows the application to terminate the string with a null terminator (0x00) or a space (0x30). It ignores characters trailing the space character and considers the string invalid if the application does not terminate the string with a null terminator (0x00) or space (0x30).

The following describes the restrictions on the field values:

<table>
<thead>
<tr>
<th>Address Format</th>
<th>Field Restrictions (values in decimal base)</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>( a &lt; 4294967296 ).</td>
</tr>
<tr>
<td>a.b</td>
<td>( a &lt; 256; b &lt; 16777216 ).</td>
</tr>
<tr>
<td>a.b.c</td>
<td>( a &lt; 256; b &lt; 256; c &lt; 16777216 ).</td>
</tr>
<tr>
<td>a.b.c.d</td>
<td>( a &lt; 256; b &lt; 2^8; c &lt; 256; d &lt; 256 ).</td>
</tr>
</tbody>
</table>

The `isinet_addr` subroutine applications can enter field values exceeding the field value restrictions specified previously; `isinet_addr` accepts the least significant bits up to an integer in length. The `isinet_addr` subroutine still checks to see if the truncated value exceeds the maximum field value. For example, if an application gives the string `0.0;0;0xFF00000001` then `isinet_addr` interprets the string as `0.0.0.0x00000001` and considers the string as valid.

`isinet_addr` applications cannot omit field values between delimiters and considers a string with successive periods as invalid.

Examples of valid strings:

<table>
<thead>
<tr>
<th>Input String</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td><code>isinet_addr</code> uses ( a ) format.</td>
</tr>
<tr>
<td>1.2</td>
<td><code>isinet_addr</code> uses ( a.b ) format.</td>
</tr>
<tr>
<td>1.2.3.4</td>
<td><code>isinet_addr</code> uses ( a.b.c.d ) format.</td>
</tr>
<tr>
<td>0x01.0X2.03.004</td>
<td><code>isinet_addr</code> uses ( a.b.c.d ) format.</td>
</tr>
<tr>
<td>1.2 3.4</td>
<td><code>isinet_addr</code> uses ( a.b ) format; and ignores &quot;3.4&quot;.</td>
</tr>
</tbody>
</table>

Examples of invalid strings:

<table>
<thead>
<tr>
<th>Input String</th>
<th>Reason</th>
</tr>
</thead>
<tbody>
<tr>
<td>...</td>
<td>No explicit field values specified.</td>
</tr>
<tr>
<td>1.2.3.4.5</td>
<td>Excessive fields.</td>
</tr>
<tr>
<td>1.2.3.4.</td>
<td>Excessive delimiters and fields.</td>
</tr>
<tr>
<td>1,2</td>
<td>Bad delimiter.</td>
</tr>
<tr>
<td>1p</td>
<td>String not terminated by null terminator nor space.</td>
</tr>
<tr>
<td>(empty string)</td>
<td>No field or delimiter present.</td>
</tr>
<tr>
<td>9999.1.1.1</td>
<td>Value for field ( a ) exceeds limit.</td>
</tr>
</tbody>
</table>
Notes:
1. The `isinet_addr` subroutine does not check the pointer to the ASCII string; the user takes responsibility for ensuring validity of the address of the ASCII string.
2. The application assumes responsibility for verifying that the network number and host number for the Internet address conforms to a class A or B or C Internet address; any other string is processed as a class C address.

All applications using `isinet_addr` must compile with the `_BSD` macro defined. Also, all socket applications must include the BSD library `libbsd` when applicable.

Parameters

`name` Address of ASCII string buffer.

Return Values
The `isinet_addr` subroutine returns 1 for valid input strings and 0 for invalid input strings. `isinet_addr` returns the value as an unsigned long type.

Files

```c
#include <ctype.h>
#include <sys/types.h>
```

Related Information

Internet address conversion subroutines: `inet_addr` subroutine, `inet_lnaof` subroutine, `inet_makeaddr` subroutine, `inet_netof` subroutine, `inet_network` subroutine, `inet_ntoa` subroutine.

Host information retrieval subroutines: `endhostent` subroutine, `gethostbyaddr` subroutine, `gethostbyname` subroutine, `gethostent` subroutine.

Network information retrieval subroutines: `getnetbyaddr` subroutine, `getnetbyname` subroutine, `getnetent` subroutine.

---

`kvalid_user` Subroutine

Purpose
This routine maps the DCE principal to the local user account and determines if the DCE principal is allowed access to the account.

Library
Valid User Library (libvaliduser.a)

Syntax

Description
This routine is called when Kerberos 5 authentication is configured to determine if the incoming Kerberos 5 ticket should allow access to the local account.

This routine determines whether the DCE principal, specified by the `princ_name` parameter, is allowed access to the user’s account identified by the `local_user` parameter. The routine accesses the `$HOME/.k5login` file for the users account. It looks for the string pointed to by `princ_name` in that file.
Access is granted if one of two things is true.

1. The $HOME/.k5login file exists and the princ_name is in it.
2. The $HOME/.k5login file does NOT exist and the DCE principal name is the same as the local user’s name.

**Parameters**

- **princ_name**
  This parameter is a single-string representation of the Kerberos 5 principal. The Kerberos 5 libraries have two services, krb5_unparse_name and krb5_parse_name, which convert a krb5_principal structure to and from a single-string format. This routine expects the princ_name parameter to be a single-string form of the krb5_principal structure.

- **local_user**
  This parameter is the character string holding the name of the local account.

**Return Values**

If the user is allowed access to the account, the kvalid_user routine returns TRUE.

If the user is NOT allowed access to the account or there was an error, the kvalid_user routine returns FALSE.

**Related Information**

The [ftp command](#), [rcp command](#), [rlogin command](#), [rsh command](#), [telnet](#), [tn](#), or [tn3270](#) command.

Using a .k5login file.

[Communications and networks](#) in Networks and communication management.

[Authentication and the secure rcmds](#) in Networks and communication management.

---

### listen Subroutine

**Purpose**

Listens for socket connections and limits the backlog of incoming connections.

**Library**

Standard C Library (libc.a)

**Syntax**

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

int listen (Socket, Backlog);
int Socket, Backlog;
```

**Description**

The listen subroutine performs the following activities:

1. Identifies the socket that receives the connections.
2. Marks the socket as accepting connections.
3. Limits the number of outstanding connection requests in the system queue.
The outstanding connection request queue length limit is specified by the parameter `backlog` per listen call. A `no` parameter - `somaxconn` - defines the maximum queue length limit allowed on the system, so the effective queue length limit will be either `backlog` or `somaxconn`, whichever is smaller.

All applications containing the `listen` subroutine must be compiled with the `_BSD` macro set to a specific value. Acceptable values are 43 and 44. In addition, all socket applications must include the BSD `libbsd.a` library.

**Parameters**

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>Socket</code></td>
<td>Specifies the unique name for the socket.</td>
</tr>
<tr>
<td><code>Backlog</code></td>
<td>Specifies the maximum number of outstanding connection requests.</td>
</tr>
</tbody>
</table>

**Return Values**

Upon successful completion, the `listen` subroutine returns a value 0.

If the `listen` subroutine is unsuccessful, the subroutine handler performs the following functions:

- Returns a value of -1 to the calling program.
- Moves an error code, indicating the specific error, into the `errno` global variable.

**Error Codes**

The subroutine is unsuccessful if any of the following errors occurs:

<table>
<thead>
<tr>
<th>Error</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>EBADF</code></td>
<td>The <code>Socket</code> parameter is not valid.</td>
</tr>
<tr>
<td><code>ECONNREFUSED</code></td>
<td>The host refused service, usually due to a server process missing at the requested name or the request exceeding the backlog amount.</td>
</tr>
<tr>
<td><code>EINVAL</code></td>
<td>The socket is already connected.</td>
</tr>
<tr>
<td><code>ENOTSOCK</code></td>
<td>The <code>Socket</code> parameter refers to a file, not a socket.</td>
</tr>
<tr>
<td><code>EOPNOTSUPP</code></td>
<td>The referenced socket is not a type that supports the <code>listen</code> subroutine.</td>
</tr>
</tbody>
</table>

**Examples**

The following program fragment illustrates the use of the `listen` subroutine with 5 as the maximum number of outstanding connections which may be queued awaiting acceptance by the server process.

```c
listen(s,5)
```

**Related Information**

The `accept` subroutine, `connect` subroutine, `socket` subroutine.

**ntohl Subroutine**

**Purpose**

Converts an unsigned long integer from Internet network standard byte order to host byte order.

**Library**

ISODE Library (`libisode.a`)
## Syntax

```c
#include <sys/types.h>
#include <netinet/in.h>

uint32_t ntohl (NetLong) 
uint32_t NetLong;
```

## Description

The `ntohl` subroutine converts an unsigned long (32-bit) integer from Internet network standard byte order to host byte order.

Receiving hosts require addresses and ports in host byte order. Use the `ntohl` subroutine to convert Internet addresses and ports to the host integer representation.

The `ntohl` subroutine is defined in the `net/nh.h` file as a null macro if the host byte order is same as the network byte order.

The `ntohl` subroutine is declared in the `net/nh.h` file as a function if the host byte order is not same as the network byte order.

All applications containing the `ntohl` subroutine must be compiled with the `_BSD` macro set to a specific value. Acceptable values are 43 and 44. In addition, all socket applications must include the BSD `libbsd.a` library.

### Parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>NetLong</code></td>
<td>Requires a 32-bit integer in network byte order.</td>
</tr>
</tbody>
</table>

### Return Values

The `ntohl` subroutine returns a 32-bit integer in host byte order.

### Related Information

The `endhostent` subroutine, `endservent` subroutine, `gethostbyaddr` subroutine, `gethostbyname` subroutine, `getservbyname` subroutine, `getservbyport` subroutine, `getservent` subroutine, `htons` subroutine, `htons` subroutine, `htons` subroutine, `ntohll` subroutine, `ntohl` subroutine, `sethostent` subroutine, `sethostent` subroutine, `sethostent` subroutine, `sethostent` subroutine.

[Sockets Overview in AIX 5L Version 5.3 Communications Programming Concepts.](#)
uint64_t ntohll (NetLong)
uint64_t NetLong;

Description
The ntohll subroutine converts an unsigned long (64-bit) integer from Internet network standard byte order to host byte order.

Receiving hosts require addresses and ports in host byte order. Use the ntohll subroutine to convert Internet addresses and ports to the host integer representation.

The ntohll subroutine is defined in the net/nh.h file as a null macro if the host byte order is the same as the network byte order.

The ntohll subroutine is declared in the net/nh.h file as a function if the host byte order is not the same as the network byte order.

All applications containing the ntohll subroutine must be compiled with the _BSD macro set to a specific value. Acceptable values are 43 and 44. In addition, all socket applications must include the BSD libbsd.a library.

Parameters
NetLong Requires a 64-bit integer in network byte order.

Return Values
The ntohll subroutine returns a 64-bit integer in host byte order.

Related Information
The endhostent subroutine, endservent subroutine, gethostbyaddr subroutine, gethostbyname subroutine, getservbyname subroutine, getservbyport subroutine, getservent subroutine, htonl subroutine, ntops subroutine, ntohll subroutine, ntohs subroutine, htonl subroutine, ntohll subroutine, sethostent subroutine, setservent subroutine.

Sockets Overview in AIX 5L Version 5.3 Communications Programming Concepts.

ntohs Subroutine

Purpose
Converts an unsigned short integer from Internet network byte order to host byte order.

Library
ISODE Library (libisode.a)

Syntax
#include <sys/types.h>
#include <netinet/in.h>

uint16_t ntohs (NetShort)
uint16_t NetShort;
**Description**

The `ntohs` subroutine converts an unsigned short (16-bit) integer from Internet network byte order to the host byte order.

Receiving hosts require Internet addresses and ports in host byte order. Use the `ntohs` subroutine to convert Internet addresses and ports to the host integer representation.

The `ntohs` subroutine is defined in the `net/nh.h` file as a null macro if the host byte order is same as the network byte order.

The `ntohs` subroutine is declared in the `net/nh.h` file as a function if the host byte order is not same as the network byte order.

All applications containing the `ntohs` subroutine must be compiled with the `_BSD` macro set to a specific value. Acceptable values are 43 and 44. In addition, all socket applications must include the BSD `libbsd.a` library.

**Parameters**

- **NetShort** Requires a 16-bit integer in network standard byte order.

**Return Values**

The `ntohs` subroutine returns a 16-bit integer in host byte order.

**Related Information**

The `endhostent` subroutine, `endservent` subroutine, `gethostbyaddr` subroutine, `gethostbyname` subroutine, `getservbyname` subroutine, `getservbyport` subroutine, `getservent` subroutine, `htonl` subroutine, `htons` subroutine, `ntohl` subroutine, `ntohll` subroutine, `sethostent` subroutine, `setservent` subroutine.

[Sockets Overview](AIX 5L Version 5.3 Communications Programming Concepts)

---

**PostQueuedCompletionStatus Subroutine**

**Purpose**

Post a completion packet to a specified I/O completion port.

**Syntax**

```c
#include <iocp.h>

boolean_t PostQueuedCompletionStatus (CompletionPort, TransferCount, CompletionKey, Overlapped, )
HANDLE CompletionPort;
DWORD TransferCount, CompletionKey;
LPOVERLAPPED Overlapped;
```

**Description**

The `PostQueuedCompletionStatus` subroutine attempts to post a completion packet to `CompletionPort` with the values of the completion packet populated by the `TransferCount`, `CompletionKey`, and `Overlapped` parameters.

The `PostQueuedCompletionStatus` subroutine returns a boolean indicating whether or not a completion packet has been posted.
The PostQueuedCompletionStatus subroutine is part of the I/O Completion Port (IOCP) kernel extension.

**Note:** This subroutine only works to a socket file descriptor. It does not work with files or other file descriptors.

**Parameters**

- **CompletionPort**
  
  Specifies the completion port that this subroutine will attempt to access.

- **TransferCount**
  
  Specifies the number of bytes transferred.

- **CompletionKey**
  
  Specifies the completion key.

- **Overlapped**
  
  Specifies the overlapped structure.

**Return Values**

Upon successful completion, the PostQueuedCompletionStatus subroutine returns a boolean indicating its success.

If the PostQueuedCompletionStatus subroutine is unsuccessful, the subroutine handler performs the following functions:

- Returns a value of 0 to the calling program.
- Moves an error code, indicating the specific error, into the errno global variable. For further explanation of the errno variable, see the link in the Related Information section of this document.

**Error Codes**

The subroutine is unsuccessful if either of the following errors occur:

- **EBADF**
  
  The CompletionPort parameter was NULL.

- **EINVAL**
  
  The CompletionPort parameter was invalid.

**Examples**

The following program fragment illustrates the use of the PostQueuedCompletionStatus subroutine to post a completion packet.

```c
int c = GetQueuedCompletionStatus (34, 128, 25, struct overlapped);
```

**Related Information**


For further explanation of the errno variable, see Error Notification Object Class in AIX 5L Version 5.3 General Programming Concepts: Writing and Debugging Programs

---

**rcmd Subroutine**

**Purpose**

Allows execution of commands on a remote host.

**Library**

Standard C Library (libc.a)
Syntax

```c
int rcmd (Host,
Port, LocalUser, RemoteUser, Command, ErrFileDesc)
char ** Host;
short Port;
char * LocalUser;
char * RemoteUser;
char * Command;
int * ErrFileDesc;
```

Description

The `rcmd` subroutine allows execution of certain commands on a remote host that supports `rshd`, `rlogin`, and `rpc` among others.

Only processes with an effective user ID of root user can use the `rcmd` subroutine. An authentication scheme based on remote port numbers is used to verify permissions. Ports in the range between 0 and 1023 can only be used by a root user. The application must pass in `Port`, which must be in the range 512 to 1023.

The `rcmd` subroutine looks up a host by way of the name server or if the local name server isn’t running, in the `/etc/hosts` file.

If the connection succeeds, a socket in the Internet domain of type `SOCK_STREAM` is returned to the calling process and given to the remote command as standard input (`stdin`) and standard output (`stdout`).

Always specify the `Host` parameter. If the local domain and remote domain are the same, specifying the domain parts is optional.

All applications containing the `rcmd` subroutine must be compiled with the `_BSD` macro set to a specific value. Acceptable values are 43 and 44. In addition, all socket applications must include the BSD `libbsd.a` library.

Parameters

- **Host**
  Specifies the name of a remote host that is listed in the `/etc/hosts` file. If the specified name of the host is not found in this file, the `rcmd` subroutine is unsuccessful.

- **Port**
  Specifies the well-known port to use for the connection. The `/etc/services` file contains the DARPA Internet services, their ports, and socket types.

- **LocalUser** and **RemoteUser**
  Points to user names that are valid at the local and remote host, respectively. Any valid user name can be given.

- **Command**
  Specifies the name of the command to be started at the remote host.
Specifies an integer controlling the set up of communication channels. Integer options are as follows:

**Non-zero**
Indicates an auxiliary channel to a control process is set up, and the `ErrFileDesc` parameter points to the file descriptor for the channel. The control process provides diagnostic output from the remote command on this channel and also accepts bytes as signal numbers to be forwarded to the process group of the command.

**0**
Indicates the standard error (stderr) of the remote command is the same as standard output (stdout). No provision is made for sending arbitrary signals to the remote process. However, it is possible to send out-of-band data to the remote command.

**Return Values**
Upon successful completion, the `rcmd` subroutine returns a valid socket descriptor.

Upon unsuccessful completion, the `rcmd` subroutine returns a value of -1. The subroutine returns a -1, if the effective user ID of the calling process is not root user or if the subroutine is unsuccessful to resolve the host.

**Files**
- `/etc/services`
- `/etc/hosts`
- `/etc/resolv.conf`
  
  Contains the service names, ports, and socket type.
  Contains host names and their addresses for hosts in a network.
  Contains the name server and domain name.

**Related Information**
The `rlogind` command, `rshd` command.

The `named` daemon.

The `gethostname` subroutine, `rresvport` subroutine, `ruserok` subroutine, `sethostname` subroutine.

[Sockets Overview in AIX 5L Version 5.3 Communications Programming Concepts](#).

**rcmd_af Subroutine**

**Purpose**
Allows execution of commands on a remote host.

**Syntax**

```c
int rcmd_af(char **ahost, unsigned short rport,
            const char *locuser, const char *remuser,
            const char *cmd, int *fd2p, int af)
```

**Description**
The `rcmd_af` subroutine allows execution of certain commands on a remote host that supports `rshd`, `rlogin`, and `rpc` among others. It behaves the same as the existing `rcmd()` function, but instead of creating
only an AF_INET TCP socket, it can also create an AF_INET6 TCP socket. The existing rcmd() function cannot transparently use AF_INET6 sockets because an application would not be prepared to handle AF_INET6 addresses returned by subroutines such as getpeername() on the file descriptor created by rcmd().

Only processes with an effective user ID of root user can use the rcmd_af subroutine. An authentication scheme based on remote port numbers is used to verify permissions. Ports in the range between 0 and 1023 can only be used by a root user.

The rcmd_af subroutine looks up a host by way of the name server or if the local name server is not running, in the /etc/hosts file.

If the connection succeeds, a socket in the Internet domain of type SOCK_STREAM is returned to the calling process and given to the remote command as standard input (stdin) and standard output (stdout).

Always specify the ahost parameter. If the local domain and remote domain are the same, specifying the domain parts is optional.

**Parameters**

- **ahost**: Specifies the name of a remote host that is listed in the /etc/hosts file. If the specified name of the host is not found in this file, the rcmd_af subroutine is unsuccessful.
- **rport**: Specifies the well-known port to use for the connection. The /etc/services file contains the DARPA Internet services, their ports, and socket types.
- **locuser**: Points to user names that are valid at the local host. Any valid user name can be given.
- **remuser**: Points to user names that are valid at the remote host. Any valid user name can be given.
- **cmd**: Specifies the name of the command to be started at the remote host.
- **fd2p**: Specifies an integer controlling the set up of communication channels. Integer options are as follows:
  - Non-zero: Indicates an auxiliary channel to a control process is set up, and the fd2p parameter points to the file descriptor for the channel. The control process provides diagnostic output from the remote command on this channel and also accepts bytes as signal numbers to be forwarded to the process group of the command.
  - 0: Indicates the standard error (stderr) of the remote command is the same as standard output (stdout). No provision is made for sending arbitrary signals to the remote process. However, it is possible to send out-of-band data to the remote command.
- **af**: The family argument is AF_INET, AF_INET6, or AF_UNSPEC. When either AF_INET or AF_INET6 is specified, this function will create a socket of the specified address family. When AF_UNSPEC is specified, it will try all possible address families until a connection can be established, and will return the associated socket of the connection.

**Return Values**

Upon successful completion, the rcmd_af subroutine returns a valid socket descriptor. Upon unsuccessful completion, the rcmd_af subroutine returns a value of −1. The subroutine returns a −1 if the effective user ID of the calling process is not the root user or if the subroutine is unsuccessful to resolve the host.

**Files**

- /etc/services
- /etc/hosts
- /etc/resolv.conf

Contains the service names, ports, and socket type.
Contains host names and their addresses for hosts in a network.
Contains the name server and domain name.
Related Information

"rcmd Subroutine" on page 151, "rexec_af Subroutine" on page 174, "resvport_af Subroutine" on page 176

ReadFile Subroutine

Purpose
Reads data from a socket.

Syntax

```c
#include <iocp.h>

boolean_t ReadFile (FileDescriptor, Buffer, ReadCount, AmountRead, Overlapped);
HANDLE FileDescriptor;
LPVOID Buffer;
DWORD ReadCount;
LPDWORD AmountRead;
LPOVERLAPPED Overlapped;
```

Description
The ReadFile subroutine reads the number of bytes specified by the ReadCount parameter from the FileDescriptor parameter into the buffer indicated by the Buffer parameter. The number of bytes read is saved in the AmountRead parameter. The Overlapped parameter indicates whether or not the operation can be handled asynchronously.

The ReadFile subroutine returns a boolean (an integer) indicating whether or not the request has been completed.

The ReadFile subroutine is part of the I/O Completion Port (IOCP) kernel extension.

Note: This subroutine only works to a socket file descriptor. It does not work with files or other file descriptors.

Parameters

- **FileDescriptor**: Specifies a valid file descriptor obtained from a call to the `socket` or `accept` subroutines.
- **Buffer**: Specifies the buffer from which the data will be read.
- **ReadCount**: Specifies the maximum number of bytes to read.
- **AmountRead**: Specifies the number of bytes read. The parameter is set by the subroutine.
- **Overlapped**: Specifies an overlapped structure indicating whether or not the request can be handled asynchronously.

Return Values
Upon successful completion, the ReadFile subroutine returns a boolean indicating the request has been completed.

If the ReadFile subroutine is unsuccessful, the subroutine handler performs the following functions:

- Returns a value of 0 to the calling program.
- Moves an error code, indicating the specific error, into the errno global variable. For further explanation of the errno variable, see the link in the Related Information section of this document.
Error Codes

The subroutine is unsuccessful if any of the following errors occur:

- **EINPROGRESS**
  The read request can not be immediately satisfied and will be handled asynchronously. A completion packet will be sent to the associated completion port upon completion.

- **EAGAIN**
  The read request cannot be immediately satisfied and cannot be handled asynchronously.

- **EINVAL**
  The `FileDescriptor` parameter is invalid.

Examples

The following program fragment illustrates the use of the **ReadFile** subroutine to synchronously read data from a socket:

```c
void buffer;
int amount_read;
b = ReadFile (34, &buffer, 128, &amount_read, NULL);
```

The following program fragment illustrates the use of the **ReadFile** subroutine to asynchronously read data from a socket:

```c
void buffer;
int amount_read;
LPOVERLAPPED overlapped;
b = ReadFile (34, &buffer, 128, &amount_read, overlapped);
```

**Note:** The request will only be handled asynchronously if it cannot be immediately satisfied.

Related Information

The ["socket Subroutine" on page 226](#), ["accept Subroutine" on page 29](#), ["CreateIoCompletionPort Subroutine" on page 36](#), ["WriteFile Subroutine" on page 243](#), ["GetQueuedCompletionStatus Subroutine" on page 91](#), and ["PostQueuedCompletionStatus Subroutine" on page 150](#).

For further explanation of the **errno** variable, see [Error Notification Object Class](#) in *AIX 5L Version 5.3 General Programming Concepts: Writing and Debugging Programs*.

recv Subroutine

**Purpose**

Receives messages from connected sockets.

**Library**

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

**Syntax**

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

int recv (Socket, Buffer, Length, Flags)
int Socket;
void * Buffer;
size_t Length;
int Flags;
```
Description

The **recv** subroutine receives messages from a connected socket. The **recvfrom** and **recvmsg** subroutines receive messages from both connected and unconnected sockets. However, they are usually used for unconnected sockets only.

The **recv** subroutine returns the length of the message. If a message is too long to fit in the supplied buffer, excess bytes may be truncated depending on the type of socket that issued the message.

If no messages are available at the socket, the **recv** subroutine waits for a message to arrive, unless the socket is nonblocking. If a socket is nonblocking, the system returns an error.

Use the **select** subroutine to determine when more data arrives.

The socket applications can be compiled with **COMPAT_43** defined. This will make the **sockaddr** structure BSD 4.3 compatible. For more details refer to **socket.h**.

Parameters

- **Socket**: Specifies the socket descriptor.
- **Buffer**: Specifies an address where the message should be placed.
- **Length**: Specifies the size of the **Buffer** parameter.
- **Flags**: Points to a value controlling the message reception. The **/usr/include/sys/socket.h** file defines the **Flags** parameter. The argument to receive a call is formed by logically ORing one or more of the following values:
  - **MSG_OOB**: Processes out-of-band data. The significance of out-of-band data is protocol-dependent.
  - **MSG_PEEK**: Peeks at incoming data. The data continues to be treated as unread and will be read by the next call to **recv()** or a similar function.
  - **MSG_WAITALL**: Requests that the function not return until the requested number of bytes have been read. The function can return fewer than the requested number of bytes only if a signal is caught, the connection is terminated, or an error is pending for the socket.

Return Values

Upon successful completion, the **recv** subroutine returns the length of the message in bytes.

If the **recv** subroutine is unsuccessful, the subroutine handler performs the following functions:
- Returns a value of -1 to the calling program.
- Returns a 0 if the connection disconnects.
- Moves an error code, indicating the specific error, into the **errno** global variable.

Error Codes

The **recv** subroutine is unsuccessful if any of the following errors occurs:

<table>
<thead>
<tr>
<th>Error</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EBADF</td>
<td>The <strong>Socket</strong> parameter is not valid.</td>
</tr>
<tr>
<td>ECONNRESET</td>
<td>The remote peer forces the connection to be closed.</td>
</tr>
<tr>
<td>EFAULT</td>
<td>The data was directed to be received into a nonexistent or protected part of the process address space. The <strong>Buffer</strong> parameter is not valid.</td>
</tr>
<tr>
<td>EINTR</td>
<td>A signal interrupted the <strong>recv</strong> subroutine before any data was available.</td>
</tr>
<tr>
<td>EINVAL</td>
<td>The <strong>MSG_OOB</strong> flag is set and no out-of-band data is available.</td>
</tr>
</tbody>
</table>
**recvfrom Subroutine**

**Purpose**

Receives messages from sockets.

**Library**

Standard C Library (libc.a)

**Syntax**

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

ssize_t recvfrom
(Socket, Buffer, Length, Flags, From, FromLength)

int Socket;
void *Buffer;
size_t Length;
int Flags;
struct sockaddr *From;
socklen_t *FromLength;
```

**Description**

The `recvfrom` subroutine allows an application program to receive messages from unconnected sockets. The `recvfrom` subroutine is normally applied to unconnected sockets as it includes parameters that allow the calling program to specify the source point of the data to be received.

To return the source address of the message, specify a nonnull value for the `From` parameter. The `FromLength` parameter is a value-result parameter, initialized to the size of the buffer associated with the `From` parameter. On return, the `recvfrom` subroutine modifies the `FromLength` parameter to indicate the actual size of the stored address. The `recvfrom` subroutine returns the length of the message. If a message is too long to fit in the supplied buffer, excess bytes may be truncated depending on the type of socket that issued the message.

If no messages are available at the socket, the `recvfrom` subroutine waits for a message to arrive, unless the socket is nonblocking. If the socket is nonblocking, the system returns an error.
The socket applications can be compiled with `COMPAT_43` defined. This will make the `sockaddr` structure BSD 4.3 compatible. For more details refer to `socket.h`.

**Parameters**

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Socket</strong></td>
<td>Specifies the socket descriptor.</td>
</tr>
<tr>
<td><strong>Buffer</strong></td>
<td>Specifies an address where the message should be placed.</td>
</tr>
<tr>
<td><strong>Length</strong></td>
<td>Specifies the size of the <code>Buffer</code> parameter.</td>
</tr>
<tr>
<td><strong>Flags</strong></td>
<td>Points to a value controlling the message reception. The argument to receive a call is formed by logically ORing one or more of the values shown in the following list:</td>
</tr>
<tr>
<td>MSG_OOB</td>
<td>Processes out-of-band data. The significance of out-of-band data is protocol-dependent.</td>
</tr>
<tr>
<td>MSG_PEEK</td>
<td>Peeks at incoming data. The data continues to be treated as unread and will be read by the next call to <code>recv()</code> or a similar function.</td>
</tr>
<tr>
<td>MSG_WAITALL</td>
<td>Requests that the function not return until the requested number of bytes have been read. The function can return fewer than the requested number of bytes only if a signal is caught, the connection is terminated, or an error is pending for the socket.</td>
</tr>
<tr>
<td><strong>From</strong></td>
<td>Points to a socket structure, filled in with the source's address.</td>
</tr>
<tr>
<td><strong>FromLength</strong></td>
<td>Specifies the length of the sender's or source's address.</td>
</tr>
</tbody>
</table>

**Return Values**

If the `recvfrom` subroutine is successful, the subroutine returns the length of the message in bytes.

If the call is unsuccessful, the subroutine handler performs the following functions:

- Returns a value of -1 to the calling program.
- Moves an error code, indicating the specific error, into the `errno` global variable.

**Error Codes**

The `recvfrom` subroutine is unsuccessful if any of the following errors occurs:

<table>
<thead>
<tr>
<th>Error</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EBADF</td>
<td>The <code>Socket</code> parameter is not valid.</td>
</tr>
<tr>
<td>ECONNRESET</td>
<td>The remote peer forces the connection to be closed.</td>
</tr>
<tr>
<td>EFAULT</td>
<td>The data was directed to be received into a nonexistent or protected part of the process address space. The buffer is not valid.</td>
</tr>
<tr>
<td>EINTR</td>
<td>The receive is interrupted by a signal delivery before any data is available.</td>
</tr>
<tr>
<td>EINVAL</td>
<td>The <code>MSG_OOB</code> flag is set but no out-of-band data is available.</td>
</tr>
<tr>
<td>ENOBUF</td>
<td>Insufficient resources are available in the system to perform the operation.</td>
</tr>
<tr>
<td>ENOPROTOOPT</td>
<td>The protocol is not 64-bit supported.</td>
</tr>
<tr>
<td>ENOTCONN</td>
<td>A receive is attempted on a <code>SOCK_STREAM</code> socket that is not connected.</td>
</tr>
<tr>
<td>ENOTSOCK</td>
<td>The <code>Socket</code> parameter refers to a file, not a socket.</td>
</tr>
<tr>
<td>EOPNOTSUPP</td>
<td><code>MSG_OOB</code> flag is set for a <code>SOCK_DGRAM</code> socket, or <code>MSG_OOB</code> flag is set for any <code>AF_UNIX</code> socket.</td>
</tr>
<tr>
<td>ETIMEDOUT</td>
<td>The connection timed out during connection establishment, or there was a transmission timeout on an active connection.</td>
</tr>
<tr>
<td>EWOULDBLOCK</td>
<td>The socket is marked nonblocking, and no connections are present to be accepted.</td>
</tr>
</tbody>
</table>
recvmsg Subroutine

Purpose
Receives a message from any socket.

Library
Standard C Library (libc.a)

Syntax
#include <sys/socket.h>

int recvmsg (Socket Message Flags)
int Socket;
struct msghdr Message [];
int Flags;

Description
The recvmsg subroutine receives messages from unconnected or connected sockets. The recvmsg subroutine returns the length of the message. If a message is too long to fit in the supplied buffer, excess bytes may be truncated depending on the type of socket that issued the message.

If no messages are available at the socket, the recvmsg subroutine waits for a message to arrive. If the socket is nonblocking and no messages are available, the recvmsg subroutine is unsuccessful.

Use the select subroutine to determine when more data arrives.

The recvmsg subroutine uses a msghdr structure to decrease the number of directly supplied parameters. The msghdr structure is defined in the sys/socket.h file. In BSD 4.3 Reno, the size and members of the msghdr structure have been modified. Applications wanting to start the old structure need to compile with COMPAT_43 defined. The default behavior is that of BSD 4.4.

All applications containing the recvmsg subroutine must be compiled with the _BSD macro set to a specific value. Acceptable values are 43 and 44. In addition, all socket applications must include the BSD libbsd.a library.

Parameters

Socket Specifies the unique name of the socket.
Message Points to the address of the msghdr structure, which contains both the address for the incoming message and the space for the sender address.
Flags

Permits the subroutine to exercise control over the reception of messages. The Flags parameter used to receive a call is formed by logically ORing one or more of the values shown in the following list:

MSG_OOB

Processes out-of-band data. The significance of out-of-band data is protocol-dependent.

MSG_PEEK

Peeks at incoming data. The data continues to be treated as unread and will be read by the next call to recv() or a similar function.

MSG_WAITALL

Requests that the function not return until the requested number of bytes have been read. The function can return fewer than the requested number of bytes only if a signal is caught, the connection is terminated, or an error is pending for the socket.

The /sys/socket.h file contains the possible values for the Flags parameter.

Return Values

Upon successful completion, the length of the message in bytes is returned.

If the recvmsg subroutine is unsuccessful, the subroutine handler performs the following functions:

• Returns a value of -1 to the calling program.
• Moves an error code, indicating the specific error, into the errno global variable.

Error Codes

The recvmsg subroutine is unsuccessful if any of the following error codes occurs:

<table>
<thead>
<tr>
<th>Error</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EBADF</td>
<td>The Socket parameter is not valid.</td>
</tr>
<tr>
<td>ECONNRESET</td>
<td>The remote peer forces the connection to be closed.</td>
</tr>
<tr>
<td>EFAULT</td>
<td>The Address parameter is not in a writable part of the user address space.</td>
</tr>
<tr>
<td>EINTR</td>
<td>The recvmsg subroutine was interrupted by delivery of a signal before any data was available for the receive.</td>
</tr>
<tr>
<td>EINVAL</td>
<td>The length of the msg hdr structure is invalid, or the MSG_OOB flag is set and no out-of-band data is available.</td>
</tr>
<tr>
<td>EMSGSIZE</td>
<td>The msg iovlen member of the msg hdr structure pointed to by Message is less than or equal to 0, or is greater than IOV_MAX.</td>
</tr>
<tr>
<td>ENOBUF</td>
<td>Insufficient resources are available in the system to perform the operation.</td>
</tr>
<tr>
<td>ENOPROTOOPT</td>
<td>The protocol is not 64-bit supported.</td>
</tr>
<tr>
<td>ENOTCONN</td>
<td>A receive is attempted on a SOCK_STREAM socket that is not connected.</td>
</tr>
<tr>
<td>ENOTSOCK</td>
<td>The Socket parameter refers to a file, not a socket.</td>
</tr>
<tr>
<td>EOPNOTSUPP</td>
<td>MSG_OOB flag is set for a SOCK_DGRAM socket, or MSG_OOB flag is set for any AF_UNIX socket.</td>
</tr>
<tr>
<td>ETIMEDOUT</td>
<td>The connection timed out during connection establishment, or there was a transmission timeout on an active connection.</td>
</tr>
<tr>
<td>EWOULDBLOCK</td>
<td>The socket is marked nonblocking, and no connections are present to be accepted.</td>
</tr>
</tbody>
</table>

Related Information

The no command.

The recv subroutine, recvfrom subroutine, select subroutine, send subroutine, sendmsg subroutine, sendto subroutine, shutdown subroutine, socket subroutine.

Sockets Overview and Understanding Socket Data Transfer in AIX 5L Version 5.3 Communications Programming Concepts.
res_init Subroutine

Purpose

Searches for a default domain name and Internet address.

Library

Standard C Library (libc.a)

Syntax

```c
#include <sys/types.h>
#include <netinet/in.h>
#include <arpa/nameser.h>
#include <resolv.h>

void res_init ( )
```

Description

The res_init subroutine reads the /etc/resolv.conf file for the default domain name and the Internet address of the initial hosts running the name server.

Note: If the /etc/resolv.conf file does not exist, the res_init subroutine attempts name resolution using the local /etc/hosts file. If the system is not using a domain name server, the /etc/resolv.conf file should not exist. The /etc/hosts file should be present on the system even if the system is using a name server. In this instance, the file should contain the host IDs that the system requires to function even if the name server is not functioning.

The res_init subroutine is one of a set of subroutines that form the resolver, a set of functions that translate domain names to Internet addresses. All resolver subroutines use the /usr/include/resolv.h file, which defines the _res structure. The res_init subroutine stores domain name information in the _res structure. Three environment variables, LOCALDOMAIN, RES_TIMEOUT, and RES_RETRY, affect default values related to the _res structure.

All applications containing the res_init subroutine must be compiled with the _BSD macro set to a specific value. Acceptable values are 43 and 44. In addition, all socket applications must include the BSD libbsd.a library.

For more information on the _res structure, see “Understanding Domain Name Resolution” in AIX 5L Version 5.3 Communications Programming Concepts.

Files

- /etc/resolv.conf
  Contains the name server and domain name.
- /etc/hosts
  Contains host names and their addresses for hosts in a network. This file is used to resolve a host name into an Internet address.

Related Information

The dn_comp subroutine, dn_expand subroutine, getlong subroutine, getshort subroutine, putlong subroutine, putshort subroutine, res_mkquery subroutine, “res_ninit Subroutine” on page 165, res_query subroutine, res_search subroutine, res_send subroutine.
res_mkquery Subroutine

Purpose
Makes query messages for name servers.

Library
Standard C Library (libc.a)

Syntax

```c
#include <sys/types.h>
#include <netinet/in.h>
#include <arpa/nameser.h>
#include <resolv.h>

int res_mkquery (Operation, DomName, Class, Type, Data, DataLength)

int res_mkquery (Reserved, Buffer, BufferLength)

int Operation;
char *DomName;
int Class, Type;
char *Data;
int DataLength;
struct rrec *Reserved;
char *Buffer;
int BufferLength;
```

Description
The res_mkquery subroutine creates packets for name servers in the Internet domain. The subroutine also creates a standard query message. The Buffer parameter determines the location of this message.

The res_mkquery subroutine is one of a set of subroutines that form the resolver, a set of functions that resolve domain names. Global information used by the resolver subroutines is kept in the _res data structure. The /usr/include/resolv.h file contains the _res structure definition.

All applications containing the res_mkquery subroutine must be compiled with the _BSD macro set to a specific value. Acceptable values are 43 and 44. In addition, all socket applications must include the BSD libbsd.a library.

Parameters

Operation
Specifies a query type. The usual type is QUERY, but the parameter can be set to any of the query types defined in the arpa/nameser.h file.

DomName
Points to the name of the domain. If the DomName parameter points to a single label and the RES_DEFNAMES structure is set, as it is by default, the subroutine appends the DomName parameter to the current domain name. The current domain name is defined by the name server in use or in the /etc/resolv.conf file.

Class
Specifies one of the following parameters:

- C_IN
  Specifies the ARPA Internet.
- C_CHAOS
  Specifies the Chaos network at MIT.
Type

Requires one of the following values:

- **T_A**: Host address
- **T_NS**: Authoritative server
- **T_MD**: Mail destination
- **T_MF**: Mail forwarder
- **T_CNAME**: Canonical name
- **T_SOA**: Start-of-authority zone
- **T_MB**: Mailbox-domain name
- **T_MG**: Mail-group member
- **T_MR**: Mail-rename name
- **T_NULL**: Null resource record
- **T_WKS**: Well-known service
- **T_PTR**: Domain name pointer
- **T_HINFO**: Host information
- **T_MINFO**: Mailbox information
- **T_MX**: Mail-routing information
- **T_UINFO**: User information
- **T_UID**: User ID
- **T_GID**: Group ID

Data

Points to the data that is sent to the name server as a search key. The data is stored as a character array.

DataLength

Defines the size of the array pointed to by the Data parameter.

Reserved

Specifies a reserved and currently unused parameter.

Buffer

Points to a location containing the query message.

BufferLength

Specifies the length of the message pointed to by the Buffer parameter.

Return Values

Upon successful completion, the **res_mkquery** subroutine returns the size of the query. If the query is larger than the value of the **BufferLength** parameter, the subroutine is unsuccessful and returns a value of -1.

Files

/etc/resolv.conf

Contains the name server and domain name.

Related Information

The **finger** command.
The `dn_comp` subroutine, `dn_expand` subroutine, `getlong` subroutine, `getshort` subroutine, `putlong` subroutine, `putshort` subroutine, `res_init` subroutine, `res_ninit` subroutine, `res_query` subroutine, `res_search` subroutine, `res_send` subroutine.

Sockets Overview and Understanding Domain Name Resolution in AIX 5L Version 5.3 Communications Programming Concepts.

res_ninit Subroutine

Purpose
Sets the default values for the members of the `_res` structure.

Library
Standard C Library (`libc.a`)

Syntax
```
#include <resolv.h>

int res_ninit (statp)
res_state statp;
```

Description
Reads the `/etc/resolv.conf` configuration file to get the default domain name, search list, and internet address of the local name server(s). It does this in order to re-initialize the resolver context for a given thread in a multi-threaded environment.

The `res_ninit` subroutine sets the default values for the members of the `_res` structure (defined in the `/usr/include/resolv.h` file) after reading the `/etc/resolv.conf` configuration file to get default domain name, search list, internet address of the local name server(s), sort list, and options (for details, please refer to the `/etc/resolv.conf` file). If no name server is configured, the server address is set to `INADDR_ANY` and the default domain name is obtained from the `gethostname` subroutine. It also allows the user to override retrans, retry, and local domain definition using three environment variables RES_TIMEOUT, RES_RETRY, and LOCALDOMAIN, respectively.

Using this subroutine, each thread can have unique local resolver context. Since the configuration file is read each time the subroutine is called, it is capable of tracking dynamic changes to the resolver state file. Changes include, addition or removal of the configuration file or any other modifications to this file and reflect the same for a given thread. The `res_ninit` subroutine can also be used in single-threaded applications to detect dynamic changes to the resolver file even while the program is running (See the example section below). For more information on the `_res` structure, see Understanding Domain Name Resolution in AIX 5L Version 5.3 Communications Programming Concepts.

Parameters

`statp` Specifies the state to be initialized.

Examples

```
# cat /etc/resolv.conf
domain in.ibm.com
nameserver 9.184.192.240
```

The following two examples use the `gethostbyname` system call to retrieve the host address of a system (florida.in.ibm.com) continuously. In the first example, `gethostbyname` is called (by a thread 'resolver') in a
multi-threaded environment. The second example is not. Before each call to `gethostbyname`, the `res_ninit` subroutine is called to reflect dynamic changes to the configuration file.

1) #include <stdio.h>
    #include <netdb.h>
    #include <resolv.h>
    #include <pthread.h>

    void *resolver (void *arg);
    main( ) {
        pthread_t thid;
        if ( pthread_create(&thid, NULL, resolver, NULL) ) {
            printf("error in thread creation\n");
            exit( );
        }
        pthread_exit(NULL);
    }

    void *resolver (void *arg) {
        struct hostent *hp;
        struct sockaddr_in client;
        while(1) {
            res_ninit(&_res); /* res_init() with RES_INIT unset would NOT work here */
            hp = (struct hostent * ) gethostbyname("florida.in.ibm.com");
            bcopy(hp->h_addr_list[0],&client.sin_addr,sizeof(client.sin_addr));
            printf("hostname: %s\n",inet_ntoa(client.sin_addr));
        }
    }

If the `/etc/resolv.conf` file is present when the thread 'resolver' is invoked, the hostname will be resolved for that thread (using the nameserver 9.184.192.210) and the output will be hostname: 9.182.21.151.

If `/etc/resolv.conf` is not present, the output will be hostname: 0.0.0.0.

2) The changes to `/etc/resolv.conf` file are reflected even while the program is running

    #include <stdio.h>
    #include <resolv.h>
    #include <sys.h>
    #include <netdb.h>
    #include <string.h>

    main() {
        struct hostent *hp;
        struct sockaddr_in client;
        while (1) {
            res_ninit(&_res);
            hp = (struct hostent * ) gethostbyname("florida.in.ibm.com");
            bcopy(hp->h_addr_list[0],&client.sin_addr,sizeof(client.sin_addr));
            printf("hostname: %s\n",inet_ntoa(client.sin_addr));
        }
    }

If `/etc/resolv.conf` is present while the program is running, the hostname will be resolved (using the nameserver 9.184.192.240) and the output will be hostname: 9.182.21.151.

If the `/etc/resolv.conf` file is not present, the output of the program will be hostname: 0.0.0.0.

Note: In the second example, the `res_init` subroutine with _res.options = ~RES_INIT can be used instead of the `res_ninit` subroutine.
Files
The /etc/resolv.conf and /etc/hosts files.

Related Information
The "dn_comp Subroutine" on page 38, "dn_expand Subroutine" on page 39, "getshort Subroutine" on page 26, "getlong Subroutine" on page 25, "putlong Subroutine" on page 27, "putshort Subroutine" on page 28, "res_init Subroutine" on page 162, "res_mkquery Subroutine" on page 163, "res_query Subroutine," "res_search Subroutine" on page 169, "res_send Subroutine" on page 171. Understanding Domain Name resolution

Understanding Domain Name Resolution in AIX 5L Version 5.3 Communications Programming Concepts.

res_query Subroutine

Purpose
Provides an interface to the server query mechanism.

Library
Standard C Library (libc.a)

Syntax
#include <sys/types.h>
#include <netinet/in.h>
#include <arpa/nameser.h>
#include <resolv.h>

int res_query (DomName, Class, Type, Answer, AnswerLength)
char *DomName;
int Class;
int Type;
uchar *Answer;
int AnswerLength;

Description
The res_query subroutine provides an interface to the server query mechanism. It constructs a query, sends it to the local server, awaits a response, and makes preliminary checks on the reply. The query requests information of the specified type and class for the fully-qualified domain name specified in the DomName parameter. The reply message is left in the answer buffer whose size is specified by the AnswerLength parameter, which is supplied by the caller.

The res_query subroutine is one of a set of subroutines that form the resolver, a set of functions that resolve domain names. The _res data structure contains global information used by the resolver subroutines. The /usr/include/resolv.h file contains the _res structure definition.

All applications containing the res_query subroutine must be compiled with the _BSD macro set to a specific value. Acceptable values are 43 and 44. In addition, all socket applications must include the BSD libbsd.a library.
Parameters

*DomName*  
Points to the name of the domain. If the *DomName* parameter points to a single-component name and the RES_DEFNAMES structure is set, as it is by default, the subroutine appends the default domain name to the single-component name. The current domain name is defined by the name server in use or is specified in the `/etc/resolv.conf` file.

*Class*  
Specifies one of the following values:
- `C_IN` Specifies the ARPA Internet.
- `C_CHAOS` Specifies the Chaos network at MIT.

*Type*  
Requires one of the following values:
- `T_A` Host address
- `T_NS` Authoritative server
- `T_MD` Mail destination
- `T_MF` Mail forwarder
- `T_CNAME` Canonical name
- `T_SOA` Start-of-authority zone
- `T_MB` Mailbox-domain name
- `T_MG` Mail-group member
- `T_MR` Mail-rename name
- `T_NULL` Null resource record
- `T_WKS` Well-known service
- `T_PTR` Domain name pointer
- `T_HINFO` Host information
- `T_MINFO` Mailbox information
- `T_MX` Mail-routing information
- `T_UINFO` User (`finger` command) information
- `T_UID` User ID
- `T_GID` Group ID

*Answer*  
Points to an address where the response is stored.

*AnswerLength*  
Specifies the size of the answer buffer.

Return Values

Upon successful completion, the `res_query` subroutine returns the size of the response. Upon unsuccessful completion, the `res_query` subroutine returns a value of -1 and sets the `h_errno` value to the appropriate error.

Files

 `/etc/resolv.conf`  
Contains the name server and domain name.
res_search Subroutine

Purpose
Makes a query and awaits a response.

Library
Standard C Library (libc.a)

Syntax
```
#include <sys/types.h>
#include <netinet/in.h>
#include <arpa/nameser.h>
#include <resolv.h>

int res_search (DomName, Class, Type, Answer, AnswerLength)
char *DomName;
int Class;
int Type;
unsigned char *Answer;
int AnswerLength);
```

Description
The res_search subroutine makes a query and awaits a response like the res_query subroutine. However, it also implements the default and search rules controlled by the RES_DEFNAMES and RES_DNSRCH options.

The res_search subroutine is one of a set of subroutines that form the resolver, a set of functions that resolve domain names. The _res data structure contains global information used by the resolver subroutines. The /usr/include/resolv.h file contains the _res structure definition.

All applications containing the res_search subroutine must be compiled with the _BSD macro set to a specific value. Acceptable values are 43 and 44. In addition, all socket applications must include the BSD libbsd.a library.
**Parameters**

*DomName*  
Points to the name of the domain. If the *DomName* parameter points to a single-component name and the *RES_DEFNAMES* structure is set, as it is by default, the subroutine appends the default domain name to the single-component name. The current domain name is defined by the name server in use or is specified in the `/etc/resolv.conf` file.

If the *RES_DNSRCH* bit is set, as it is by default, the *res_search* subroutine searches for host names in both the current domain and in parent domains.

*Class*  
Specifies one of the following values:

<table>
<thead>
<tr>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>C_IN</td>
<td>Specifies the ARPA Internet.</td>
</tr>
<tr>
<td>C_CHAOS</td>
<td>Specifies the Chaos network at MIT.</td>
</tr>
</tbody>
</table>

*Type*  
Requires one of the following values:

<table>
<thead>
<tr>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>T_A</td>
<td>Host address</td>
</tr>
<tr>
<td>T_NS</td>
<td>Authoritative server</td>
</tr>
<tr>
<td>T_MD</td>
<td>Mail destination</td>
</tr>
<tr>
<td>T_MF</td>
<td>Mail forwarder</td>
</tr>
<tr>
<td>T_CNAME</td>
<td>Canonical name</td>
</tr>
<tr>
<td>T_SOA</td>
<td>Start-of-authority zone</td>
</tr>
<tr>
<td>T_MB</td>
<td>Mailbox-domain name</td>
</tr>
<tr>
<td>T_MG</td>
<td>Mail-group member</td>
</tr>
<tr>
<td>T_MR</td>
<td>Mail-rename name</td>
</tr>
<tr>
<td>T_NULL</td>
<td>Null resource record</td>
</tr>
<tr>
<td>T_WKS</td>
<td>Well-known service</td>
</tr>
<tr>
<td>T_PTR</td>
<td>Domain name pointer</td>
</tr>
<tr>
<td>T_HINFO</td>
<td>Host information</td>
</tr>
<tr>
<td>T_MINFO</td>
<td>Mailbox information</td>
</tr>
<tr>
<td>T_MX</td>
<td>Mail-routing information</td>
</tr>
<tr>
<td>T_UINFO</td>
<td>User (finger command) information</td>
</tr>
<tr>
<td>T_UID</td>
<td>User ID</td>
</tr>
<tr>
<td>T_GID</td>
<td>Group ID</td>
</tr>
</tbody>
</table>

*Answer*  
Points to an address where the response is stored.

*AnswerLength*  
Specifies the size of the answer buffer.

**Return Values**

Upon successful completion, the *res_search* subroutine returns the size of the response. Upon unsuccessful completion, the *res_search* subroutine returns a value of -1 and sets the *h_errno* value to the appropriate error.
Related Information

The `finger` command.

The `dn_comp` subroutine, `dn_expand` subroutine, `getlong` subroutine, `getshort` subroutine, `putlong` subroutine, `putshort` subroutine, `res_init` subroutine, `res_mkquery` subroutine, `res_ninit` subroutine, `res_query` subroutine, `res_send` subroutine.

Sockets Overview and Understanding Domain Name Resolution in *AIX 5L Version 5.3 Communications Programming Concepts*.

---

res_send Subroutine

**Purpose**

Sends a query to a name server and retrieves a response.

**Library**

Standard C Library (`libc.a`)

**Syntax**

```c
#include <sys/types.h>
#include <netinet/in.h>
#include <arpa/nameser.h>
#include <resolv.h>

int res_send(MessagePtr, MessageLength, Answer, AnswerLength)
char *MsgPtr;
int MsgLength;
char *Answer;
int AnswerLength;
```

**Description**

The `res_send` subroutine sends a query to name servers and calls the `res_init` subroutine if the `RES_INIT` option of the `_res` structure is not set. This subroutine sends the query to the local name server and handles time outs and retries.

The `res_send` subroutine is one of a set of subroutines that form the resolver a set of functions that resolve domain names. Global information used by the resolver subroutines is kept in the `_res` structure. The `/usr/include/resolv.h` file contains the `_res` structure definition.

All applications containing the `res_send` subroutine must be compiled with the `_BSD` macro set to a specific value. Acceptable values are 43 and 44. In addition, all socket applications must include the BSD `libbsd.a` library.

**Parameters**

- `MessagePtr` Points to the beginning of a message.
- `MessageLength` Specifies the length of the message.
Return Values
Upon successful completion, the res_send subroutine returns the length of the message.

If the res_send subroutine is unsuccessful, the subroutine returns a -1.

Files
/etc/resolv.conf
Contains general name server and domain name information.

Related Information
The dn_comp subroutine, dn_expand subroutine, getlong subroutine, getshort subroutine, putlong subroutine, putshort subroutine, res_init subroutine, res_mkquery subroutine, res_search subroutine.

Sockets Overview and Understanding Domain Name Resolution in AIX 5L Version 5.3 Communications Programming Concepts.

reexec Subroutine

Purpose
Allows command execution on a remote host.

Library
Standard C Library (libc.a)

Syntax
int rexec (Host, Port, User, Passwd, Command, ErrFileDescParam);
char **Host;
int Port;
char *User, *Passwd,
*Command;
int *ErrFileDescParam;

Description
The rexec subroutine allows the calling process to start commands on a remote host.

If the rexec connection succeeds, a socket in the Internet domain of type SOCK_STREAM is returned to the calling process and is given to the remote command as standard input and standard output.

All applications containing the rexec subroutine must be compiled with the _BSD macro set to a specific value. Acceptable values are 43 and 44. In addition, all socket applications must include the BSD libbsd.a library.
Parameters

**Host**
Contains the name of a remote host that is listed in the [**/etc/hosts**](#) file or [**/etc/resolv.conf**](#) file. If the name of the host is not found in either file, the **rexec** subroutine is unsuccessful.

**Port**
Specifies the well-known DARPA Internet port to use for the connection. A pointer to the structure that contains the necessary port can be obtained by issuing the following library call:

```c
getservbyname("exec","tcp")
```

**User and Passwd**
Points to a user ID and password valid at the host. If these parameters are not supplied, the **rexec** subroutine takes the following actions until finding a user ID and password to send to the remote host:

1. Searches the current environment for the user ID and password on the remote host.
2. Searches the user’s home directory for a file called **$HOME/.netrc** that contains a user ID and password.
3. Prompts the user for a user ID and password.

**Command**
Points to the name of the command to be executed at the remote host.

**ErrFileDescParam**
Specifies one of the following values:

- **Non-zero**
  Indicates an auxiliary channel to a control process is set up, and a descriptor for it is placed in the **ErrFileDescParam** parameter. The control process provides diagnostic output from the remote command on this channel and also accepts bytes as signal numbers to be forwarded to the process group of the command. This diagnostic information does not include remote authorization failure, since this connection is set up after authorization has been verified.

- **0**
  Indicates the standard error of the remote command is the same as standard output, and no provision is made for sending arbitrary signals to the remote process. In this case, however, it may be possible to send out-of-band data to the remote command.

Return Values
Upon successful completion, the system returns a socket to the remote command.

If the **rexec** subroutine is unsuccessful, the system returns a -1 indicating that the specified host name does not exist.

Files

- **/etc/hosts**
  Contains host names and their addresses for hosts in a network. This file is used to resolve a host name into an Internet address.

- **/etc/resolv.conf**
  Contains the name server and domain name.

- **$HOME/.netrc**
  Contains automatic login information.

Related Information

The **getservbyname** subroutine, **rcmd** subroutine, **resvport** subroutine, **ruserok** subroutine.

The **rexecd** daemon.

**Transmission Control Protocol/Internet Protocol** in *Networks and communication management.*
**rexec_af Subroutine**

**Purpose**
Allows command execution on a remote host.

**Syntax**
```c
int rexec_af(char *ahost, unsigned short rport, const char *name, const char *pass, const char *cmd, int *fd2p, int af)
```

**Description**
The `rexec_af` subroutine allows the calling process to start commands on a remote host. It behaves the same as the existing `rexec()` function, but instead of creating only an AF_INET TCP socket, it can also create an AF_INET6 TCP socket.

The `rexec_af` subroutine is useful because the existing `rexec()` function cannot transparently use AF_INET6 sockets. This is because an application would not be prepared to handle AF_INET6 addresses returned by functions such as `getpeername()` on the file descriptor created by `rexec()`.

If the `rexec_af` connection succeeds, a socket in the Internet domain of type `SOCK_STREAM` is returned to the calling process and is given to the remote command as standard input and standard output.

All applications containing the `rexec_af` subroutine must be compiled with the _BSD macro set to a specific value. Acceptable values are 43 and 44. In addition, all socket applications must include the BSD libbsd.a library.

**Parameters**
- **ahost**
  Contains the name of a remote host that is listed in the `/etc/hosts` file or `/etc/resolv.config` file. If the name of the host is not found in either file, the `rexec` subroutine is unsuccessful.

- **rport**
  Specifies the well-known DARPA Internet port to use for the connection. A pointer to the structure that contains the necessary port can be obtained by issuing the following library call: `getservbyname("exec","tcp")`

- **name and pass**
  Points to a valid user ID and password at the host. If these parameters are not supplied, the `rexec_af` subroutine takes the following actions until it finds a user ID and password to send to the remote host:
  1. Searches the current environment for the user ID and password on the remote host.
  2. Searches the user’s home directory for a file called `$HOME/.netrc` that contains a user ID and password.
  3. Prompts the user for a user ID and password.

- **cmd**
  Points to the name of the command to be executed at the remote host.
Specifies one of the following values:

**Non-zero**
Indicates that an auxiliary channel to a control process is set up, and a descriptor for it is placed in the `fd2p` parameter. The control process provides diagnostic output from the remote command on this channel and also accepts bytes as signal numbers to be forwarded to the process group of the command. This diagnostic information does not include remote authorization failure, since this connection is set up after authorization has been verified.

**0**
Indicates that the standard error of the remote command is the same as standard output, and no provision is made for sending arbitrary signals to the remote process. In this case, however, it might be possible to send out-of-band data to the remote command.

**af**
The family argument is AF_INET, AF_INET6, or AF_UNSPEC. When either AF_INET or AF_INET6 is specified, this subroutine will create a socket of the specified address family. When AF_UNSPEC is specified, it will try all possible address families until a connection can be established, and will return the associated socket of the connection.

**Return Values**
Upon successful completion, the system returns a socket to the remote command. If the `rexec_af` subroutine is unsuccessful, the system returns a –1, indicating that the specified host name does not exist.

**Files**
- `/etc/hosts`
  Contains host names and their addresses for hosts in a network. This file is used to resolve a host name into an Internet address.
- `/etc/resolv.conf`
  Contains the name server and domain name.
- `~/.netrc`
  Contains automatic login information.

**Related Information**
- "rcmd_af Subroutine" on page 153, "rexec Subroutine" on page 172, "rresvport_af Subroutine" on page 176.

---

**rresvport Subroutine**

**Purpose**
Retrieves a socket with a privileged address.

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

**Syntax**
```c
int rresvport (Port)
int *Port;
```

**Description**
The `rresvport` subroutine obtains a socket with a privileged address bound to the socket. A privileged Internet port is one that falls in a range between 0 and 1023.

Only processes with an effective user ID of root user can use the `rresvport` subroutine. An authentication scheme based on remote port numbers is used to verify permissions.
If the connection succeeds, a socket in the Internet domain of type `SOCK_STREAM` is returned to the calling process.

All applications containing the `rresvport` subroutine must be compiled with the `_BSD` macro set to a specific value. Acceptable values are 43 and 44. In addition, all socket applications must include the BSD `libbsd.a` library.

**Parameters**

*Port*  
Specifies the port to use for the connection.

**Return Values**

Upon successful completion, the `rresvport` subroutine returns a valid, bound socket descriptor.

If the `rresvport` subroutine is unsuccessful, the subroutine handler performs the following functions:

- Returns a value of -1 to the calling program.
- Moves an error code, indicating the specific error, into the `errno` global variable.

**Error Codes**

The `rresvport` subroutine is unsuccessful if any of the following errors occurs:

<table>
<thead>
<tr>
<th>Error</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EAGAIN</td>
<td>All network ports are in use.</td>
</tr>
<tr>
<td>EAFNOSUPPORT</td>
<td>The addresses in the specified address family cannot be used with this socket.</td>
</tr>
<tr>
<td>EMFILE</td>
<td>Two hundred file descriptors are currently open.</td>
</tr>
<tr>
<td>ENFILE</td>
<td>The system file table is full.</td>
</tr>
<tr>
<td>ENOBUFS</td>
<td>Insufficient buffers are available in the system to complete the subroutine.</td>
</tr>
</tbody>
</table>

**Files**

`/etc/services`  
Contains the service names.

**Related Information**

The `rcmd` subroutine, `ruserok` subroutine.

Sockets Overview in *AIX 5L Version 5.3 Communications Programming Concepts*.

---

**rresvport_af Subroutine**

**Purpose**

Retrieves a socket with a privileged address.

**Syntax**

```c
int rresvport_af(int *port, int family);
```

**Description**

The `rresvport_af` subroutine obtains a socket with a privileged address bound to the socket. A privileged Internet port is one that falls in a range between 0 and 1023.
This subroutine is similar to the existing rresvport() subroutine, except that rresvport_af also takes and address family as an argument. This function is capable of creating either an AF_INET/TCP or an AF_INET6/TCP socket.

Only processes with an effective user ID of root user can use the rresvport subroutine. An authentication scheme based on remote port numbers is used to verify permissions.

If the connection succeeds, a socket in the Internet domain of type [SOCK_STREAM] is returned to the calling process.

All applications containing the rresvport subroutine must be compiled with the _BSD macro set to a specific value. Acceptable values are 43 and 44. In addition, all socket applications must include the BSD libbsd.a library.

Parameters

- `port` Specifies the port to use for the connection.
- `family` Specifies either AF_INET or AF_INET6 to accommodate the appropriate version.

Return Values

Upon successful completion, the rresvport_af subroutine returns a valid, bound socket descriptor.

If the rresvport_af subroutine is unsuccessful, the subroutine handler performs the following functions:

- Returns a value of -1 to the calling program.
- Moves an error code, indicating the specific error, into the errno global variable.

Error Codes

- **EAFNOSUPPORT** The address family is not supported.
- **EAGAIN** All network ports are in use.
- **EMFILE** Two hundred file descriptors are currently open.
- **ENFILE** The system file table is full.
- **ENOBUFFS** Insufficient buffers are available in the system to complete the subroutine.

Files

- `/etc/services` Contains the service names.

Related Information

- “rcmd_af Subroutine” on page 153
- “rexec_af Subroutine” on page 174
- “rresvport Subroutine” on page 175

ruserok Subroutine

Purpose

Allows servers to authenticate clients.

Library

Standard C Library (libc.a)
Syntax

```c
int ruserok (Host, RootUser, RemoteUser, LocalUser)
char **Host;
int RootUser;
char *RemoteUser,
*LocalUser;
```

Description

The `ruserok` subroutine allows servers to authenticate clients requesting services.

Always specify the host name. If the local domain and remote domain are the same, specifying the domain parts is optional. To determine the domain of the host, use the `gethostname` subroutine.

All applications containing the `ruserok` subroutine must be compiled with the `_BSD` macro set to a specific value. Acceptable values are 43 and 44. In addition, all socket applications must include the BSD `libbsd.a` library.

Parameters

**Host**
Specifies the name of a remote host. The `ruserok` subroutine checks for this host in the `/etc/hosts.equiv` file. Then, if necessary, the subroutine checks a file in the user's home directory at the server called `/$HOME/rhosts` for a host and remote user ID.

**RootUser**
Specifies a value to indicate whether the effective user ID of the calling process is a root user. A value of 0 indicates the process does not have a root user ID. A value of 1 indicates that the process has local root user privileges, and the `/etc/hosts.equiv` file is not checked.

**RemoteUser**
Points to a user name that is valid at the remote host. Any valid user name can be specified.

**LocalUser**
Points to a user name that is valid at the local host. Any valid user name can be specified.

Return Values

The `ruserok` subroutine returns a 0, if the subroutine successfully locates the name specified by the `Host` parameter in the `/etc/hosts.equiv` file or the IDs specified by the `Host` and `RemoteUser` parameters are found in the `/$HOME/rhosts` file.

If the name specified by the `Host` parameter was not found, the `ruserok` subroutine returns a -1.

Files

- `/etc/services`
  Contains service names.
- `/etc/hosts.equiv`
  Specifies foreign host names.
- `/$HOME/rhosts`
  Specifies the remote users of a local user account.

Related Information

The `rlogind` command, `rshd` command.

The `gethostname` subroutine, `rcmd` subroutine, `rresvport` subroutine, `sethostname` subroutine.

[Sockets Overview](#) in *AIX 5L Version 5.3 Communications Programming Concepts*. 
**sctp_opt_info Subroutine**

**Purpose**
Passes information both into and out of SCTP stack.

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

**Syntax**
```c
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <sctpapi.h>

int sctp_opt_info(int sd, sctp_assoc_t id, int opt, void *arg_size, size_t *size);
```

**Description**
Applications use the `sctp_opt_info` subroutine to get information about various SCTP socket options from the stack. For the sockets with multiple associations, the association ID can be specified to apply the operation on any particular association of a socket. Because an SCTP association supports multihoming, this operation can be used to specify any particular peer address using a `sockaddr_storage` structure. In this case, the result of the operation will be applied to only that particular peer address.

**Implementation Specifics**
The `sctp_opt_info` subroutine is part of Base Operating System (BOS) Runtime.

**Parameters**
- **sd**: Specifies the UDP style socket descriptor returned from the `socket` system call.
- **id**: Specifies the identifier of the association to query.
- **opt**: Specifies the socket option to get.
- **arg_size**: Specifies an option specific structure buffer provided by the caller.
- **size**: Specifies the size of the option returned.

**Return Values**
Upon successful completion, the `sctp_opt_info` subroutine returns 0.

If the `sctp_opt_info` subroutine is unsuccessful, the subroutine handler returns a value of -1 to the calling program and sets `errno` to the appropriate error code.

**Error Codes**
The `sctp_opt_info` subroutine is unsuccessful if any of the following errors occurs:

- **EFAULT**: Indicates that the user has insufficient authority to access the data, or the address specified in the `uaddr` parameter is not valid.
- **EIO**: Indicates that a permanent I/O error occurred while referencing data.
- **ENOMEM**: Indicates insufficient memory for the required paging operation.
Related Information

The "sctpctrl Subroutine" on page 181, "sctp_peeloff Subroutine."

Stream Control Transmission Protocol in Networks and communication management.

sctp_peeloff Subroutine

Purpose
Branches off an association into a separate socket.

Library
Standard C Library (libc.a)

Syntax

```c
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <sctpapi.h>

int sctp_peeloff(int sd, sctp_assoc_t *assoc_id);
int sd;
sctp_assoc_t *assoc_id;
```

Description
An application uses the sctp_peeloff subroutine when it wants to branch-off an existing association into a separate socket/file descriptor. It returns a new socket descriptor, which in turn can be used to send and receive subsequent SCTP packets. After it has been branched off, an association becomes completely independent of the original socket. Any subsequent data or control operations to that association must be passed using the new socket descriptor. Also, a close on the original socket descriptor will not close the new socket descriptor branched out of the association.

All the associations under the same socket share the same socket buffer space of the socket that they belong to. If an association gets branched off to a new socket using sctp_peeloff, then it inherits the socket buffer space associated with the new socket descriptor. This way, the association that got peeled off keeps more buffer space.

Implementation Specifics
The sctp_peeloff subroutine is part of Base Operating System (BOS) Runtime.

Parameters

- `sd` Specifies the UDP style socket descriptor returned from the socket system call.
- `assoc_id` Specifies the identifier of the association that is to be branched-off to a separate socket descriptor.
Return Values
Upon successful completion, the `sctp_peeloff` subroutine returns the nonnegative socket descriptor of the branched-off socket.

If the `sctp_peeloff` subroutine is unsuccessful, the subroutine handler returns a value of -1 to the calling program and moves an error code to the `errno` global variable.

Error Codes
The `sctp_peeloff` subroutine is unsuccessful if any of the following errors occurs:

<table>
<thead>
<tr>
<th>Error Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EINVAL</td>
<td>Invalid argument.</td>
</tr>
<tr>
<td>EBADF</td>
<td>Bad file descriptor.</td>
</tr>
<tr>
<td>EAFNOSUPPORT</td>
<td>The addresses in the specified address family cannot be used with this socket.</td>
</tr>
<tr>
<td>ESOCKTNOSUPPORT</td>
<td>The socket in the specified address family is not supported.</td>
</tr>
<tr>
<td>EMFILE</td>
<td>The per-process descriptor table is full.</td>
</tr>
<tr>
<td>ENOBUFFS</td>
<td>Insufficient resources were available in the system to complete the call.</td>
</tr>
<tr>
<td>ECONNABORTED</td>
<td>The client aborted the connection.</td>
</tr>
</tbody>
</table>

Related Information
"sctpctrl Subroutine," "sctp_opt_info Subroutine" on page 179

Stream Control Transmission Protocol in Networks and communication management.

sctpctrl Subroutine

Purpose
Controls and configures SCTP.

Syntax
```
sctpctrl { load | unload | set }
sctpctrl stats [ reset ] [ interval ]
sctpctrl set { name=value | default [ name ] }
sctpctrl get [ name ]
```

Description
The `sctpctrl` subroutine controls and configures the SCTP kernel extension. This subroutine can be used to load and unload the SCTP kernel extension. It can also be used to dump SCTP data, and set and retrieve various SCTP tunables. In addition, the `sctpctrl` subroutine can be used to read and reset the SCTP specific network statistics.

Parameters

- `load` Loads the SCTP kernel extension if not loaded.
- `unload` Unloads the SCTP kernel extension if loaded.
- `stats [ reset ] [ interval ]` Displays SCTP statistics. The optional `reset` command will clear (0) the statistics. If the `interval` parameter (in seconds) is added, the program does not exit; instead, it outputs the statistics every `interval` seconds.
set {name=value|default [name]}  Sets the SCTP tunable to a value. If default is specified, all the tunables are set to their default values. If optional [name] is specified followed by default, the tunable described by name is set to its default value.

get [name]  Gets the value of the tunable described by its optional name parameter. If the name parameter is not specified, get gets the values of all the tunables.

Examples
1. To load the sctp kernel extension, enter:
   sctpctrl load
2. To unload the sctp kernel extension, enter:
   sctpctrl unload
3. To reset the SCTP statistics, enter:
   sctpctrl stats reset
   This command will zero-out all the SCTP statistics.
4. To get the values of the SCTP tunable, enter:
   sctpctrl get
   This will list all the SCTP tunables and their values. Here is a sample output.
   
   sctp_assoc_maxerr = 10
   sctp_cookie_life = 60
   sctp_delack_timer = 4
   sctp_dontdelayack = 1
   sctp_ecn = 1
   sctp_ephemeral_high = 65535
   sctp_ephemeral_low = 32768
   sctp_instreams = 2048
   sctp_maxburst = 8
   sctp_outstreams = 10
   sctp_path_maxerr = 5
   sctp_pmtu_discover = 1
   sctp_rttmax = 60
   sctp_rttmin = 1
   sctp_recvspace = 65536
   sctp_sendspace = 65536
   sctp_send_fewacks = 0

5. To set sctp_path_maxerr to a value of 6, enter:
   sctpctrl set sctp_path_maxerr=6

Files
/usr/sbin/sctpctrl  Contains the sctpctrl command.
/usr/lib/drivers/sctp  Contains the SCTP kernel extension.

Related Information
The "sctp_peeloff Subroutine" on page 180,"sctp_opt_info Subroutine" on page 179.

Stream Control Transmission Protocol in Networks and communication management.

send Subroutine

Purpose
Sends messages from a connected socket.
Library
Standard C Library (libc.a)

Syntax

```c
#include <sys/types.h>
#include <sys/socketvar.h>
#include <sys/socket.h>

int send (Socket, Message, Length, Flags)
```

**Description**

The `send` subroutine sends a message only when the socket is connected. This subroutine on a socket is not thread safe. The `sendto` and `sendmsg` subroutines can be used with unconnected or connected sockets.

To broadcast on a socket, first issue a `setsockopt` subroutine using the `SO_BROADCAST` option to gain broadcast permissions.

Specify the length of the message with the `Length` parameter. If the message is too long to pass through the underlying protocol, the system returns an error and does not transmit the message.

No indication of failure to deliver is implied in a `send` subroutine. A return value of -1 indicates some locally detected errors.

If no space for messages is available at the sending socket to hold the message to be transmitted, the `send` subroutine blocks unless the socket is in a nonblocking I/O mode. Use the `select` subroutine to determine when it is possible to send more data.

The socket applications can be compiled with `COMPAT_43` defined. This will make the `sockaddr` structure BSD 4.3 compatible. For more details refer to `socket.h`.

**Parameters**

- **Socket**: Specifies the unique name for the socket.
- **Message**: Points to the address of the message to send.
- **Length**: Specifies the length of the message in bytes.
- **Flags**: Allows the sender to control the transmission of the message. The `Flags` parameter used to send a call is formed by logically ORing one or both of the values shown in the following list:
  - **MSG_OOB**: Processes out-of-band data on sockets that support `SOCK_STREAM` communication.
  - **MSG_DONTROUTE**: Sends without using routing tables.
  - **MSG_MPEG2**: Indicates that this block is a MPEG2 block. This flag is valid `SOCK_CONN_DGRAM` types of sockets only.
Return Values
Upon successful completion, the send subroutine returns the number of characters sent.

If the send subroutine is unsuccessful, the subroutine handler performs the following functions:
- Returns a value of -1 to the calling program.
- Moves an error code, indicating the specific error, into the errno global variable.

Error Codes
The subroutine is unsuccessful if any of the following errors occurs:

<table>
<thead>
<tr>
<th>Error</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EACCES</td>
<td>Write access to the named socket is denied, or the socket trying to send a broadcast packet does not have broadcast capability.</td>
</tr>
<tr>
<td>EADDRNOTAVAIL</td>
<td>The specified address is not a valid address.</td>
</tr>
<tr>
<td>EAFNOSUPPORT</td>
<td>The specified address is not a valid address for the address family of this socket.</td>
</tr>
<tr>
<td>EBADF</td>
<td>The Socket parameter is not valid.</td>
</tr>
<tr>
<td>ECONNRESET</td>
<td>A connection was forcibly closed by a peer.</td>
</tr>
<tr>
<td>EDESTADDRREQ</td>
<td>The socket is not in connection-mode and no peer address is set.</td>
</tr>
<tr>
<td>EFAULT</td>
<td>The Address parameter is not in a writable part of the user address space.</td>
</tr>
<tr>
<td>EHOSTUNREACH</td>
<td>The destination host cannot be reached.</td>
</tr>
<tr>
<td>EINVAL</td>
<td>The Length parameter is invalid.</td>
</tr>
<tr>
<td>EINTR</td>
<td>A signal interrupted send before any data was transmitted.</td>
</tr>
<tr>
<td>ENOBUFS</td>
<td>Insufficient resources were available in the system to perform the operation.</td>
</tr>
<tr>
<td>ENOENT</td>
<td>The path name does not name an existing file, or the path name is an empty string.</td>
</tr>
<tr>
<td>ENOMEM</td>
<td>The available data space in memory is not large enough to hold group/ACL information.</td>
</tr>
<tr>
<td>ENOTSOCK</td>
<td>The Socket parameter refers to a file, not a socket.</td>
</tr>
<tr>
<td>EOPNOTSUPP</td>
<td>The socket argument is associated with a socket that does not support one or more of the values set in Flags.</td>
</tr>
<tr>
<td>EPIPE</td>
<td>An attempt was made to send on a socket that was connected, but the connection has been shut down either by the remote peer or by this side of the connection. If the socket is of type SOCK_STREAM, the SIGPIPE signal is generated to the calling process.</td>
</tr>
<tr>
<td>EWOULDBLOCK</td>
<td>The socket is marked nonblocking, and no connections are present to be accepted.</td>
</tr>
</tbody>
</table>

Related Information
The connect subroutine, getsockopt subroutine, recv subroutine, recvfrom subroutine, recvmsg subroutine, select subroutine, sendmsg subroutine, sendto subroutine, setsockopt subroutine, shutdown subroutine, socket subroutine.

Sockets Overview and Understanding Socket Data Transfer in AIX 5L Version 5.3 Communications Programming Concepts.

sendmsg Subroutine

Purpose
Sends a message from a socket using a message structure.

Library
Standard C Library (libc.a)
Syntax

```c
#include <sys/types.h>
#include <sys/socketvar.h>
#include <sys/socket.h>

int sendmsg (Socket, Message, Flags);

Socket; const struct msghdr Message []; int Flags;
```

Description

The `sendmsg` subroutine sends messages through connected or unconnected sockets using the `msghdr` message structure. The `/usr/include/sys/socket.h` file contains the `msghdr` structure and defines the structure members. In BSD 4.4, the size and members of the `msghdr` message structure have been modified. Applications wanting to start the old structure need to compile with `COMPAT_43` defined. The default behaviour is that of BSD 4.4.

To broadcast on a socket, the application program must first issue a `setsockopt` subroutine using the `SO_BROADCAST` option to gain broadcast permissions.

The `sendmsg` subroutine supports only 15 message elements.

All applications containing the `sendmsg` subroutine must be compiled with the `_BSD` macro set to a specific value. Acceptable values are 43 and 44. In addition, all socket applications must include the BSD `libbsd.a` library.

The `sendmsg` routine supports IPv6 ancillary data elements as defined in the Advanced Sockets API for IPv6.

Parameters

- **Socket**: Specifies the socket descriptor.
- **Message**: Points to the `msghdr` message structure containing the message to be sent.
- **Flags**: Allows the sender to control the message transmission. The `sys/socket.h` file contains the `Flags` parameter. The `Flags` parameter used to send a call is formed by logically ORing one or both of the following values:
  - **MSG_OOB**: Processes out-of-band data on sockets that support `SOCK_STREAM`.
  - **MSG_DONTROUTE**: Sends without using routing tables.
  - **MSG_MPEG2**: Indicates that this block is a MPEG2 block. It only applies to `SOCK_CONN_DGRAM` types of sockets only.

Note: The following value is not for general use. It is an administrative tool used for debugging or for routing programs.

Return Values

Upon successful completion, the `sendmsg` subroutine returns the number of characters sent.

If the `sendmsg` subroutine is unsuccessful, the system handler performs the following functions:
- Returns a value of -1 to the calling program.
- Moves an error code, indicating the specific error, into the `errno` global variable.
Error Codes

The sendmsg subroutine is unsuccessful if any of the following errors occurs:

<table>
<thead>
<tr>
<th>Error</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EACCES</td>
<td>Write access to the named socket is denied, or the socket trying to send a broadcast packet does not have broadcast capability.</td>
</tr>
<tr>
<td>EADDRNOTAVAIL</td>
<td>The specified address is not a valid address.</td>
</tr>
<tr>
<td>EAFNOSUPPORT</td>
<td>The specified address is not a valid address for the address family of this socket.</td>
</tr>
<tr>
<td>EBADF</td>
<td>The Socket parameter is not valid.</td>
</tr>
<tr>
<td>ECONNRESET</td>
<td>A connection was forcibly closed by a peer.</td>
</tr>
<tr>
<td>EDESTADDRREQ</td>
<td>The socket is not in connection-mode and does not have its peer address set.</td>
</tr>
<tr>
<td>EFAULT</td>
<td>The Address parameter is not in a writable part of the user address space.</td>
</tr>
<tr>
<td>EHOSTUNREACH</td>
<td>The destination host cannot be reached.</td>
</tr>
<tr>
<td>EINTR</td>
<td>A signal interrupted sendmsg before any data was transmitted.</td>
</tr>
<tr>
<td>EINVAL</td>
<td>The length of the msghdr structure is invalid.</td>
</tr>
<tr>
<td>EMSGSIZE</td>
<td>The message is too large to be sent all at once (as the socket requires), or the msg_iovlen member of the msghdr structure pointed to by Message is less than or equal to 0 or is greater than IOV_MAX.</td>
</tr>
<tr>
<td>ENOENT</td>
<td>The path name does not name an existing file, or the path name is an empty string.</td>
</tr>
<tr>
<td>ENETUNREACH</td>
<td>The destination network is not reachable.</td>
</tr>
<tr>
<td>ENOBUFS</td>
<td>The system ran out of memory for an internal data structure.</td>
</tr>
<tr>
<td>ENOMEM</td>
<td>The available data space in memory is not large enough to hold group/ACL information.</td>
</tr>
<tr>
<td>ENOPROTOOPT</td>
<td>The protocol is not 64-bit supported.</td>
</tr>
<tr>
<td>ENOTCONN</td>
<td>The socket is in connection-mode but is not connected.</td>
</tr>
<tr>
<td>ENOTSOCK</td>
<td>The Socket parameter refers to a file, not a socket.</td>
</tr>
<tr>
<td>EOPNOTSUPP</td>
<td>The socket argument is associated with a socket that does not support one or more of the values set in flags.</td>
</tr>
<tr>
<td>EPIPE</td>
<td>An attempt was made to send on a socket that was connected, but the connection has been shut down either by the remote peer or by this side of the connection. If the socket is of type SOCK_STREAM, the SIGPIPE signal is generated to the calling process.</td>
</tr>
<tr>
<td>EWOULDBLOCK</td>
<td>The socket is marked nonblocking, and no connections are present to be accepted.</td>
</tr>
</tbody>
</table>

Related Information

The no command.

The connect subroutine, getsockopt subroutine, recv subroutine, recvfrom subroutine, recvmsg subroutine, select subroutine, send subroutine, sendto subroutine, setsockopt subroutine, shutdown subroutine.

Sockets Overview and Understanding Socket Data Transfer in AIX 5L Version 5.3 Communications Programming Concepts.

sendto Subroutine

Purpose

Sends messages through a socket.

Library

Standard C Library (libc.a)

Syntax

#include <sys/socket.h>
The `sendto` subroutine allows an application program to send messages through an unconnected socket by specifying a destination address.

To broadcast on a socket, first issue a `setsockopt` subroutine using the `SO_BROADCAST` option to gain broadcast permissions.

Provide the address of the target using the `To` parameter. Specify the length of the message with the `Length` parameter. If the message is too long to pass through the underlying protocol, the error `EMSGSIZE` is returned and the message is not transmitted.

If the `sending` socket has no space to hold the message to be transmitted, the `sendto` subroutine blocks the message unless the socket is in a nonblocking I/O mode.

Use the `select` subroutine to determine when it is possible to send more data.

The socket applications can be compiled with `COMPAT_43` defined. This will make the `sockaddr` structure BSD 4.3 compatible. For more details refer to `socket.h`.

### Parameters

- **Socket**
  - Specifies the unique name for the socket.

- **Message**
  - Specifies the address containing the message to be sent.

- **Length**
  - Specifies the size of the message in bytes.

- **Flags**
  - Allows the sender to control the message transmission. The `Flags` parameter used to send a call is formed by logically ORing one or both of the following values:

    - `MSG_OOB`
      - Processes out-of-band data on sockets that support `SO_STREAM`.

    - `MSG_DONTROUTE`
      - Sends without using routing tables.

    The `/usr/include/sys/socket.h` file defines the `Flags` parameter.

- **To**
  - Specifies the destination address for the message. The destination address is a `sockaddr` structure defined in the `/usr/include/sys/socket.h` file.

- **ToLength**
  - Specifies the size of the destination address.

### Return Values

Upon successful completion, the `sendto` subroutine returns the number of characters sent.

If the `sendto` subroutine is unsuccessful, the system returns a value of -1, and the `errno` global variable is set to indicate the error.
Error Codes
The subroutine is unsuccessful if any of the following errors occurs:

<table>
<thead>
<tr>
<th>Error</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EACCES</td>
<td>Write access to the named socket is denied, or the socket trying to send a broadcast packet does not have broadcast capability.</td>
</tr>
<tr>
<td>EADDRNOTAVAIL</td>
<td>The specified address is not a valid address.</td>
</tr>
<tr>
<td>EAFNOSUPPORT</td>
<td>The specified address is not a valid address for the address family of this socket.</td>
</tr>
<tr>
<td>EBADF</td>
<td>The [Socket] parameter is not valid.</td>
</tr>
<tr>
<td>ECONNRESET</td>
<td>A connection was forcibly closed by a peer.</td>
</tr>
<tr>
<td>EDESTADDRREQ</td>
<td>The socket is not in connection-mode and no peer address is set.</td>
</tr>
<tr>
<td>EFAULT</td>
<td>The Address parameter is not in a writable part of the user address space.</td>
</tr>
<tr>
<td>EHOSTUNREACH</td>
<td>The destination host cannot be reached.</td>
</tr>
<tr>
<td>EINVAL</td>
<td>A signal interrupted sendto before any data was transmitted.</td>
</tr>
<tr>
<td>EINVAL</td>
<td>The Length or ToLength parameter is invalid.</td>
</tr>
<tr>
<td>EISCONN</td>
<td>A SOCK_DGRAM socket is already connected.</td>
</tr>
<tr>
<td>EMSGSIZE</td>
<td>The message is too large to be sent all at once as the socket requires.</td>
</tr>
<tr>
<td>ENETUNREACH</td>
<td>The destination network is not reachable.</td>
</tr>
<tr>
<td>ENOBUFS</td>
<td>The system ran out of memory for an internal data structure.</td>
</tr>
<tr>
<td>ENOENT</td>
<td>The path name does not name an existing file, or the path name is an empty string.</td>
</tr>
<tr>
<td>ENOMEM</td>
<td>The available data space in memory is not large enough to hold group/ACL information.</td>
</tr>
<tr>
<td>ENOPROTOOPT</td>
<td>The protocol is not 64-bit supported.</td>
</tr>
<tr>
<td>ENOTCONN</td>
<td>The socket is in connection-mode but is not connected.</td>
</tr>
<tr>
<td>ENOTSOCK</td>
<td>The [Socket] parameter refers to a file, not a socket.</td>
</tr>
<tr>
<td>EOPNOTSUPP</td>
<td>The socket argument is associated with a socket that does not support one or more of the values set in Flags.</td>
</tr>
<tr>
<td>EPIPE</td>
<td>An attempt was made to send on a socket that was connected, but the connection has been shut down either by the remote peer or by this side of the connection. If the socket is of type SOCK_STREAM, the SIGPIPE signal is generated to the calling process.</td>
</tr>
<tr>
<td>EWOULDBLOCK</td>
<td>The socket is marked nonblocking, and no connections are present to be accepted.</td>
</tr>
</tbody>
</table>

Related Information

Sending UNIX Datagrams Example Program  Sending Internet Datagrams Example Program  Sockets Overview  Understanding Socket Data Transfer in AIX 5L Version 5.3 Communications Programming Concepts.

send_file Subroutine

Purpose
Sends the contents of a file through a socket.

Library
Standard C Library (libc.a)

Syntax
#include <sys/socket.h>

ssize_t send_file(Socket_p, sf_iobuf, flags)
int *Socket_p;
struct sf_parms *sf_iobuf;
uint_t flags;

Description
The send_file subroutine sends data from the opened file specified in the sf_iobuf parameter, over the connected socket pointed to by the Socket_p parameter.

Note: Currently, the send_file only supports the TCP/IP protocol (SOCK_STREAM socket in AF_INET). An error will be returned when this function is used on any other types of sockets.

Parameters

Socket_p Points to the socket descriptor of the socket which the file will be sent to.
Note: This is different from most of the socket functions.


sf_iobuf  Points to a sf_parms structure defined as follows:

/*
 * Structure for the send_file system call
 */
#ifdef __64BIT__
#define SF_INT64(x)          int64_t x;
#define SF_UINT64(x)         uint64_t x;
#else
#ifdef __LONG_LONG__
#define SF_INT64(x)          int64_t x;
#define SF_UINT64(x)         uint64_t x;
#else
#define SF_INT64(x)          int filler_##x; int x;
#define SF_UINT64(x)         int filler_##x; uint_t x;
#endif
#endif

struct sf_parms {
    /* ----------- header parms ----------- */
    void    *header_data;       /* Input/Output. Points to header buf */
    uint_t   header_length;     /* Input/Output. Length of the header */
    /* ----------- file parms ----------- */
    int file_descriptor;        /* Input. File descriptor of the file */
    SF_UINT64(file_size);       /* Output. Size of the file */
    SF_UINT64(file_offset);     /* Input/Output. Starting offset */
    SF_INT64(file_bytes);       /* Input/Output. number of bytes to send */
    /* ----------- trailer parms ----------- */
    void    *trailer_data;      /* Input/Output. Points to trailer buf */
    uint_t   trailer_length;    /* Input/Output. Length of the trailer */
    /* ----------- return info ----------- */
    SF_UINT64(bytes_sent);      /* Output. number of bytes sent */
};

header_data
Points to a buffer that contains header data which is to be sent before the file data. May be a
NULL pointer if header_length is 0. This field will be updated by send_file when header is
transmitted - that is, header_data + number of bytes of the header sent.

header_length
Specifies the number of bytes in the header_data. This field must be set to 0 to indicate that
header data is not to be sent. This field will be updated by send_file when header is transmitted
- that is, header_length - number of bytes of the header sent.

file_descriptor
Specifies the file descriptor for a file that has been opened and is readable. This is the
descriptor for the file that contains the data to be transmitted. The file_descriptor is ignored
when file_bytes = 0. This field is not updated by send_file.

file_size
Contains the byte size of the file specified by file_descriptor. This field is filled in by the kernel.

file_offset
Specifies the byte offset into the file from which to start sending data. This field is updated by
the send_file when file data is transmitted - that is, file_offset + number of bytes of the file
data sent.
file_bytes
Specifications the number of bytes from the file to be transmitted. Setting file_bytes to −1 transmits the entire file from the file_offset. When this field is not set to -1, it is updated by send_file when file data is transmitted - that is, file_bytes - number of bytes of the file data sent.

trailer_data
Points to a buffer that contains trailer data which is to be sent after the file data. May be a NULL pointer if trailer_length is 0. This field will be updated by send_file when trailer is transmitted - that is, trailer_data + number of bytes of the trailer sent.

trailer_length
Specifies the number of bytes in the trailer_data. This field must be set to 0 to indicate that trailer data is not to be sent. This field will be updated by send_file when trailer is transmitted - that is, trailer_length - number of bytes of the trailer sent.

bytes_sent
Contains number of bytes that were actually sent in this call to send_file. This field is filled in by the kernel.

All fields marked with Input in the sf_parms structure requires setup by an application prior to the send_file calls. All fields marked with Output in the sf_parms structure adjusts by send_file when it successfully transmitted data, that is, either the specified data transmission is partially or completely done.

The send_file subroutine attempts to write header_length bytes from the buffer pointed to by header_data, followed by file_bytes from the file associated with file_descriptor, followed by trailer_length bytes from the buffer pointed to by trailer_data, over the connection associated with the socket pointed to by Socket_p.

As the data is sent, the kernel updates the parameters pointed by sf_iobuf so that if the send_file has to be called multiple times (either due to interruptions by signals, or due to non-blocking I/O mode) in order to complete a file data transmission, the application can reissue the send_file command without setting or re-adjusting the parameters over and over again.

If the application sets file_offset greater than the actual file size, or file_bytes greater than (the actual file size - file_offset), the return value will be -1 with errno EINVAL.
flags

Specifies the following attributes:

**SF_CLOSE**
Closes the socket pointed to by *Socket_p* after the data has been successfully sent or queued for transmission.

**SF_REUSE**
Prepares the socket for reuse after the data has been successfully sent or queued for transmission and the existing connection closed.

*Note:* This option is currently not supported on this operating system.

**SF_DONT_CACHE**
Does not put the specified file in the Network Buffer Cache.

**SF_SYNC_CACHE**
Verifies/Updates the Network Buffer Cache for the specified file before transmission.

When the **SF_CLOSE** flag is set, the connected socket specified by *Socket_p* will be disconnected and closed by *send_file* after the requested transmission has been successfully done. The socket descriptor pointed to by *Socket_p* will be set to -1. This flag won’t take effect if *send_file* returns non-0.

The flag **SF_REUSE** currently is not supported by AIX. When this flag is specified, the socket pointed by *Socket_p* will be closed and returned as -1. A new socket needs to be created for the next connection.

*send_file* will take advantage of a Network Buffer Cache in kernel memory to dynamically cache the output file data. This will help to improve the *send_file* performance for files which are:

1. accessed repetitively through network and
2. not changed frequently.

Applications can exclude the specified file from being cached by using the **SF_DONT_CACHE** flag.

*send_file* will update the cache every so often to make sure that the file data in cache is valid for a certain time period. The network option parameter “send_file_duration” controlled by the **no** command can be modified to configure the interval of the *send_file* cache validation, the default is 300 (in seconds). Applications can use the **SF_SYNC_CACHE** flag to ensure that a cache validation of the specified file will occur before the file is sent by *send_file*, regardless the value of the “send_file_duration”. Other Network Buffer Cache related parameters are “nbc_limit”, “nbc_max_cache”, and “nbc_min_cache”. For additional information, see the **no** command.

**Return Value**

There are three possible return values from *send_file*:

<table>
<thead>
<tr>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>-1</td>
<td>an error has occurred, errno contains the error code.</td>
</tr>
<tr>
<td>0</td>
<td>the command has completed successfully.</td>
</tr>
<tr>
<td>1</td>
<td>the command was completed partially, some data has been transmitted but the command has to return for some reason, for example, the command was interrupted by signals.</td>
</tr>
</tbody>
</table>

The fields marked with *Output* in the *sf_parms* structure (pointed to by *sf_iobuf*) is updated by *send_file* when the return value is either 0 or 1. The *bytes_sent* field contains the total number of bytes that were sent in this call. It is always true that *bytes_sent* \(\leq\) *header_length*(Input) + *file_bytes*(Input) + *trailer_length* (Input).

The *send_file* supports the blocking I/O mode and the non-blocking I/O mode. In the blocking I/O mode, *send_file* blocks until all file data (plus the header and the trailer) is sent. It adjusts the *sf_iobuf* to reflect the transmission results, and return 0. It is possible that *send_file* can be interrupted before the request is fully done, in that case, it adjusts the *sf_iobuf* to reflect the transmission progress, and return 1.

In the non-blocking I/O mode, the *send_file* transmits as much as the socket space allows, adjusts the *sf_iobuf* to reflect the transmission progress, and returns either 0 or 1. When there is no socket space in
the system to buffer any of the data, the `send_file` returns -1 and sets errno to EWOULDBLOCK. `select` or `poll` can be used to determine when it is possible to send more data.

**Possible errno returned:**

- **EBADF**: Either the socket or the file descriptor parameter is not valid.
- **ENOTSOCK**: The socket parameter refers to a file, not a socket.
- **EPROTONOSUPPORT**: Protocol not supported.
- **EFAULT**: The addresses specified in the HeaderTailor parameter is not in a writable part of the user-address space.
- **EINTR**: The operation was interrupted by a signal before any data was sent. (If some data was sent, `send_file` returns the number of bytes sent before the signal, and EINTR is not set).
- **EINVAL**: The offset, length of the HeaderTrailer, or flags parameter is invalid.
- **ENOTCONN**: A `send_file` on a socket that is not connected, a `send_file` on a socket that has not completed the connect sequence with its peer, or is no longer connected to its peer.
- **EWOULDBLOCK**: The socket is marked non-blocking and the requested operation would block.
- **ENOMEM**: No memory is available in the system to perform the operation.

**PerformanceNote**

By taking advantage of the Network Buffer Cache, `send_file` provides better performance and network throughput for file transmission. It is recommended for files bigger than 4K bytes.

**Related Information**

The `connect` subroutine, `getsockopt` subroutine, `recv` subroutine, `recvfrom` subroutine, `recvmsg` subroutine, `select` subroutine, `sendfile` subroutine, `sendto` subroutine, `setsockopt` subroutine, `shutdown` subroutine, `socket` subroutine.

[Sockets Overview](#) and [Understanding Socket Data Transfer](#) in *AIX 5L Version 5.3 Communications Programming Concepts*.

---

**set_auth_method Subroutine**

**Purpose**

Sets the authentication methods for the rcmds for this system.

**Library**

Authentication Methods Library (libauthm.a)

**Syntax**

**Description**

This method configures the authentication methods for the system. The authentication methods should be passed to the function in the order in which they should be attempted in the unsigned integer pointer in which the user passed.

The list is an array of unsigned integers terminated by a zero. Each integer identifies an authentication method. The order that a client should attempt to authenticate is defined by the order of the list.

The flags identifying the authentication methods are defined in the `/usr/include/authm.h` file.
Any undefined bits in the input parameter invalidate the entire command. If the same authentication method is specified twice or if any authentication method is specified after Standard AIX, the command fails.

The user must have root authority or this method fails.

**Parameter**

authm  Points to an array of unsigned integers. The list of authentication methods to be set is terminated by a zero.

**Return Values**

Upon successful completion, the `set_auth_method` subroutine returns a zero.

Upon unsuccessful completion, the `set_auth_method` subroutine returns an `errno`.

**Related Information**

The `chauthent` command, `ftp` command, `lsauthent` command, `rcp` command, `rlogin` command, `rsh` command, `telnet`, `tn`, or `tn3270` command.

The `get_auth_method` subroutine.

Communications and networks in *Networks and communication management*.

Authentication and the secure rcmds in *Networks and communication management*.

**setdomainname Subroutine**

**Purpose**

Sets the name of the current domain.

**Library**

Standard C Library (`libc.a`)

**Syntax**

```c
int setdomainname (Name, Namelen);
char *Name;
int Namelen;
```

**Description**

The `setdomainname` subroutine sets the name of the domain for the host machine. It is normally used when the system is bootstrapped. You must have root user authority to run this subroutine.

The purpose of domains is to enable two distinct networks that may have host names in common to merge. Each network would be distinguished by having a different domain name. At the current time, only Network Information Service (NIS) makes use of domains set by this subroutine.

All applications containing the `setdomainname` subroutine must be compiled with the `_BSD` macro set to a specific value. Acceptable values are 43 and 44. In addition, all socket applications must include the BSD `libbsd.a` library.
Note: Domain names are restricted to 256 characters.

Parameters

*Name* Specifies the domain name to be set.
*Namelen* Specifies the size of the array pointed to by the *Name* parameter.

Return Values

If the call succeeds, a value of 0 is returned. If the call is unsuccessful, a value of -1 is returned and an error code is placed in the `errno` global variable.

Error Codes

The following errors may be returned by this subroutine:

<table>
<thead>
<tr>
<th>Error</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EFAULT</td>
<td>The <em>Name</em> parameter gave an invalid address.</td>
</tr>
<tr>
<td>EPERM</td>
<td>The caller was not the root user.</td>
</tr>
</tbody>
</table>

Related Information

The `getdomainname` subroutine, `gethostname` subroutine, `sethostname` subroutine.

Sockets Overview in *AIX 5L Version 5.3 Communications Programming Concepts*.

sethostent Subroutine

Purpose

Opens network host file.

Library

Standard C Library (`libc.a`)
( `libbind` )
( `libnis` )
( `liblocal` )

Syntax

```c
#include <netdb.h>
sethostent (StayOpen)
int StayOpen;
```

Description

When using the `sethostent` subroutine in DNS/BIND name service resolution, `sethostent` allows a request for the use of a connected socket using TCP for queries. If the *StayOpen* parameter is non-zero, this sets the option to send all queries to the name server using TCP and to retain the connection after each call to `gethostbyname` or `gethostbyaddr`.

When using the `sethostent` subroutine to search the `/etc/hosts` file, `sethostent` opens and rewinds the `/etc/hosts` file. If the *StayOpen* parameter is non-zero, the hosts database is not closed after each call to `gethostbyname` or `gethostbyaddr`. 
Parameters

StayOpen When used in NIS name resolution and to search the local /etc/hosts file, it contains a value used to indicate whether to close the host file after each call to gethostbyname and gethostbyaddr. A non-zero value indicates not to close the host file after each call and a zero value allows the file to be closed.

When used in DNS/BIND name resolution, a non-zero value retains the TCP connection after each call to gethostbyname and gethostbyaddr. A value of zero allows the connection to be closed.

Files

/etc/hosts Contains the host name database.
/etc/netsvc.conf Contains the name services ordering.
/etc/include/netdb.h Contains the network database structure.

Related Information

The endhostent subroutine, gethostbyaddr subroutine, gethostbyname subroutine, gethostent subroutine.

Sockets Overview and Network Address Translation in AIX 5L Version 5.3 Communications Programming Concepts.

sethostent_r Subroutine

Purpose
Opens network host file.

Library
Standard C Library (libc.a)
(libbind)
(libnis)
(liblocal)

Syntax
#include <netdb.h>
sethostent_r (StayOpenflag, ht_data)

int StayOpenflag;
struct hostent_data *ht_data;

Description
When using the sethostent_r subroutine in DNS/BIND name service resolution, sethostent_r allows a request for the use of a connected socket using TCP for queries. If the StayOpen parameter is non-zero, this sets the option to send all queries to the name server using TCP and to retain the connection after each call to gethostbyname_r or gethostbyaddr_r.

When using the sethostent_r subroutine to search the /etc/hosts file, sethostent_r opens and rewinds the /etc/hosts file. If the StayOpen parameter is non-zero, the hosts database is not closed after each call to gethostbyname_r or gethostbyaddr_r. It internally runs the sethostent command.
Parameters

*StayOpenflag*  
When used in NIS name resolution and to search the local `/etc/hosts` file, it contains a value used to indicate whether to close the host file after each call to the *gethostbyname* and *gethostbyaddr* subroutines. A non-zero value indicates not to close the host file after each call, and a zero value allows the file to be closed.

When used in DNS/BIND name resolution, a non-zero value retains the TCP connection after each call to *gethostbyname* and *gethostbyaddr*. A value of zero allows the connection to be closed.

*ht_data*  
Points to the *hostent_data* structure.

Files

`/etc/hosts`  
Contains the host name database.

`/etc/netsvc.conf`  
Contains the name services ordering.

`/etc/include/netdb.h`  
Contains the network database structure.

Related Information

“endhostent_r Subroutine” on page 41, “gethostbyname_r Subroutine” on page 70, “gethostbyaddr_r Subroutine” on page 67, and “gethostent_r Subroutine” on page 72.

sethostid Subroutine

**Purpose**  
Sets the unique identifier of the current host.

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

**Syntax**

```c
int sethostid (HostID);
```

**Description**  
The *sethostid* subroutine allows a calling process with a root user ID to set a new 32-bit identifier for the current host. The *sethostid* subroutine enables an application program to reset the host ID.

All applications containing the *sethostid* subroutine must be compiled with the `_BSD` macro set to a specific value. Acceptable values are 43 and 44. In addition, all socket applications must include the BSD *libbsd.a* library.

**Parameters**

*HostID*  
Specifies the unique 32-bit identifier for the current host.

**Return Values**  
Upon successful completion, the *sethostid* subroutine returns a value of 0.

If the *sethostid* subroutine is unsuccessful, the subroutine handler performs the following functions:
Error Codes
The `sethostid` subroutine is unsuccessful if the following is true:

<table>
<thead>
<tr>
<th>Error</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EPERM</td>
<td>The calling process did not have an effective user ID of root user.</td>
</tr>
</tbody>
</table>

Related Information
The `getsockname` subroutine, `gethostid` subroutine, `gethostname` subroutine.

Sockets Overview in AIX 5L Version 5.3 Communications Programming Concepts.

sethostname Subroutine

Purpose
Sets the name of the current host.

Library
Standard C Library (`libc.a`)

Syntax
```c
int sethostname (char *Name, int NameLength);
```

Description
The `sethostname` subroutine sets the name of a host machine. Only programs with a root user ID can use this subroutine.

The `sethostname` subroutine allows a calling process with root user authority to set the internal host name of a machine on a network.

All applications containing the `sethostname` subroutine must be compiled with the `_BSD` macro set to a specific value. Acceptable values are 43 and 44. In addition, all socket applications must include the BSD `libbsd.a` library.

Parameters
- **Name**: Specifies the name of the host machine.
- **NameLength**: Specifies the length of the `Name` array.

Return Values
Upon successful completion, the system returns a value of 0.

If the `sethostname` subroutine is unsuccessful, the subroutine handler performs the following functions:

- Returns a value of -1 to the calling program.
• Moves an error code, indicating the specific error, into the \texttt{errno} global variable. For further explanation of the \texttt{errno} variable see Error Notification Object Class in AIX 5L Version 5.3 General Programming Concepts: Writing and Debugging Programs.

Error Codes
The \texttt{sethostname} subroutine is unsuccessful if any of the following errors occurs:

<table>
<thead>
<tr>
<th>Error</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EFAULT</td>
<td>The Name parameter or NameLength parameter gives an address that is not valid.</td>
</tr>
<tr>
<td>EPERM</td>
<td>The calling process did not have an effective root user ID.</td>
</tr>
</tbody>
</table>

Related Information
The \texttt{gethostid} subroutine, \texttt{gethostname} subroutine, \texttt{sethostid} subroutine.

Sockets Overview and Understanding Network Address Translation in AIX 5L Version 5.3 Communications Programming Concepts.

\texttt{setnetent} Subroutine

Purpose
Opens the \texttt{/etc/networks} file and sets the file marker.

Library
Standard C Library (\texttt{libc.a})

Syntax

```
#include <netdb.h>
void setnetent (\texttt{StayOpen})
int \texttt{StayOpen};
```

Description
The \texttt{setnetent} subroutine is threadsafe in AIX 4.3 and later. However, the return value points to static data that is overwritten by subsequent calls. This data must be copied to be saved for use by subsequent calls.

The \texttt{setnetent} subroutine opens the \texttt{/etc/networks} file and sets the file marker at the beginning of the file.

All applications containing the \texttt{setnetent} subroutine must be compiled with the \_BSD macro set to a specific value. Acceptable values are 43 and 44. In addition, all socket applications must include the BSD \texttt{libbsd.a} library.

Parameters

\begin{itemize}
\item \texttt{StayOpen} Contains a value used to indicate when to close the \texttt{/etc/networks} file.
\end{itemize}

Specifying a value of 0 closes the \texttt{/etc/networks} file after each call to the \texttt{getnetent} subroutine.

Specifying a nonzero value leaves the \texttt{/etc/networks} file open after each call.

Return Values
If an error occurs or the end of the file is reached, the \texttt{setnetent} subroutine returns a null pointer.
Files

/etc/networks

Contains official network names.

Related Information

The `endnetent_r` subroutine, `getnetbyaddr_r` subroutine, `getnetbyname_r` subroutine, `getnetent_r` subroutine.

Sockets Overview and Understanding Network Address Translation in AIX 5L Version 5.3 Communications Programming Concepts.

setnetent_r Subroutine

Purpose

Opens the /etc/networks file and sets the file marker.

Library

Standard C Library (libc.a)

Syntax

```c
#include <netdb.h>
int setnetent_r(StayOpenflag, net_data)
struct netent_data *net_data;
int StayOpenflag;
```

Description

The setnetent_r subroutine opens the /etc/networks file and sets the file marker at the beginning of the file.

Parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>StayOpenflag</td>
<td>Contains a value used to indicate when to close the /etc/networks file.</td>
</tr>
<tr>
<td>Specifying a value of 0 closes the /etc/networks file after each call to the getnetent subroutine. Specifying a nonzero value leaves the /etc/networks file open after each call.</td>
<td></td>
</tr>
<tr>
<td>net_data</td>
<td>Points to the netent_data structure.</td>
</tr>
</tbody>
</table>

Files

/etc/networks

Contains official network names.

Related Information

"endnetent_r Subroutine" on page 43, "getnetbyaddr_r Subroutine" on page 77, "getnetbyname_r Subroutine" on page 79, and "getnetent_r Subroutine" on page 81.

setnetgrent_r Subroutine

Purpose

Handles the group network entries.
Library
Standard C Library (libc.a)

Syntax
```
#include <netdb.h>
int setnetgrent_r(NetGroup,ptr)
char *NetGroup;
void **ptr;
```

Description
The `setnetgrent_r` subroutine functions the same as the `setnetgrent` subroutine.

The `setnetgrent_r` subroutine establishes the network group from which the `getnetgrent_r` subroutine will obtain members. This subroutine also restarts calls to the `getnetgrent_r` subroutine from the beginning of the list. If the previous `setnetgrent_r` call was to a different network group, an `endnetgrent_r` call is implied. The `endnetgrent_r` subroutine frees the space allocated during the `getnetgrent_r` calls.

Parameters
- **NetGroup**
  Points to a network group.
- **ptr**
  Keeps the function threadsafe.

Return Values
The `setnetgrent_r` subroutine returns a 0 if successful and a -1 if unsuccessful.

Files
- `/etc/netgroup`
  Contains network groups recognized by the system.
- `/usr/include/netdb.h`
  Contains the network database structures.

Related Information
- “getnetgrent_r Subroutine” on page 82
- “endnetgrent_r Subroutine” on page 44

setprotoent Subroutine

Purpose
Opens the `/etc/protocols` file and sets the file marker.

Library
Standard C Library (libc.a)

Syntax
```
#include <netdb.h>
void setprotoent (StayOpen)
int StayOpen;
```
Description
The `setprotoent` subroutine is threadsafe in AIX 4.3 and later. However, the return value points to static data that is overwritten by subsequent calls. This data must be copied to be saved for use by subsequent calls.

The `setprotoent` subroutine opens the `/etc/protocols` file and sets the file marker to the beginning of the file.

All applications containing the `setprotoent` subroutine must be compiled with the `_BSD` macro set to a specific value. Acceptable values are 43 and 44. In addition, all socket applications must include the BSD `libbsd.a` library.

Parameters
- **StayOpen**
  - Indicates when to close the `/etc/protocols` file.
  - Specifying a value of 0 closes the file after each call to `getprotoent`.
  - Specifying a nonzero value allows the `/etc/protocols` file to remain open after each subroutine.

Return Values
The return value points to static data that is overwritten by subsequent calls.

Files
- **/etc/protocols**
  - Contains the protocol names.

Related Information
The `endprotoent`, `getprotobyname`, `getprotobynumber` subroutine, `getprotoent` subroutine.

Sockets Overview and Understanding Network Address Translation in `AIX 5L Version 5.3 Communications Programming Concepts`.

`setprotoent_r` Subroutine

Purpose
Opens the `/etc/protocols` file and sets the file marker.

Library
Standard C Library (`libc.a`)

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

int setprotoent_r(StayOpenflag, proto_data);
int StayOpenflag;
struct protoent_data *proto_data;
```
Description
The `setprotoent_r` subroutine opens the `/etc/protocols` file and sets the file marker to the beginning of the file.

Parameters
StayOpenflag Indicates when to close the `/etc/protocols` file.
Specifying a value of 0 closes the file after each call to `getprotoent`. Specifying a nonzero value allows the `/etc/protocols` file to remain open after each subroutine.

Files
`/etc/protocols` Contains the protocol names.

Related Information
"endprotoent_r Subroutine" on page 45, "getprotobynum_r Subroutine" on page 86, "getprotobyname_r Subroutine" on page 88, and "getprotoent_r Subroutine" on page 90.

setservent Subroutine

Purpose
Opens `/etc/services` file and sets the file marker.

Library
Standard C Library (`libc.a`)

Syntax
```c
#include <netdb.h>
void setservent ( StayOpen )
int StayOpen;
```

Description
The `setservent` subroutine is threadsafe in AIX 4.3 and later. However, the return value points to static data that is overwritten by subsequent calls. This data must be copied to be saved for use by subsequent calls.

The `setservent` subroutine opens the `/etc/services` file and sets the file marker at the beginning of the file.

All applications containing the `setservent` subroutine must be compiled with the _BSD macro set to a specific value. Acceptable values are 43 and 44. In addition, all socket applications must include the BSD `libbsd.a` library.

Parameters
`StayOpen` Indicates when to close the `/etc/services` file.
Specifying a value of 0 closes the file after each call to the `getservent` subroutine.
Specifying a nonzero value allows the file to remain open after each call.
Return Values
If an error occurs or the end of the file is reached, the setservent subroutine returns a null pointer.

Files
/etc/services Contains service names.

Related Information
The endprotoent subroutine, endservent subroutine, getprotobynumber subroutine, getprotobyname subroutine, getservbyport subroutine, getservbyname subroutine, getservbyport subroutine, getservent subroutine.

Sockets Overview and Understanding Network Address Translation in AIX 5L Version 5.3 Communications Programming Concepts.

setservent_r Subroutine

Purpose
Opens /etc/services file and sets the file marker.

Library
Standard C Library (libc.a)

Syntax
#include <netdb.h>

int setservent_r(StayOpenflag, serv_data)
int StayOpenflag;
struct servent_data serv_data;

Description
The setservent_r subroutine opens the /etc/services file and sets the file marker at the beginning of the file.

Parameters

StayOpenflag Indicates when to close the /etc/services file.

Specifying a value of 0 closes the file after each call to the getservent subroutine. Specifying a nonzero value allows the file to remain open after each call.

serv_data Points to the servent_data structure.

Files
/etc/services Contains service names.

Related Information
"endservent_r Subroutine" on page 47, "getservbyport_r Subroutine" on page 96, "getservent_r Subroutine" on page 98, and "getservbyname_r Subroutine" on page 93.
setsockopt Subroutine

Purpose
Sets socket options.

Library
Standard C Library (libc.a)

Syntax
```c
#include <sys/types.h>
#include <sys/socket.h>
#include <sys/socketvar.h>
#include <sys/atmsock.h>  /*Needed for SOCK_CONN_DGRAM socket type only*/

int setsockopt
(Socket, Level, OptionName, OptionValue, OptionLength)
int Socket, Level, OptionName;
const void *OptionValue;
size_t OptionLength;
```

Description
The `setsockopt` subroutine sets options associated with a socket. Options can exist at multiple protocol levels. The options are always present at the uppermost socket level.

The `setsockopt` subroutine provides an application program with the means to control a socket communication. An application program can use the `setsockopt` subroutine to enable debugging at the protocol level, allocate buffer space, control time outs, or permit socket data broadcasts. The `/usr/include/sys/socket.h` file defines all the options available to the `setsockopt` subroutine.

When setting socket options, specify the protocol level at which the option resides and the name of the option.

Use the parameters `OptionValue` and `OptionLength` to access option values for the `setsockopt` subroutine. These parameters identify a buffer in which the value for the requested option or options is returned.

All applications containing the `setsockopt` subroutine must be compiled with the `_BSD` macro set to a specific value. Acceptable values are 43 and 44. In addition, all socket applications must include the BSD `libbsd.a` library.

Parameters
- **Socket**: Specifies the unique socket name.
- **Level**: Specifies the protocol level at which the option resides. To set options at:
  - **Socket level**: Specifies the `Level` parameter as `SOL_SOCKET`.
  - **Other levels**: Supplies the appropriate protocol number for the protocol controlling the option. For example, to indicate that an option will be interpreted by the TCP protocol, set the `Level` parameter to the protocol number of TCP, as defined in the `netinet/in.h` file. Similarly, to indicate that an option will be interpreted by ATM protocol, set the `Level` parameter to NDDPROTO_ATM, as defined in `sys/atmsock.h`. 
OptionName Specifies the option to set. The OptionName parameter and any specified options are passed uninterpreted to the appropriate protocol module for interpretation. The sys/socket.h file defines the socket protocol level options. The netinet/tcp.h file defines the TCP protocol level options. The socket level options can be enabled or disabled; they operate in a toggle fashion.

The following list defines socket protocol level options found in the sys/socket.h file:

**SO_DEBUG**

Turns on recording of debugging information. This option enables or disables debugging in the underlying protocol modules. Set this option in one of the following ways at the command level:

- Use the **sodebug** command, which turns on or off this option for existing sockets.
- Specify `[DEBUG=[level]]` in the **wait**/**nowait** field of a service in **inetd.conf** in order to turn on this option for the specific service.
- Set the **sodebug_env** parameter to **no**, and specify **SODEBUG=level** in the process environment. This turns on or off this option for all subsequent sockets created by the process.

The value for **level** can be either **min**, **normal**, or **detail**.

**SO_REUSEADDR**

Specifies that the rules used in validating addresses supplied by a **bind** subroutine should allow reuse of a local port.

**SO_REUSEADDR** allows an application to explicitly deny subsequent **bind** subroutine to the port/address of the socket with **SO_REUSEADDR** set. This allows an application to block other applications from binding with the **bind** subroutine.

**SO_REUSEPORT**

Specifies that the rules used in validating addresses supplied by a **bind** subroutine should allow reuse of a local port/address combination. Each binding of the port/address combination must specify the **SO_REUSEPORT** socket option

**SO_CKSUMREV**

Enables performance enhancements in the protocol layers. If the protocol supports this option, enabling causes the protocol to defer checksum verification until the user’s data is moved into the user’s buffer (on **recv**, **recvfrom**, **read**, or **recvmsg** thread). This can cause applications to be awakened when no data is available, in the case of a checksum error. In this case, **EAGAIN** is returned. Applications that set this option must handle the **EAGAIN** error code returned from a receive call.

**SO_KEEPALIVE**

Monitors the activity of a connection by enabling or disabling the periodic transmission of ACK messages on a connected socket. The idle interval time can be designated using the **TCP/IP no** command. Broken connections are discussed in “Understanding Socket Types and Protocols” in AIX 5L Version 5.3 Communications Programming Concepts.

**SO_DONTROUTE**

Does not apply routing on outgoing messages. Indicates that outgoing messages should bypass the standard routing facilities. Instead, they are directed to the appropriate network interface according to the network portion of the destination address.
SO_BROADCAST
Permits sending of broadcast messages.

SO_LINGER
Lingers on a close subroutine if data is present. This option controls the action taken when an unsent messages queue exists for a socket, and a process performs a close subroutine on the socket.

If SO_LINGER is set, the system blocks the process during the close subroutine until it can transmit the data or until the time expires. If SO_LINGER is not specified and a close subroutine is issued, the system handles the call in a way that allows the process to continue as quickly as possible.

The sys/socket.h file defines the linger structure that contains the l_linger member for specifying linger time interval. If linger time is set to anything but 0, the system tries to send any messages queued on the socket. The maximum value that the l_linger member can be set to is 65535. If the application has requested SPEC1170 compliant behavior by exporting the XPG_SUS_ENV environment variable, the linger time is n seconds; otherwise, the linger time is n/100 seconds (ticks), where n is the value of the l_linger member.

SO_OOBINLINE
Leaves received out-of-band data (data marked urgent) in line.

SO_SNDBUF
Sets send buffer size.

SO_RCVBUF
Sets receive buffer size.

SO_SNDLOWAT
Sets send low-water mark.

SO_RCVLOWAT
Sets receive low-water mark.

SO_SNDTIMEO
Sets send time out. This option is setable, but currently not used.

SO_RCVTIMEO
Sets receive time out. This option is setable, but currently not used.

SO_ERROR
Sets the retrieval of error status and clear.

SO_TYPE
Sets the retrieval of a socket type.
The following list defines TCP protocol level options found in the `netinet/tcp.h` file:

**TCP_KEEPCNT**
Specifies the maximum number of keepalive packets to be sent to validate a connection. This socket option value is inherited from the parent socket. The default is 8.

**TCP_KEEPIDLE**
Specifies the number of seconds of idle time on a connection after which TCP sends a keepalive packet. This socket option value is inherited from the parent socket from the accept system call. The default value is 7200 seconds (14400 half seconds).

**TCP_KEEPINTVL**
Specifies the interval of time between keepalive packets. It is measured in seconds. This socket option value is inherited from the parent socket from the accept system call. The default value is 75 seconds (150 half seconds).

**TCP_NODELAY**
Specifies whether TCP should follow the Nagle algorithm for deciding when to send data. By default, TCP will follow the Nagle algorithm. To disable this behavior, applications can enable `TCP_NODELAY` to force TCP to always send data immediately. For example, `TCP_NODELAY` should be used when there is an application using TCP for a request/response.

**TCP_RFC1323**
Enables or disables RFC 1323 enhancements on the specified TCP socket. An application might contain the following lines to enable RFC 1323:

```c
int on=1;
setsockopt(s,IPPROTO_TCP,TCP_RFC1323,&on,sizeof(on));
```

**TCP_STDURG**
Enables or disables RFC 1122 compliant urgent point handling. By default, TCP implements urgent pointer behavior compliant with the 4.2 BSD operating system, i.e., this option defaults to 0.

**TCP_NODELAYACK**
Specifies if TCP needs to send immediate acknowledgement packets to the sender. If this option is not set, TCP delays sending the acknowledgement packets by up to 200 ms. This allows the acknowledgements to be sent along with the data on a response and minimizes system overhead. Setting this TCP option might cause a slight increase in system overhead, but can result in higher performance for network transfers if the sender is waiting on the receiver’s acknowledgements.
Beginning at AIX 4.3.2, TCP protocol level socket options are inherited from listening sockets to new sockets. Prior to 4.3.2, only the TCP_RFC1323 option was inherited.

The following list defines ATM protocol level options found in the `sys/atmsock.h` file:

**SO_ATM_PARAM**
- Sets all ATM parameters. This socket option can be used instead of using individual sockets options described below. It uses the `connect_ie` structure defined in `sys/call_ie.h` file.

**SO_ATM_AAL_PARM**
- Sets ATM AAL(Adaptation Layer) parameters. It uses the `aal_parm` structure defined in `sys/call_ie.h` file.

**SO_ATM_TRAFFIC_DES**
- Sets ATM Traffic Descriptor values. It uses the `traffic` structure defined in `sys/call_ie.h` file.

**SO_ATM_BEARER**
- Sets ATM Bearer capability. It uses the `bearer` structure defined in `sys/call_ie.h` file.

**SO_ATM_BHLI**
- Sets ATM Broadband High Layer Information. It uses the `bhli` structure defined in `sys/call_ie.h` file.

**SO_ATM_BLLI**
- Sets ATM Broadband Low Layer Information. It uses the `blli` structure defined in `sys/call_ie.h` file.

**SO_ATM_QOS**
- Sets ATM Quality Of Service values. It uses the `qos_parm` structure defined in `sys/call_ie.h` file.

**SO_ATM_TRANSIT_SEL**
- Sets ATM Transit Selector Carrier. It uses the `transit_sel` structure defined in `sys/call_ie.h` file.

**SO_ATM_ACCEPT**
- Indicates acceptance of an incoming ATM call, which was indicated to the application via `ACCEPT` system call. This must be issues for the incoming connection to be fully established. This allows negotiation of ATM parameters.

**SO_ATM_MAX_PEND**
- Sets the number of outstanding transmit buffers that are permitted before an error indication is returned to applications as a result of a transmit operation. This option is only valid for non best effort types of virtual circuits. `OptionValue/OptionLength` point to a byte which contains the value that this parameter will be set to.

*OptionValue*
- The `OptionValue` parameter takes an `Int` parameter. To enable a Boolean option, set the `OptionValue` parameter to a nonzero value. To disable an option, set the `OptionValue` parameter to 0.

The following options enable and disable in the same manner:

- **SO_DEBUG**
- **SO_REUSEADDR**
- **SO_KEEPALIVE**
- **SO_DONTROUTE**
- **SO_BROADCAST**
- **SO_OOBINLINE**
- **SO_LINGER**
- **TCP_RFC1323**

*OptionLength*
- The `OptionLength` parameter contains the size of the buffer pointed to by the `OptionValue` parameter.
Options at other protocol levels vary in format and name.

**IP level (IPPROTO_IP level) options are defined as follows:**

**IP_DONTFRAG**
Sets DF bit from now on for every packet in the IP header. Beginning with AIX 5.3, to detect decreases in Path MTU, UDP applications will always need to set this option.

**IP_FNDPMTU**
Sets enable/disable PMTU discovery for this path. Protocol level path MTU discovery should be enabled for the discovery to happen.

**IP_PMTUAGE**
Sets the age of PMTU. Specifies the frequency of PMT reductions discovery for the session. Setting it to 0 (zero) implies infinite age and PMTU reduction discovery will not be attempted. This will replace the previously set PMTU age. The new PMTU age will become effective after the currently set timer expires. Beginning with AIX 5.3, this option is unused because UDP applications will always need to set the IP_DONTFRAG socket option to detect decreases in PMTU immediately.

**IP_TTL**
Sets the time-to-live field in the IP header for every packet. However, for raw sockets, the default MAXTTL value will be used while sending the messages irrespective of the value set using the setsockopt subroutine.

**IP_HDRINCL**
This option allows users to build their own IP header. It indicates that the complete IP header is included with the data and can be used only for raw sockets.

**IPV6 level (IPPROTO_IPV6 level) options are defined as follows:**

<table>
<thead>
<tr>
<th>IPV6_V6ONLY</th>
<th>Restricts AF_INET6 sockets to IPV6 communications only.</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Option Type:</strong></td>
<td>int (boolean interpretation)</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>IPV6_UNICAST_HOPS</th>
<th>Allows the user to set the outgoing hop limit for unicast IPV6 packets.</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Option Type:</strong></td>
<td>int (x)</td>
</tr>
<tr>
<td><strong>Option Value:</strong></td>
<td>\n  x &lt; -1 Error EINVAL \n  x == -1 Use kernel default \n  0 &lt;= x &lt;= 255 Use x \n  x &gt;= 256 Error EINVAL</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>IPV6_MULTICAST_HOPS</th>
<th>Allows the user to set the outgoing hop limit for multicast IPV6 packets.</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Option Type:</strong></td>
<td>int (x)</td>
</tr>
<tr>
<td><strong>Option Value:</strong></td>
<td>Interpretation is the same as IPV6_UNICAST_HOPS (listed above).</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>IPV6_MULTICAST_IF</th>
<th>Allows the user to specify the interface being used for outgoing multicast packets. If specified as 0, the system selects the outgoing interface.</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Option Type:</strong></td>
<td>unsigned int (index of interface to use)</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>IPV6_MULTICAST_LOOP</th>
<th>If a multicast datagram is sent to a group that the sending host belongs to, a copy of the datagram is looped back by the IP layer for local delivery (if the option is set to 1). If the option is set to 0, a copy is not looped back.</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Option Type:</strong></td>
<td>unsigned int</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>IPV6_JOIN_GROUP</th>
<th>Joins a multicast group on a specified local interface. If the interface index is specified as 0, the kernel chooses the local interface.</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Option Type:</strong></td>
<td>struct ipv6_mreq as defined in the netinet/in.h file</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>IPV6_LEAVE_GROUP</th>
<th>Leaves a multicast group on a specified interface.</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Option Type:</strong></td>
<td>struct ipv6_mreq as defined in the netinet/in.h file</td>
</tr>
<tr>
<td>Option Name</td>
<td>Description</td>
</tr>
<tr>
<td>------------------</td>
<td>-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>IPV6_CHECKSUM</td>
<td>Specifies that the kernel computes checksums over the data and the pseudo-IPv6 header for a raw socket. The kernel will compute the checksums for outgoing packets as well as verify checksums for incoming packets on that socket. Incoming packets with incorrect checksums will be discarded. This option is disabled by default.</td>
</tr>
<tr>
<td>IPV6_RECVPKTINFO</td>
<td>Causes the destination IPv6 address and arriving interface index of incoming IPv6 packets to be received as ancillary data on UDP and raw sockets.</td>
</tr>
<tr>
<td>IPV6_RECVHOPLIMIT</td>
<td>Causes the hop limit of incoming IPv6 packets to be received as ancillary data on UDP and raw sockets.</td>
</tr>
<tr>
<td>IPV6_RECVTCLASS</td>
<td>Causes the traffic class of incoming IPv6 packets to be received as ancillary data on UDP and raw sockets.</td>
</tr>
<tr>
<td>IPV6_RECRTHDR</td>
<td>Causes the routing header (if any) of incoming IPv6 packets to be received as ancillary data on UDP and raw sockets.</td>
</tr>
<tr>
<td>IPV6_RECVHOPOPTS</td>
<td>Causes the hop-by-hop options header (if any) of incoming IPv6 packets to be received as ancillary data on UDP and raw sockets.</td>
</tr>
<tr>
<td>IPV6_RECVDSTOPTS</td>
<td>Causes the destination options header (if any) of incoming IPv6 packets to be received as ancillary data on UDP and raw sockets.</td>
</tr>
<tr>
<td>IPV6_PKTINFO</td>
<td>Sets the source IPv6 address and outgoing interface index for all IPv6 packets being sent on this socket. This option can be cleared by doing a regular setsockopt with ipi6_addr being in6addr_any and ipi6_ifindex being 0.</td>
</tr>
<tr>
<td>IPV6_NEXTHOP</td>
<td>Sets the next hop for outgoing IPv6 datagrams on this socket. This option can be cleared by doing a regular setsockopt with a 0 length. Note that a memory pointer must still be supplied for the option value in this case.</td>
</tr>
</tbody>
</table>
| IPV6_TCLASS      | Sets the traffic class for outgoing IPv6 datagrams on this socket. To clear this option, the application can specify -1 as the value.                                                                                                                                                                                                                                                                                 | int (x)                   | x < -1 Error EINVAL  
 x == -1 Use kernel default  
 0 <= x <= 255 Use x  
 x >= 256 Error EINVAL                                                                                                           |
| IPV6_RTHDR       | Sets the routing header to be used for outgoing IPv6 datagrams on this socket. This option can be cleared by doing a regular setsockopt with a 0 length. Note that a memory pointer must still be supplied for the option value in this case.                                                                                                                                                               | struct ip6_rthdr defined in the netinet/ip6.h file. |                                                                                                      |
| IPV6_HOPOPTS     | Sets the hop-by-hop options header to be used for outgoing IPv6 datagrams on this socket. This option can be cleared by doing a regular setsockopt with a 0 length. Note that a memory pointer must still be supplied for the option value in this case.                                                                                                           | struct ip6_hbh defined in the netinet/ip6.h file. |                                                                                                      |
**IPV6_DSTOPTS**

Sets the destination options header to be used for outgoing IPv6 datagrams on this socket. This header will follow a routing header (if present) and will also be used when there is no routing header specified. This option can be cleared by doing a regular `setsockopt` with a 0 length. Note that a memory pointer must still be supplied for the option value in this case.

**Option Type:** struct `ip6_dest` defined in the `netinet/ip6.h` file.

**IPV6_RTHDRDSTOPTS**

Sets the destination options header to be used for outgoing IPv6 datagrams on this socket. This header will precede a routing header (if present). If no routing header is specified, this option will be silently ignored. This option can be cleared by doing a regular `setsockopt` with a 0 length. Note that a memory pointer must still be supplied for the option value in this case.

**Option Type:** struct `ip6_dest` defined in the `netinet/ip6.h` file.

**IPV6_USE_MIN_MTU**

Sets this option to control IPv6 path MTU discovery.

**Option Type:** int

**Option Value:**
- `-1` Performs path MTU discovery for unicast destinations, but does not perform it for multicast destinations.
- `0` Always performs path MTU discovery.
- `1` Always disables path MTU discovery and sends packets at the minimum MTU.

**IPV6_DONTFRAG**

Setting this option prevents fragmentation of outgoing IPv6 packets on this socket. If a packet is being sent that is larger than the outgoing interface MTU, the packet will be discarded.

**Option Type:** int (boolean interpretation)

**IPV6_RECVPATHMTU**

Enables the receipt of `IPV6_PATHMTU` ancillary data items by setting this option.

**Option Type:** int (boolean interpretation)

**ICMPV6 level (IPPROTO_ICMPV6 level) options are defined as follows:**

**ICMP6_FILTER**

Allows the user to filter ICMPV6 messages by the ICMPV6 type field. In order to clear an existing filter, issue a `setsockopt` call with zero length.

**Option Type:** The `icmp6_filter` structure defined in the `netinet/icmp6.h` file.

The following values (defined in the `/usr/include/netint/tcp.h` file) are used by the `setsockopt` subroutine to configure the `dacinet` functions.

**Note:** The DACinet facility is available only in a CAPP/EAL4+ configured AIX system.

```
tcp.h:#define TCP_ACLFLUSH 0x21 /* clear all DACinet ACLs */
tcp.h:#define TCP_ACLCLEAR 0x22 /* clear DACinet ACL */
tcp.h:#define TCP_ACLADD 0x23 /* Add to DACinet ACL */
tcp.h:#define TCP_ACLDEL 0x24 /* Delete from DACinet ACL */
tcp.h:#define TCP_ACLLS 0x25 /* List DACinet ACL */
tcp.h:#define TCP_ACLBIND 0x26 /* Set port number for TCP_ACLLS */
tcp.h:#define TCP_ACLGID 0x01 /* id being added to ACL is a gid */
tcp.h:#define TCP_ACLUID 0x02 /* id being added to ACL is a gid */
tcp.h:#define TCP_ACLSUBNET 0x04 /* address being added to ACL is a subnet */
tcp.h:#define TCP_ACLDENY 0x08 /* this ACL entry is for denying access */
```

**Return Values**

Upon successful completion, a value of 0 is returned.

If the `setsockopt` subroutine is unsuccessful, the subroutine handler performs the following functions:
- Returns a value of -1 to the calling program.
• Moves an error code, indicating the specific error, into the errno global variable. For further explanation of the errno variable see Error Notification Object Class in AIX 5L Version 5.3 General Programming Concepts: Writing and Debugging Programs.

Error Codes
The setsockopt subroutine is unsuccessful if any of the following errors occurs:

- EBADF: The Socket parameter is not valid.
- EFAULT: The Address parameter is not in a writable part of the user address space.
- EINVAL: The OptionValue parameter or the OptionLength parameter is invalid or the socket has been shutdown.
- ENOBUFS: There is insufficient memory for an internal data structure.
- ENOTSOCK: The Socket parameter refers to a file, not a socket.
- ENOPROTOOPT: The option is unknown.
- EOPNOTSUPP: The option is not supported by the socket family or socket type.

Examples
To mark a socket for broadcasting:
```c
int on=1;
setsockopt(s, SOL_SOCKET, SO_BROADCAST, &on, sizeof(on));
```

To turn on the TCP_NODELAYACK option, run the following:
```c
int on=1;
setsockopt(s, IPPROTO_TCP, TCP_NODELAYACK, &on, sizeof(on));
```

Related Information
The no command.

The bind subroutine, endprotoent subroutine, getprotobynumber subroutine, getprotoent subroutine, getsockopt subroutine, setprotoent subroutine, setsockopt subroutine.

Sockets Overview, Understanding Socket Options, Understanding Socket Types and Protocols in AIX 5L Version 5.3 Communications Programming Concepts.

shutdown Subroutine

Purpose
Shuts down all socket send and receive operations.

Library
Standard C Library (libc.a)

Syntax
```c
#include <sys/socket.h>

int shutdown (Socket, How)
int Socket, How;
```

Description
The shutdown subroutine disables all receive and send operations on the specified socket.
All applications containing the `shutdown` subroutine must be compiled with the `_BSD` macro set to a specific value. Acceptable values are 43 and 44. In addition, all socket applications must include the BSD `libbsd.a` library.

### Parameters

**Socket**  
Specifies the unique name of the socket.

**How**  
Specifies the type of subroutine shutdown. Use the following values:

- 0  
  Disables further receive operations.
- 1  
  Disables further send operations.
- 2  
  Disables further send operations and receive operations.

### Return Values

Upon successful completion, a value of 0 is returned.

If the `shutdown` subroutine is unsuccessful, the subroutine handler performs the following functions:

- Returns a value of -1 to the calling program.
- Moves an error code, indicating the specific error, into the `errno` global variable. For further explanation of the `errno` variable see [Error Notification Object Class](#) in AIX 5L Version 5.3 General Programming Concepts: Writing and Debugging Programs.

### Error Codes

The `shutdown` subroutine is unsuccessful if any of the following errors occurs:

<table>
<thead>
<tr>
<th>Error</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EBADF</td>
<td>The <code>Socket</code> parameter is not valid.</td>
</tr>
<tr>
<td>EINVAL</td>
<td>The <code>How</code> parameter is invalid.</td>
</tr>
<tr>
<td>ENOTCONN</td>
<td>The socket is not connected.</td>
</tr>
<tr>
<td>ENOTSOCK</td>
<td>The <code>Socket</code> parameter refers to a file, not a socket.</td>
</tr>
</tbody>
</table>

### Files

- `/usr/include/sys/socket.h`  
  Contains socket definitions.
- `/usr/include/sys/types.h`  
  Contains definitions of unsigned data types.

### Related Information

- `getsockopt` subroutine, `recv` subroutine, `recvfrom` subroutine, `recvmsg` subroutine, `read` subroutine, `select` subroutine, `send` subroutine, `sendto` subroutine, `setsockopt` subroutine, `socket` subroutine, `write` subroutine.

[Sockets Overview](#) in AIX 5L Version 5.3 Communications Programming Concepts.

### SLPAttrCallback Subroutine

#### Purpose

Returns the same callback type as the `SLPFindAttrs()` function.
**Syntax**

typedef SLPBoolean SLPAttrCallback(SLPHandle hSLP, 
const char* pcAttrList, 
SLPError errCode, 
void *pvCookie);

**Description**
The `SLPAttrCallback` type is the type of the callback function parameter to the `SLPFindAttrs()` function.

The `pcAttrList` parameter contains the requested attributes as a comma-separated list (or is empty if no attributes matched the original tag list).

**Parameters**

- **hSLP**
  The `SLPHandle` used to initiate the operation.

- **pcAttrList**
  A character buffer containing a comma-separated, null-terminated list of attribute ID/value assignments, in SLP wire format: "(attr-id=attr-value-list)"

- **errCode**
  An error code indicating if an error occurred during the operation. The callback should check this error code before processing the parameters. If the error code is other than `SLP_OK`, then the API library can choose to terminate the outstanding operation.

- **pvCookie**
  Memory passed down from the client code that called the original API function, starting the operation. Can be NULL.

**Return Values**
The client code should return `SLP_TRUE` if more data is desired; otherwise `SLP_FALSE` is returned.

**Related Information**

- "SLPClose Subroutine," "SLPEscape Subroutine" on page 216, "SLPFindAttrs Subroutine" on page 217, "SLPFindScopes Subroutine" on page 218, "SLPFindSrvs Subroutine" on page 218, "SLPFindSrvTypes Subroutine" on page 219, "SLPFree Subroutine" on page 220, "SLGetProperty Subroutine" on page 221, "SLPOpen Subroutine" on page 222, "SLPParseSrvURL Subroutine" on page 223, "SLPSrvTypeCallback Subroutine" on page 223, "SLPSrvURLCallback Subroutine" on page 224, "SLPUnescape Subroutine" on page 225

The `/etc/slp.conf` File.

The Service Location Protocol (SLP) API

**SLPClose Subroutine**

**Purpose**
Frees all resources associated with the handle.

**Syntax**

```c
void SLPClose(SLPHandle hSLP);
```

**Description**
The `SLPClose` subroutine frees all resources associated with the handle. If the handle was invalid, the function returns silently. Any outstanding synchronous or asynchronous operations are cancelled so that their callback functions will not be called any further.
Parameters

\[ hSLP \]

The \texttt{SLPHandle} handle returned from a call to \texttt{SLPOpen()}.  

---

Related Information

- "SLPAttrCallback Subroutine" on page 214
- "SLPEscape Subroutine" on page 217
- "SLPFindAttrs Subroutine" on page 218
- "SLPFindScopes Subroutine" on page 217
- "SLPFindSrvs Subroutine" on page 218
- "SLPFindSrvTypes Subroutine" on page 219
- "SLPFree Subroutine" on page 220
- "SLPGetProperty Subroutine" on page 221
- "SLPSrvTypeCallback Subroutine" on page 223
- "SLPSrvURLCallback Subroutine" on page 224
- "SLPUnescape Subroutine" on page 225

---

The \texttt{/etc/slp.conf} File

The Service Location Protocol (SLP) API

---

\textbf{SLPEscape Subroutine}

\textbf{Purpose}

Processes an input string and escapes any characters reserved for SLP.

\textbf{Syntax}

\begin{verbatim}
SLPError SLPEscape(const char* \textit{pcInbuf}, char** \textit{ppcOutBuf},
                   SLPBoolean \textit{isTag});
\end{verbatim}

\textbf{Description}

The \texttt{SLPEscape} subroutine processes the input string in \textit{pcInbuf} and escapes any characters reserved for SLP. If the \textit{isTag} parameter is \texttt{SLPTrue}, \texttt{SLPEscape} looks for bad tag characters and signals an error if any are found by returning the \texttt{SLP_PARSE_ERROR} code. The results are put into a buffer allocated by the API library and returned in the \textit{ppcOutBuf} parameter. This buffer should be deallocated using \texttt{SLPFree()} when the memory is no longer needed.

\textbf{Parameters}

- \textit{pcInbuf}  
  Pointer to the input buffer to process for escape characters.

- \textit{ppcOutBuf}  
  Pointer to a pointer for the output buffer with the characters reserved for SLP escaped. Must be freed using \texttt{SLPFree()} when the memory is no longer needed.

- \textit{isTag}  
  When true, the input buffer is checked for bad tag characters.

\textbf{Return Values}

The \texttt{SLPEscape} subroutine returns \texttt{SLP_PARSE_ERROR} if any characters are bad tag characters and the \textit{isTag} flag is true; otherwise, it returns \texttt{SLP_OK}, or the appropriate error code if another error occurs.

\textbf{Related Information}

- "SLPAttrCallback Subroutine" on page 214
- "SLPClose Subroutine" on page 215
- "SLPFindAttrs Subroutine" on page 217
- "SLPFindScopes Subroutine" on page 218
- "SLPFindSrvs Subroutine" on page 218
- "SLPFindSrvTypes Subroutine" on page 219
- "SLPFree Subroutine" on page 220
- "SLPGetProperty Subroutine" on page 221
- "SLPSrvTypeCallback Subroutine" on page 223
- "SLPSrvURLCallback Subroutine" on page 224
- "SLPUnescape Subroutine" on page 225
The /etc/slp.conf File.

The Service Location Protocol (SLP) API

SLPFindAttrs Subroutine

Purpose
Returns service attributes that match the attribute IDs for the indicated service URL or service type.

Syntax

```c
SLPError SLPFindAttrs(SLPHandle hSLP,
const char *pcURLOrServiceType,
const char *pcScopeList,
const char *pcAttrIds,
SLPAttrCallback callback,
void *pvCookie);
```

Description
The SLPFindAttrs subroutine returns service attributes matching the attribute IDs for the indicated service URL or service type. If `pcURLOrServiceType` is a service URL, the attribute information returned is for that particular advertisement in the language locale of the `SLPHandle`.

If `pcURLOrServiceType` is a service type name (including naming authority if any), then the attributes for all advertisements of that service type are returned regardless of the language of registration. Results are returned through the `callback`.

The result is filtered with an SLP attribute request filter string parameter. If the filter string is the empty string (""), all attributes are returned.

Parameters

- **hSLP**
  The `SLPHandle` on which to search for attributes.

- **pcURLOrServiceType**
  The service URL or service type. Cannot be the empty string.

- **pcScopeList**
  A pointer to a char containing a comma-separated list of scope names. Cannot be the empty string, "".

- **pcAttrIds**
  The filter string indicating which attribute values to return. Use the empty string ("") to indicate all values. Wildcards matching all attribute IDs having a particular prefix or suffix are also possible.

- **callback**
  A callback function through which the results of the operation are reported.

- **pvCookie**
  Memory passed to the callback code from the client. Can be NULL.

Return Values
If SLPFindAttrs is successful, it returns SLP_OK. If an error occurs in starting the operation, one of the SLPError codes is returned.

Related Information

- "SLPAttrCallback Subroutine" on page 214
- "SLPClose Subroutine" on page 215
- "SLPEscape Subroutine" on page 216
- "SLPFindScopes Subroutine" on page 218
- "SLPFindSrvs Subroutine" on page 218
- "SLPFindSrvTypes Subroutine" on page 219
- "SLPFree Subroutine" on page 220
- "SLGetProperty Subroutine" on page 221
- "SLPOpen Subroutine" on page 222
- "SLPParseSrvURL Subroutine" on page 223
- "SLPSrvTypeCallback Subroutine" on page 223
- "SLPSrvURLCallback Subroutine" on page 224
- "SLPUnescape Subroutine" on page 225
SLPFindScopes Subroutine

**Purpose**
Sets the `ppcScopeList` parameter to point to a comma-separated list that includes all available scope values.

**Syntax**
```c
SLPError SLPFindScopes(SLPHandle hSLP, char** ppcScopeList);
```

**Description**
The `SLPFindScopes` subroutine sets the `ppcScopeList` parameter to point to a comma-separated list that includes all available scope values. If there is any order to the scopes, preferred scopes are listed before less desirable scopes. There is always at least one name in the list, the default scope, `DEFAULT`.

**Parameters**
- `hSLP` The `SLPHandle` on which to search for scopes.
- `ppcScopeList` A pointer to a char pointer into which the buffer pointer is placed upon return. The buffer is null terminated. The memory should be freed by calling `SLPFree()`.

**Return Values**
If no error occurs, `SLPFindScopes` returns `SLP_OK`; otherwise, it returns the appropriate error code.

**Related Information**
- “SLPAttrCallback Subroutine” on page 214
- “SLPClose Subroutine” on page 215
- “SLPEscape Subroutine” on page 216
- “SLPFindAttrs Subroutine” on page 217
- “SLPFindSrvs Subroutine”
- “SLPFindSrvTypes Subroutine” on page 219
- “SLPFree Subroutine” on page 220
- “SLGetProperty Subroutine” on page 221
- “SLPOpen Subroutine” on page 222
- “SLPParseSrvURL Subroutine” on page 223
- “SLPSrvTypeCallback Subroutine”
- “SLPSrvURLCallback Subroutine” on page 224
- “SLPUunescape Subroutine” on page 225

The `/etc/slp.conf` File

The Service Location Protocol (SLP) API

SLPFindSrvs Subroutine

**Purpose**
Issues the query for services on the language-specific `SLPHandle` and returns the results through the `callback`.

**Syntax**
```c
SLPError SLPFindSrvs(SLPHandle hSLP, const char *pcServiceType, const char *pcScopeList);
```
Description

The `SLPFindSrvs` subroutine issues the query for services on the language-specific `SLPHandle` and returns the results through the callback. The parameters determine the results.

Parameters

- `hSLP`: The language-specific `SLPHandle` on which to search for services.
- `pcServiceType`: The Service Type String, including authority string if any, for the request, which can be discovered using `SLPSrvTypes()`. This could be, for example, "service:printer:lp" or "service:nfs". This cannot be the empty string ("").
- `pcScopeList`: A pointer to a char containing a comma-separated list of scope names. This cannot be the empty string ("").
- `pcSearchFilter`: A query formulated of attribute pattern matching expressions in the form of a LDAPv3 Search Filter. If this filter is empty (""), all services of the requested type in the specified scopes are returned.
- `callback`: A callback function through which the results of the operation are reported.
- `pvCookie`: Memory passed to the callback code from the client. Can be NULL.

Return Values

If `SLPFindSrvs` is successful, it returns `SLP_OK`. If an error occurs in starting the operation, one of the `SLPError` codes is returned.

Related Information

- "SLPAttrCallback Subroutine" on page 214
- "SLPClose Subroutine" on page 215
- "SLPEscape Subroutine" on page 216
- "SLPFindAttrs Subroutine" on page 217
- "SLPFindScopes Subroutine" on page 218
- "SLPFindSrvTypes Subroutine" on page 219
- "SLPParseSrvURL Subroutine" on page 220
- "SLPGetProperty Subroutine" on page 221
- "SLPOpen Subroutine" on page 222
- "SLPSrvURLCallback Subroutine" on page 224
- "SLPUnescape Subroutine" on page 225

The `/etc/slp.conf` File.

The Service Location Protocol (SLP) API.
Description
The SLPFindSrvType() subroutine issues an SLP service type request for service types in the scopes indicated by the pcScopeList. The results are returned through the callback parameter. The service types are independent of language locale, but only for services registered in one of the scopes and for the naming authority indicated by pcNamingAuthority.

If the naming authority is "*", then results are returned for all naming authorities. If the naming authority is the empty string, "", then the default naming authority, "IANA", is used. "IANA" is not a valid naming authority name, and it returns a PARAMETER_BAD error when it is included explicitly.

The service type names are returned with the naming authority intact. If the naming authority is the default (that is, the empty string), then it is omitted, as is the separating ".". Service type names from URLs of the service: scheme are returned with the "service:" prefix intact.

Parameters
\[hSLP\] The SLPHandle on which to search for types.
\[pcNamingAuthority\] The naming authority to search. Use "*" for all naming authorities and the empty string, "", for the default naming authority.
\[pcScopeList\] A pointer to a char containing a comma-separated list of scope names to search for service types. Cannot be the empty string, "".
\[callback\] A callback function through which the results of the operation are reported.
\[pvCookie\] Memory passed to the callback code from the client. Can be NULL.

Return Values
If SLPFindSrvTypes is successful, it returns SLP_OK. If an error occurs in starting the operation, one of the SLPError codes is returned.

Related Information
"SLPAttrCallback Subroutine" on page 214, "SLPClose Subroutine" on page 215, "SLPEscape Subroutine" on page 216, "SLPFindAttrs Subroutine" on page 217, "SLPFindScopes Subroutine" on page 218, "SLPFindSrvs Subroutine" on page 218, "SLPFree Subroutine", "SLPGetProperty Subroutine" on page 221, "SLPOpen Subroutine" on page 222, "SLPParseSrvURL Subroutine" on page 223, "SLPSrvTypeCallback Subroutine" on page 223, "SLPSrvURLCallback Subroutine" on page 224, "SLPUnescape Subroutine" on page 225.

The /etc/slp.conf File.

The Service Location Protocol (SLP) API.

SLPFree Subroutine

Purpose
Frees memory returned from SLPParseSrvURL(), SLPFindScopes(), SLPEscape(), and SLPUnescape().

Syntax
\[
\text{void SLPFree(\text{void* pvMem});}
\]

Description
The SLPFree subroutine frees memory returned from SLPParseSrvURL(), SLPFindScopes(), SLPEscape(), and SLPUnescape().
Parameters

pvMem A pointer to the storage allocated by the SLPParseSrvURL(), SLPEscape(), SLPUnescape(), or SLPFindScopes() function. Ignored if NULL.

Related Information


The /etc/slp.conf File.

The Service Location Protocol (SLP) API

---

SLGetProperty Subroutine

Purpose
Returns the value of the corresponding SLP property name.

Syntax

const char* SLPGetProperty(const char* pcName);

Description
The SLPGetProperty subroutine returns the value of the corresponding SLP property name. The returned string is owned by the library and must not be freed.

Parameters

pcName Null-terminated string with the property name.

Return Values
If no error, the SLPGetProperty subroutine returns a pointer to a character buffer containing the property value. If the property was not set, the subroutine returns the default value. If an error occurs, it returns NULL. The returned string must not be freed.

Related Information


The /etc/slp.conf File

The Service Location Protocol (SLP) API
SLPOpen Subroutine

Purpose
Returns an SLPHandle handle that encapsulates the language locale for SLP requests.

Syntax
SLPError SLPOpen(const char *pcLang, SLPBoolean isAsync, SLPHandle *phSLP);

Description
The SLPOpen subroutine returns an SLPHandle handle in the phSLP parameter for the language locale passed in as the pcLang parameter. The client indicates if operations on the handle are to be synchronous or asynchronous through the isAsync parameter. The handle encapsulates the language locale for SLP requests issued through the handle, and any other resources required by the implementation. However, SLP properties are not encapsulated by the handle; they are global. The return value of the function is an SLPError code indicating the status of the operation. Upon failure, the phSLP parameter is NULL.

Implementation Specifics
An SLPHandle can only be used for one SLP API operation at a time. If the original operation was started asynchronously, any attempt to start an additional operation on the handle while the original operation is pending results in the return of an SLP_HANDLE_IN_USE error from the API function. The SLPClose() API function terminates any outstanding calls on the handle. If an implementation is unable to support an asynchronous (resp. synchronous) operation, because of memory constraints or lack of threading support, the SLP_NOT_IMPLEMENTED flag might be returned when the isAsync flag is SLP_TRUE (resp. SLP_FALSE).

Parameters
pcLang A pointer to an array of characters (AIX supports “en” only).
isAsync An SLPBoolean indicating whether the SLPHandle should be opened for asynchronous operation or not. AIX supports synchronous operation only.
phSLP A pointer to an SLPHandle, in which the open SLPHandle is returned. If an error occurs, the value upon return is NULL.

Return Values
If SLPOpen is successful, it returns SLP_OK and an SLPHandle handle in the phSLP parameter for the language locale passed in as the pcLang parameter.

Error Codes
SLPError Indicates the status of the operation

Related Information
"SLPAttrCallback Subroutine" on page 214, "SLPClose Subroutine" on page 215, "SLPEscape Subroutine" on page 216, "SLPFindAttr Callback Subroutine" on page 217, "SLPFindScopes Subroutine" on page 218, "SLPFindSrvs Subroutine" on page 218, "SLPFindSrvTypes Subroutine" on page 219, "SLPFree Subroutine" on page 220, "SLPGetProperty Subroutine" on page 221, "SLPParseSrvURL Subroutine" on page 223, "SLPUnescape Subroutine" on page 225

The /etc/slp.conf File
**SLPParseSrvURL Subroutine**

**Purpose**
Parses the URL passed in as the argument into a service URL structure and returns it in the `ppSrvURL` pointer.

**Syntax**
```c
SLPError SLPParseSrvURL(char *pcSrvURL, SLPSrvURL **ppSrvURL);
```

**Description**
The `SLPParseSrvURL` subroutine parses the URL passed in as the argument into a service URL structure and returns it in the `ppSrvURL` pointer. If a parse error occurs, returns `SLP_PARSE_ERROR`. The input buffer `pcSrvURL` is destructively modified during the parse and used to fill in the fields of the return structure. The structure returned in `ppSrvURL` should be freed with `SLPFreeURL()`. If the URL has no service part, the `s_pcSrvPart` string is the empty string (""), not NULL. If `pcSrvURL` is not a service: URL, then the `s_pcSrvtType` field in the returned data structure is the URL’s scheme, which might not be the same as the service type under which the URL was registered. If the transport is IP, the `s_pcTransport` field is the empty string. If the transport is not IP or there is no port number, the `s_iPort` field is 0.

**Parameters**
- `pcSrvURL` A pointer to a character buffer containing the null-terminated URL string to parse. It is destructively modified to produce the output structure.
- `ppSrvURL` A pointer to a pointer for the `SLPSrvURL` structure to receive the parsed URL. The memory should be freed by a call to `SLPFree()` when no longer needed.

**Return Values**
If no error occurs, the return value is `SLP_OK`. Otherwise, the appropriate error code is returned.

**Related Information**
- “SLPAttrCallback Subroutine” on page 214
- “SLPClose Subroutine” on page 215
- “SLPEscape Subroutine” on page 216
- “SLPFindAttrs Subroutine” on page 217
- “SLPFindScopes Subroutine” on page 218
- “SLPFindSrvs Subroutine” on page 219
- “SLPFindSrvTypes Subroutine” on page 220
- “SLPGetProperty Subroutine” on page 221
- “SLPOpen Subroutine” on page 222
- “SLPSrvTypeCallback Subroutine
- “SLPSrvURLCallback Subroutine” on page 224
- “SLPUnescape Subroutine” on page 225

The `/etc/slp.conf` File

The Service Location Protocol (SLP) API

**SLPSrvTypeCallback Subroutine**

**Purpose**
Returns the same callback type as the `SLPFindSrvTypes()` function.
Syntax

typedef SLPBoolean SLPSrvTypeCallback(SLPHandle hSLP, const char* pcSrvTypes, SLPError errCode, void* pvCookie);

Description

The **SLPSrvTypeCallback** type is the type of the callback function parameter to the **SLPFindSrvTypes()** function.

Parameters

- **hSLP**
  The **SLPHandle** used to initiate the operation.

- **pcSrvTypes**
  A character buffer containing a comma-separated, null-terminated list of service types.

- **errCode**
  An error code indicating if an error occurred during the operation. The callback should check this error code before processing the parameters. If the error code is other than **SLP_OK**, then the API library can choose to terminate the outstanding operation.

- **pvCookie**
  Memory passed down from the client code that called the original API function, starting the operation. Can be NULL.

Return Values

The client code should return **SLP_TRUE** if more data is desired; otherwise **SLP_FALSE** is returned.

Related Information

- "SLPAttrCallback Subroutine" on page 214
- "SLPClose Subroutine" on page 215
- "SLPEscape Subroutine" on page 216
- "SLPFindAttrs Subroutine" on page 217
- "SLPFindScopes Subroutine" on page 218
- "SLPFindSrvs Subroutine" on page 218
- "SLPFindSrvTypes Subroutine" on page 219
- "SLPFree Subroutine" on page 220
- "SLPGetProperty Subroutine" on page 221
- "SLPOpen Subroutine" on page 222
- "SLPParseSrvURL Subroutine" on page 223
- "SLPSrvURLCallback Subroutine"
- "SLPUnescape Subroutine" on page 225

The /etc/slp.conf File

The Service Location Protocol (SLP) API

---

**SLPSrvURLCallback Subroutine**

Purpose

Returns the same callback type as the **SLPFindSrvs()** function.

Syntax

```c
typedef SLPBoolean SLPSrvURLCallback(SLPHandle hSLP, const char* pcSrvURL, unsigned short sLifetime, SLPError errCode, void* pvCookie);
```

Description

The **SLPSrvURLCallback** type is the type of the callback function parameter to the **SLPFindSrvs()** function.
Parameters

**hSLP**

The **SLPHandle** used to initiate the operation.

**pcSrvURL**

A character buffer containing the returned service URL.

**sLifetime**

An unsigned short giving the lifetime of the service advertisement, in seconds. The value must be an unsigned integer less than or equal to **SLP_LIFETIME_MAXIMUM**.

**errCode**

An error code indicating if an error occurred during the operation. The callback should check this error code before processing the parameters. If the error code is other than **SLP_OK**, then the API library can choose to terminate the outstanding operation.

**pvCookie**

Memory passed down from the client code that called the original API function, starting the operation. Can be NULL.

Return Values

The client code should return **SLP_TRUE** if more data is desired; otherwise **SLP_FALSE** is returned.

Related Information


The `/etc/slp.conf` File.

The Service Location Protocol (SLP) API

SLPUnescape Subroutine

Purpose

Processes an input string and unescapes any characters reserved for SLP.

Syntax

```c
SLPError SLPUnescape(const char* pcInbuf, char** ppcOutBuf, SLPBoolean isTag);
```

Description

The **SLPUnescape** subroutine processes the input string in `pcInbuf` and unescapes any characters reserved for SLP. If the `isTag` parameter is **SLP_TRUE**, **SLPUnescape** looks for bad tag characters and signals an error if any are found by returning the **SLP_PARSE_ERROR** code. No transformation is performed if the input string is opaque. The results are put into a buffer allocated by the API library and returned in the `ppcOutBuf` parameter. This buffer should be deallocated using **SLPFree()** when the memory is no longer needed.

Parameters

**pcInbuf**

Pointer to the input buffer to process for escape characters.

**ppcOutBuf**

Pointer to a pointer for the output buffer with the characters reserved for SLP escaped. Must be freed using **SLPFree()** when the memory is no longer needed.
Return Values
The `SLPUnescape` subroutine returns `SLP_PARSE_ERROR` if any characters are bad tag characters and
the `isTag` flag is true; otherwise, it returns `SLP_OK`, or the appropriate error code if another error occurs.

Related Information
- “SLPAttrCallback Subroutine” on page 214
- “SLPClose Subroutine” on page 215
- “SLPEscape Subroutine” on page 216
- “SLPFindAttrs Subroutine” on page 217
- “SLPFindScopes Subroutine” on page 218
- “SLPFindSys Subroutine” on page 218
- “SLPFindSysTypes Subroutine” on page 219
- “SLPFree Subroutine” on page 220
- “SLPGetProperty Subroutine” on page 221
- “SLPOpen Subroutine” on page 222
- “SLPParseSrvURL Subroutine” on page 223
- “SLPSrvTypeCallback Subroutine” on page 223
- “SLPSrvURLCallback Subroutine” on page 224

The `/etc/slp.conf` File

The Service Location Protocol (SLP) API

socket Subroutine

Purpose
Creates an end point for communication and returns a descriptor.

Library
Standard C Library (`libc.a`)

Syntax
```c
#include <sys/types.h>
#include <sys/socket.h>
#include <sys/socketvar.h>
int socket (AddressFamily, Type, Protocol);
int AddressFamily, Type, Protocol;
```

Description
The `socket` subroutine creates a socket in the specified `AddressFamily` and of the specified type. A
protocol can be specified or assigned by the system. If the protocol is left unspecified (a value of 0), the
system selects an appropriate protocol from those protocols in the address family that can be used to
support the requested socket type.

The `socket` subroutine returns a descriptor (an integer) that can be used in later subroutines that operate
on sockets.

Socket level options control socket operations. The `getsockopt` and `setsockopt` subroutines are used to
get and set these options, which are defined in the `/usr/include/sys/socket.h` file.
Parameters

AddressFamily

Specifies an address family with which addresses specified in later socket operations should be interpreted. The `/usr/include/sys/socket.h` file contains the definitions of the address families. Commonly used families are:

- **AF_UNIX**
  Denotes the operating system path names.

- **AF_INET**
  Denotes the ARPA Internet addresses.

- **AF_NS**
  Denotes the XEROX Network Systems protocol.

Type

Specifies the semantics of communication. The `/usr/include/sys/socket.h` file defines the socket types. The operating system supports the following types:

- **SOCK_STREAM**
  Provides sequenced, two-way byte streams with a transmission mechanism for out-of-band data.

- **SOCK_DGRAM**
  Provides datagrams, which are connectionless messages of a fixed maximum length (usually short).

- **SOCK_RAW**
  Provides access to internal network protocols and interfaces. This type of socket is available only to the root user, or to non-root users who have the `CAP_NUMA_ATTACH` capability. (For non-root raw socket access, the `CAP_NUMA_ATTACH` capability, along with `CAP_PROPAGATE`, is assigned using the `chuser` command. For more information about the `chuser` command, see `chuser Command` in *AIX 5L Version 5.3 Commands Reference, Volume 1.*)

- **SOCK_SEQPACKET**
  Provides sequenced, reliable, and unduplicated flow of information. This type of socket is used for UDP-style socket creation in case of Stream Control Transmission Protocol.

Protocol

Specifies a particular protocol to be used with the socket. Specifying the Protocol parameter of 0 causes the `socket` subroutine to default to the typical protocol for the requested type of returned socket. For SCTP sockets, the protocol parameter will be IPPROTO_SCTP.

Return Values

Upon successful completion, the `socket` subroutine returns an integer (the socket descriptor).

If the `socket` subroutine is unsuccessful, the subroutine handler performs the following functions:

- Returns a value of -1 to the calling program.
- Moves an error code, indicating the specific error, into the `errno` global variable. For further explanation of the `errno` variable see `Error Notification Object Class` in *AIX 5L Version 5.3 General Programming Concepts: Writing and Debugging Programs.*

Error Codes

The `socket` subroutine is unsuccessful if any of the following errors occurs:

<table>
<thead>
<tr>
<th>Error</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EAFNOSUPPORT</td>
<td>The addresses in the specified address family cannot be used with this socket.</td>
</tr>
<tr>
<td>EMFILE</td>
<td>The per-process descriptor table is full.</td>
</tr>
<tr>
<td>ENOBUFS</td>
<td>Insufficient resources were available in the system to complete the call.</td>
</tr>
<tr>
<td>ESOCKTNOSUPPORT</td>
<td>The socket in the specified address family is not supported.</td>
</tr>
</tbody>
</table>
Examples
The following program fragment illustrates the use of the `socket` subroutine to create a datagram socket for on-machine use:

```c
s = socket(AF_UNIX, SOCK_DGRAM,0);
```

Implementation Specifics
The `socket` subroutine is part of Base Operating System (BOS) Runtime.

The `socket` subroutine can be compiled with `COMPAT_43` defined. This will make the `sockaddr` structure BSD 4.3 compatible. For more details refer to `socket.h`.

Related Information
The `accept`, `bind`, `connect`, `getsockname`, `getsockopt`, `ioctl`, `listen`, `recv`, `recvfrom`, `recvmsg`, `select`, `send`, `sendmsg`, `sendto`, `setsockopt`, `shutdown`, `socketpair`, `select`, `send`, `sendmsg`, `sendto`, `setsockopt`, `shutdown`, `socketpair` subroutine.

socketpair Subroutine

Purpose
Creates a pair of connected sockets.

Library
Standard C Library (`libc.a`)

Syntax
```c
#include <sys/socket.h>

int socketpair (Domain, Type, Protocol, SocketVector[0])
int Domain, Type, Protocol;
int SocketVector[2];
```

Description
The `socketpair` subroutine creates an unnamed pair of connected sockets in a specified domain, of a specified type, and using the optionally specified protocol. The two sockets are identical.

Note: Create sockets with this subroutine only in the `AF_UNIX` protocol family.

The descriptors used in referencing the new sockets are returned in the `SocketVector[0]` and `SocketVector[1]` parameters.

The `/usr/include/sys/socket.h` file contains the definitions for socket domains, types, and protocols.

All applications containing the `socketpair` subroutine must be compiled with the `_BSD` macro set to a value of 43 or 44. Socket applications must include the BSD `libbsd.a` library.
Parameters

Domain
Specifies the communications domain within which the sockets are created. This subroutine does not create sockets in the Internet domain.

Type
Specifies the communications method, whether SOCK_DGRAM or SOCK_STREAM, that the socket uses.

Protocol
Points to an optional identifier used to specify which standard set of rules (such as UDP/IP and TCP/IP) governs the transfer of data.

SocketVector
Points to a two-element vector that contains the integer descriptors of a pair of created sockets.

Return Values
Upon successful completion, the socketpair subroutine returns a value of 0.

If the socketpair subroutine is unsuccessful, the subroutine handler performs the following functions:
• Returns a value of -1 to the calling program.
• Moves an error code, indicating the specific error, into the errno global variable.

Error Codes
If the socketpair subroutine is unsuccessful, it returns one of the following errors codes:

<table>
<thead>
<tr>
<th>Error</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EAFNOSUPPORT</td>
<td>The addresses in the specified address family cannot be used with this socket.</td>
</tr>
<tr>
<td>EFAULT</td>
<td>The SocketVector parameter is not in a writable part of the user address space.</td>
</tr>
<tr>
<td>EMFILE</td>
<td>This process has too many descriptors in use.</td>
</tr>
<tr>
<td>ENFILE</td>
<td>The maximum number of files allowed are currently open.</td>
</tr>
<tr>
<td>ENOBFS</td>
<td>Insufficient resources were available in the system to perform the operation.</td>
</tr>
<tr>
<td>EOPNOTSUPP</td>
<td>The specified protocol does not allow the creation of socket pairs.</td>
</tr>
<tr>
<td>EPROTONOSUPPORT</td>
<td>The specified protocol cannot be used on this system.</td>
</tr>
<tr>
<td>EPROTOTYPE</td>
<td>The socket type is not supported by the protocol.</td>
</tr>
</tbody>
</table>

Related Information
The socket subroutine.

socks5_getserv Subroutine

Purpose
Return the address of the SOCKSv5 server (if any) to use when connecting to a given destination.

Library
Standard C Library (libc.a)

Syntax
```
#include <stdlib.h>
#include <netinet/in.h>
#include <sys/socket.h>
```


```c
struct sockaddr * socks5_getserv (Dst, DstLen);
struct sockaddr * Dst;
size_t DstLen;
```

## Description

The `socks5_getserv` subroutine determines which (if any) SOCKSv5 server should be used as an intermediary when connecting to the address specified in `Dst`.

The address returned in `Dst` may be IPv4 or IPv6 or some other family. The user should check the address family before using the returned data.

The socket applications can be compiled with `COMPAT_43` defined. This will make the `sockaddr` structure BSD 4.3 compatible. For more details refer to `socket.h`.

### Parameters

- **Dst** Specifies the external address of the target socket to use as a key for looking up the appropriate SOCKSv5 server.
- **DstLength** Specifies the length of the address structure in `Dst`.

### Return Values

- Upon successful lookup, the `socks_getserv` subroutine returns a reference to a sockaddr struct.
- If the `socks5tcp_connect` subroutine is unsuccessful in finding a server, for any reason, a value of NULL is returned. If an error occurred, an error code, indicating the generic system error, is moved into the `errno` global variable.

### Error Codes (placed in `errno`)

The `socks5_getserv` subroutine is unsuccessful if no server is indicated or if any of the following errors occurs:

<table>
<thead>
<tr>
<th>Error</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EAFNOSUPPORT</td>
<td>The addresses in the specified address family cannot be used with this socket.</td>
</tr>
<tr>
<td>EFAULT</td>
<td>The <code>Dst</code> parameter is not in a writable part of the user address space.</td>
</tr>
<tr>
<td>EINVAL</td>
<td>One or more of the specified arguments is invalid.</td>
</tr>
<tr>
<td>ENOMEM</td>
<td>The <code>Dst</code> parameter is not large enough to hold the server address.</td>
</tr>
</tbody>
</table>

### Examples

The following program fragment illustrates the use of the `socks5_getserv` subroutine by a client to request a connection from a server’s socket.

```c
struct sockaddr_in6 dst;

struct sockaddr *srv;

srv = socks5_getserv((struct sockaddr*)&dst, sizeof(dst));

if (srv != NULL) {
    /* Success: srv should be used as the socks5 server */
} else {
    /* Failure: no server could be returned. check errno */
}
```

230 Technical Reference: Communications, Volume 2
Related Information

The socks5tcp_connect subroutine, socks5tcp_bind subroutine, socks5tcp_accept subroutine, socks5udp_associate subroutine, socks5udp_sendto subroutine, /etc/socks5c.conf file, connect subroutine.

Sockets Overview and Understanding Socket Connections in AIX 5L Version 5.3 Communications Programming Concepts.

SOCKS5C_CONFIG Environment Variable in AIX 5L Version 5.3 Files Reference.

/etc/socks5c.conf File

Purpose
Contains mappings between network destinations and SOCKSv5 servers.

Description
The /etc/socks5c.conf file contains basic mappings between network destinations (hosts or networks) and SOCKSv5 servers to use when accessing those destinations. This is an ASCII file that contains records for server mappings. Text following a pound character (‘#’) is ignored until the end of line. Each record appears on a single line and is the following format:

<destination>[/<prefixlength>] <server>[:<port>]

You must separate fields with whitespace. Records are separated by new-line characters. The fields and modifiers in a record have the following values:

destination Specifies a network destination; destination may be either a name fragment or a numeric address (with optional prefixlength). If destination is an address, it may be either IPv4 or IPv6.

prefixlength If specified, indicates the number of leftmost (network order) bits of an address to use when comparing to this record. Only valid if destination is an address. If not specified, all bits are used in comparisons.

server Specifies the SOCKSv5 server associated with destination. If server is “NONE” (must be all uppercase), this record indicates that target addresses matching destination should not use any SOCKSv5 server, but rather be contacted directly.

port If specified, indicates the port to use when contacting server. If not specified, the default of 1080 is assumed.

Note: Server address in IPv6 format must be followed by a port number.

If a name fragment destination is present in /etc/socks5c.conf, all target addresses is SOCKSv5 operations will be converted into hostnames for name comparison (in addition to numeric comparisons with numeric records). The resulting hostname is considered to match if the last characters in the hostname match the specified name fragment.

When using this configuration information to determine the address of the appropriate SOCKSv5 server for a target destination, the “best” match is used. The “best” match is defined as:

destination is numeric Most bits in comparison (i.e. largest prefixlength)

destination is a name fragment Most characters in name fragment.

When both name fragment and numeric addresses are present, all name fragment entries are “better” than numeric address entries.
Two implicit records:
0.0.0.0/0 NONE #All IPv4 destinations; no associated server.
::/0 NONE #All IPv6 destinations; no associated server.

are assumed as defaults for all destinations not specified in /etc/socks5c.conf.

Security
Access Control: This file should grant read (r) access to all users and grant write (w) access only to the root user.

Examples
#Sample socks5c.conf file
9.0.0.0/8 NONE #Direct communication with all hosts in the 9 network.
129.35.0.0/16 sox1.austin.ibm.com
ibm.com NONE #Direct communication will all hosts matching "ibm.com" (e.g. "aguila.austin.ibm.com")

Related Information
The socks5tcp_connect subroutine, socks5tcp_bind subroutine, socks5tcp_accept subroutine, socks5tcp_sendto subroutine, socks5udp_associate subroutine, socks5udp_sendto subroutine, socks5_getserv subroutine, connect subroutine.

socks5tcp_accept Subroutine

Purpose
Awaits an incoming connection to a socket from a previous socks5tcp_bind() call.

Library
Standard C Library (libc.a)

Syntax
#include <stdlib.h>
#include <netinet/in.h>
#include <sys/socket.h>

int socks5tcp_accept(Socket, Dst, DstLen, Svr, SvrLen)
int Socket;
struct sockaddr *Dst;
size_t DstLen;
struct sockaddr *Svr;
size_t SvrLen;

Description
The socks5tcp_accept subroutine blocks until an incoming connection is established on a listening socket that was requested in a previous call to socks5tcp_bind. Upon success, subsequent writes to and reads from Socket will be relayed through Svr.

Socket must be an open socket descriptor of type SOCK_STREAM.

The socket applications can be compiled with COMPAT_43 defined. This will make the sockaddr structure BSD 4.3 compatible. For more details refer to socket.h.
Parameters

*Socket* Specifies the unique name of the socket.

*Dst* If non-NULL, buffer for receiving the address of the remote client which initiated an incoming connection.

*DstLength* Specifies the length of the address structure in *Dst*.

*Svr* If non-NULL, specifies the address of the SOCKSv5 server to use to request the relayed connection; on success, this space will be overwritten with the server-side address of the incoming connection.

*SvrLength* Specifies the length of the address structure in *Svr*.

Return Values

Upon successful completion, the `socks5tcp_accept` subroutine returns a value of 0, and modifies *Dst* and *Svr* to reflect the actual endpoints of the incoming external socket.

If the `socks5tcp_accept` subroutine is unsuccessful, the system handler performs the following functions:

- Returns a value of -1 to the calling program.
- Moves an error code, indicating the generic system error, into the `errno` global variable.
- Moves an error code, indicating the specific SOCKSv5 error, into the `socks5_errno` global variable.

Error Codes (placed in `errno`; inherited from underlying call to `connect()`)

The `socks5tcp_bindaccept` subroutine is unsuccessful if any of the following errors occurs:

<table>
<thead>
<tr>
<th>Error</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EBADF</td>
<td>The <em>Socket</em> parameter is not valid.</td>
</tr>
<tr>
<td>EAFNOSUPPORT</td>
<td>The addresses in the specified address family cannot be used with this socket.</td>
</tr>
<tr>
<td>ENETUNREACH</td>
<td>No route to the network or host is present.</td>
</tr>
<tr>
<td>EFAULT</td>
<td>The <em>Dst</em> or <em>Svr</em> parameter is not in a writable part of the user address space.</td>
</tr>
<tr>
<td>EINVAL</td>
<td>One or more of the specified arguments is invalid.</td>
</tr>
<tr>
<td>ENETDOWN</td>
<td>The specified physical network is down.</td>
</tr>
<tr>
<td>ENOSPC</td>
<td>There is no space left on a device or system table.</td>
</tr>
<tr>
<td>ENOTCONN</td>
<td>The socket could not be connected.</td>
</tr>
</tbody>
</table>

Error Codes (placed in `socks5_errno`; SOCKSv5-specific errors)

The `socks5tcp_connect` subroutine is unsuccessful if any of the following errors occurs:

<table>
<thead>
<tr>
<th>Error</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>S5_ESRVFAIL</td>
<td>General SOCKSv5 server failure.</td>
</tr>
<tr>
<td>S5_EPERM</td>
<td>SOCKSv5 server ruleset rejection.</td>
</tr>
<tr>
<td>S5_ENETUNREACH</td>
<td>SOCKSv5 server could not reach target network.</td>
</tr>
<tr>
<td>S5_EHOSTUNREACH</td>
<td>SOCKSv5 server could not reach target host.</td>
</tr>
<tr>
<td>S5_ECONNREFUSED</td>
<td>SOCKSv5 server connection request refused by target host.</td>
</tr>
<tr>
<td>S5_ETIMEDOUT</td>
<td>SOCKSv5 server connection failure due to TTL expiry.</td>
</tr>
<tr>
<td>S5_EOPNOTSUPP</td>
<td>Command not supported by SOCKSv5 server.</td>
</tr>
<tr>
<td>S5_EAFNOSUPPORT</td>
<td>Address family not supported by SOCKSv5 server.</td>
</tr>
<tr>
<td>S5_EADDRINUSE</td>
<td>Requested bind address is already in use (at the SOCKSv5 server).</td>
</tr>
<tr>
<td>S5_ENOSERV</td>
<td>No server found.</td>
</tr>
</tbody>
</table>
Examples

The following program fragment illustrates the use of the `socks5tcp_accept` and `socks5tcp_bind` subroutines by a client to request a listening socket from a server and wait for an incoming connection on the server side.

```c
struct sockaddr_in svr;
struct sockaddr_in dst;
.
.
socks5tcp_bind(s,(struct sockaddr*)&dst, sizeof(dst), (struct sockaddr*)&svr, sizeof(svr), &res, sizeof(svr));
.
.
socks5tcp_accept(s, (struct sockaddr *)&dst, sizeof(dst), (struct sockaddr *)&svr, sizeof(svr));
```

Related Information

The `socks5tcp_connect` subroutine, `socks5tcp_bind` subroutine, `socks5udp_associate` subroutine, `socks5udp_sendto` subroutine, `socks5_getserv` subroutine, `/etc/socks5c.conf` file, `accept` subroutine, `bind` subroutine, `getsockname` subroutine, `send` subroutine, `socket` subroutine.

Initiating UNIX Stream Connections Example Program, Sockets Overview and Understanding Socket Connections in AIX 5L Version 5.3 Communications Programming Concepts.

SOCK5C_CONFIG Environment Variable in AIX 5L Version 5.3 Files Reference.

socks5tcp_bind Subroutine

Purpose

Connect to a SOCKSv5 server and request a listening socket for incoming remote connections.

Library

Standard C Library (`libc.a`)

Syntax

```c
#include <stdlib.h>
#include <netinet/in.h>
#include <sys/socket.h>

int socks5tcp_bind(Socket, Dst, DstLen, Svr, SvrLen)
int Socket;
struct sockaddr *Dst;
size_t DstLen;
struct sockaddr *Svr;
size_t SvrLen;
```

Description

The `socks5tcp_bind` subroutine requests a listening socket on the SOCKSv5 server specified in `Svr`, in preparation for an incoming connection from a remote destination, specified by `Dst`. Upon success, `Svr` will be overwritten with the actual address of the newly bound listening socket, and `Socket` may be used in a subsequent call to `socks5tcp_accept`.

`Socket` must be an open socket descriptor of type `SOCK_STREAM`.

The socket applications can be compiled with `COMPAT_43` defined. This will make the `sockaddr` structure BSD 4.3 compatible. For more details refer to `socket.h`. 
Parameters

- **Socket** Specifies the unique name of the socket.
- **Dst** Specifies the address of the SOCKSv5 server to use to request the relayed connection; on success, this space will be overwritten with the actual bound address on the server.
- **DstLength** Specifies the length of the address structure in **Dst**.
- **Svr** If non-NULL, specifies the address of the SOCKSv5 server to use to request the relayed connection; on success, this space will be overwritten with the actual bound address on the server.
- **SvrLength** Specifies the length of the address structure in **Svr**.

Return Values

Upon successful completion, the **socks5tcp_bind** subroutine returns a value of 0, and modifies **Svr** to reflect the actual address of the newly bound listener socket.

If the **socks5tcp_bind** subroutine is unsuccessful, the system handler performs the following functions:

- Returns a value of -1 to the calling program.
- Moves an error code, indicating the generic system error, into the **errno** global variable.
- Moves an error code, indicating the specific SOCKSv5 error, into the **socks5_errno** global variable.

Error Codes (placed in **errno**; inherited from underlying call to connect())

The **socks5tcp_bindaccept** subroutine is unsuccessful if any of the following errors occurs:

<table>
<thead>
<tr>
<th>Error</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EBADF</td>
<td>The <strong>Socket</strong> parameter is not valid.</td>
</tr>
<tr>
<td>ENOTSOCK</td>
<td>The <strong>Socket</strong> parameter refers to a file, not a socket.</td>
</tr>
<tr>
<td>EADDRNOTAVAIL</td>
<td>The specified address is not available from the local machine.</td>
</tr>
<tr>
<td>EAFNOSUPPORT</td>
<td>The addresses in the specified address family cannot be used with this socket.</td>
</tr>
<tr>
<td>EISCONN</td>
<td>The socket is already connected.</td>
</tr>
<tr>
<td>ETIMEDOUT</td>
<td>The establishment of a connection timed out before a connection was made.</td>
</tr>
<tr>
<td>ECONNREFUSED</td>
<td>The attempt to connect was rejected.</td>
</tr>
<tr>
<td>ENETUNREACH</td>
<td>No route to the network or host is present.</td>
</tr>
<tr>
<td>EADDRINUSE</td>
<td>The specified address is already in use.</td>
</tr>
<tr>
<td>EFAULT</td>
<td>The <strong>Address</strong> parameter is not in a writable part of the user address space.</td>
</tr>
<tr>
<td>EINPROGRESS</td>
<td>The socket is marked as nonblocking. The connection cannot be immediately completed. The application program can select the socke for writing during the connection process.</td>
</tr>
<tr>
<td>EINVAL</td>
<td>One or more of the specified arguments is invalid.</td>
</tr>
<tr>
<td>ENETDOWN</td>
<td>The specified physical network is down.</td>
</tr>
<tr>
<td>ENOSPC</td>
<td>There is no space left on a device or system table.</td>
</tr>
<tr>
<td>ENOTCONN</td>
<td>The socket could not be connected.</td>
</tr>
</tbody>
</table>

Error Codes (placed in **socks5_errno**; SOCKSv5-specific errors)

The **socks5tcp_connect** subroutine is unsuccessful if any of the following errors occurs:

<table>
<thead>
<tr>
<th>Error</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>S5_ESRVFAIL</td>
<td>General SOCKSv5 server failure.</td>
</tr>
<tr>
<td>S5_EPERM</td>
<td>SOCKSv5 server ruleset rejection.</td>
</tr>
<tr>
<td>S5_ENETUNREACH</td>
<td>SOCKSv5 server could not reach target network.</td>
</tr>
<tr>
<td>S5_EHOSTUNREACH</td>
<td>SOCKSv5 server could not reach target host.</td>
</tr>
<tr>
<td>S5_ECONNREFUSED</td>
<td>SOCKSv5 server connection request refused by target host.</td>
</tr>
<tr>
<td>S5_ETIMEDOUT</td>
<td>SOCKSv5 server connection failure due to TTL expiry.</td>
</tr>
<tr>
<td>S5_EOPNOTSUPP</td>
<td>Command not supported by SOCKSv5 server.</td>
</tr>
</tbody>
</table>
Error Description
S5_EAFNOSUPPORT Address family not supported by SOCKSv5 server.
S5_EADDRINUSE Requested bind address is already in use (at the SOCKSv5 server).
S5_ENOERV No server found.

Examples
The following program fragment illustrates the use of the `socks5tcp_bind` subroutine by a client to request a listening socket from a server.

```c
struct sockaddr_in svr;
struct sockaddr_in dst;
.
.
socks5tcp_bind(s, (struct sockaddr *)&dst, sizeof(dst), (struct sockaddr *)&svr, sizeof(svr));
```

Related Information
The `socks5tcp_accept` subroutine, `socks5tcp_connect` subroutine, `socks5_getserv` subroutine, `/etc/socks5c.conf` file, `accept` subroutine, `bind` subroutine, `getsockname` subroutine, `send` subroutine, `socket` subroutine.

Sockets Overview and Understanding Socket Connections in AIX 5L Version 5.3 Communications Programming Concepts.

SOCKS5C_CONFIG Environment Variable in AIX 5L Version 5.3 Files Reference.

socks5tcp_connect Subroutine

Purpose
Connect to a SOCKSv5 server and request a connection to an external destination.

Library
Standard C Library (`libc.a`)

Syntax
```c
#include <stdlib.h>
#include <netinet/in.h>
#include <sys/socket.h>

int socks5tcp_connect (Socket, Dst, DstLen, Svr, SvrLen)
int Socket;
struct sockaddr *Dst;
size_t DstLen;
struct sockaddr *Svr;
size_t SvrLen;
```

Description
The `socks5tcp_connect` subroutine requests a connection to `Dst` from the SOCKSv5 server specified in `Svr`. If successful, `Dst` and `Svr` will be overwritten with the actual addresses of the external connection and subsequent writes to and reads from `Socket` will be relayed through `Svr`.

`Socket` must be an open socket descriptor of type SOCK_STREAM; `Dst` and `Svr` may be either IPv4 or IPv6 addresses.
The socket applications can be compiled with COMPAT_43 defined. This will make the sockaddr structure BSD 4.3 compatible. For more details refer to socket.h.

Parameters

- **Socket**: Specifies the unique name of the socket.
- **Dst**: Specifies the external address of the target socket to which the SOCKSv5 server will attempt to connect.
- **DstLength**: Specifies the length of the address structure in Dst.
- **Svr**: If non-NULL, specifies the address of the SOCKSv5 server to use to request the relayed connection.
- **SvrLength**: Specifies the length of the address structure in Svr.

Return Values

Upon successful completion, the socks5tcp_connect subroutine returns a value of 0, and modifies Dst and Svr to reflect the actual endpoints of the created external socket.

If the socks5tcp_connect subroutine is unsuccessful, the system handler performs the following functions:

- Returns a value of -1 to the calling program.
- Moves an error code, indicating the generic system error, into the errno global variable.
- Moves an error code, indicating the specific SOCKSv5 error, into the socks5_errno global variable.
- Dst and Svr are left unmodified.

Error Codes (placed in errno; inherited from underlying call to connect())

The socks5tcp_connect subroutine is unsuccessful if any of the following errors occurs:

<table>
<thead>
<tr>
<th>Error</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EBADF</td>
<td>The Socket parameter is not valid.</td>
</tr>
<tr>
<td>ENOTSOCK</td>
<td>The Socket parameter refers to a file, not a socket.</td>
</tr>
<tr>
<td>EADDRNOTAVAIL</td>
<td>The specified address is not available from the local machine.</td>
</tr>
<tr>
<td>EAFNOSUPPORT</td>
<td>The addresses in the specified address family cannot be used with this socket.</td>
</tr>
<tr>
<td>EISCONN</td>
<td>The socket is already connected.</td>
</tr>
<tr>
<td>ETIMEDOUT</td>
<td>The establishment of a connection timed out before a connection was made.</td>
</tr>
<tr>
<td>ECONNREFUSED</td>
<td>The attempt to connect was rejected.</td>
</tr>
<tr>
<td>ENETUNREACH</td>
<td>No route to the network or host is present.</td>
</tr>
<tr>
<td>EADDRINUSE</td>
<td>The specified address is already in use.</td>
</tr>
<tr>
<td>EFAULT</td>
<td>The Address parameter is not in a writable part of the user address space.</td>
</tr>
<tr>
<td>EINVAL</td>
<td>The socket is marked as nonblocking. The connection cannot be immediately completed. The application program can select the socket for writing during the connection process.</td>
</tr>
<tr>
<td>ENINVAL</td>
<td>One or more of the specified arguments is invalid.</td>
</tr>
<tr>
<td>ENETDOWN</td>
<td>The specified physical network is down.</td>
</tr>
<tr>
<td>ENOSPC</td>
<td>There is no space left on a device or system table.</td>
</tr>
<tr>
<td>ENOTCONN</td>
<td>The socket could not be connected.</td>
</tr>
</tbody>
</table>

Error Codes (placed in socks5_errno; SOCKSv5-specific errors)

The socks5tcp_connect subroutine is unsuccessful if any of the following errors occurs:

<table>
<thead>
<tr>
<th>Error</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>S5_ESRVFAIL</td>
<td>General SOCKSv5 server failure.</td>
</tr>
<tr>
<td>S5_EPERM</td>
<td>SOCKSv5 server ruleset rejection.</td>
</tr>
</tbody>
</table>
Error Description
S5_ENETUNREACH SOCKSv5 server could not reach target network.
S5_EHOSTUNREACH SOCKSv5 server could not reach target host.
S5_ECONNREFUSED SOCKSv5 server connection request refused by target host.
S5_ETIMEDOUT SOCKSv5 server connection failure due to TTL expiry.
S5_EOPNOTSUPP Command not supported by SOCKSv5 server.
S5_EAFNOSUPPORT Address family not supported by SOCKSv5 server.
S5_ENOSERV No server found.

Examples
The following program fragment illustrates the use of the `socks5tcp_connect` subroutine by a client to request a connection from a server's socket.

```c
struct sockaddr_in svr;
struct sockaddr_in6 dst;
.
.
socks5tcp_connect(s,(struct sockaddr*)&dst, sizeof(dst), (struct sockaddr *)&svr, sizeof(svr));
```

Related Information
The `socks5_getserv` subroutine, `/etc/socks5c.conf` file, `connect` subroutine, `accept` subroutine, `bind` subroutine, `getsockname` subroutine, `send` subroutine, `socket` subroutine.

Initiating UNIX Stream Connections Example Program, Sockets Overview and Understanding Socket Connections in AIX 5L Version 5.3 Communications Programming Concepts.

SOCKS5C_CONFIG Environment Variable in AIX 5L Version 5.3 Files Reference.

`socks5udp_associate` Subroutine

Purpose
Connects to a SOCKSv5 server, and requests a UDP association for subsequent UDP socket communications.

Library
Standard C Library (`libc.a`)

Syntax
```
#include <stdlib.h>
#include <netinet/in.h>
#include <sys/socket.h>

int socks5udp_associate (Socket, Dst, DstLen, Svr, SvrLen)

int Socket;
const struct sockaddr *Dst;
size_t DstLen;
const struct sockaddr *Svr;
size_t SvrLen;
```

Description
The `socks5udp_associate` subroutine requests a UDP association for `Dst` on the SOCKSv5 server specified in `Svr`. Upon success, `Dst` is overwritten with a rendezvous address to which subsequent UDP packets should be sent for relay by `Svr`.

238 Technical Reference: Communications, Volume 2
Socket must be an open socket descriptor of type SOCK_STREAM; Dst and Svr may be either IPv4 or IPv6 addresses.

Note that Socket cannot be used to send subsequent UDP packets (a second socket of type SOCK_DGRAM must be created).

The socket applications can be compiled with COMPAT_43 defined. This will make the sockaddr structure BSD 4.3 compatible. For more details refer to socket.h.

Parameters

Socket Specifies the unique name of the socket.
Dst Specifies the external address of the target socket to which the SOCKSv5 client expects to send UDP packets.
DstLength Specifies the length of the address structure in Dst.
Svr Specifies the address of the SOCKSv5 server to use to request the association.
SvrLength Specifies the length of the address structure in Svr.

Return Values

Upon successful completion, the socks5udp_associate subroutine returns a value of 0 and overwrites Dst with the rendezvous address.

If the socks5udp_associate subroutine is unsuccessful, the system handler performs the following functions:

- Returns a value of -1 to the calling program.
- Moves an error code, indicating the generic system error, into the errno global variable.
- Moves an error code, indicating the specific SOCKSv5 error, into the socks5_errno global variable.

Error Codes (placed in errno; inherited from underlying call to connect())

The socks5udp_associate subroutine is unsuccessful if any of the following errors occurs:

<table>
<thead>
<tr>
<th>Error</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EBADF</td>
<td>The Socket parameter is not valid.</td>
</tr>
<tr>
<td>ENOTSOCK</td>
<td>The Socket parameter refers to a file, not a socket.</td>
</tr>
<tr>
<td>EADDRNOTAVAIL</td>
<td>The specified address is not available from the local machine.</td>
</tr>
<tr>
<td>EAFNOSUPPORT</td>
<td>The addresses in the specified address family cannot be used with this socket.</td>
</tr>
<tr>
<td>EISCONN</td>
<td>The socket is already connected.</td>
</tr>
<tr>
<td>ETIMEDOUT</td>
<td>The establishment of a connection timed out before a connection was made.</td>
</tr>
<tr>
<td>ECONNREFUSED</td>
<td>The attempt to connect was rejected.</td>
</tr>
<tr>
<td>ENETUNREACH</td>
<td>No route to the network or host is present.</td>
</tr>
<tr>
<td>EADDRINUSE</td>
<td>The specified address is already in use.</td>
</tr>
<tr>
<td>EFAULT</td>
<td>The Address parameter is not in a writable part of the user address space.</td>
</tr>
<tr>
<td>EINPROGRESS</td>
<td>The socket is marked as nonblocking. The connection cannot be immediately completed.</td>
</tr>
<tr>
<td>EINVAL</td>
<td>One or more of the specified arguments is invalid.</td>
</tr>
<tr>
<td>ENETDOWN</td>
<td>The specified physical network is down.</td>
</tr>
<tr>
<td>ENOSPC</td>
<td>There is no space left on a device or system table.</td>
</tr>
<tr>
<td>ENOTCONN</td>
<td>The socket could not be connected.</td>
</tr>
</tbody>
</table>
Error Codes (placed in socks5_errno; SOCKSv5-specific errors)

The **socks5tcp_connect** subroutine is unsuccessful if any of the following errors occurs:

<table>
<thead>
<tr>
<th>Error</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>S5_ESRVFAIL</td>
<td>General SOCKSv5 server failure.</td>
</tr>
<tr>
<td>S5_EPERM</td>
<td>SOCKSv5 server ruleset rejection.</td>
</tr>
<tr>
<td>S5_ENETUNREACH</td>
<td>SOCKSv5 server could not reach target network.</td>
</tr>
<tr>
<td>S5_EHOSTUNREACH</td>
<td>SOCKSv5 server could not reach target host.</td>
</tr>
<tr>
<td>S5_ECONNREFUSED</td>
<td>SOCKSv5 server connection request refused by target host.</td>
</tr>
<tr>
<td>S5_ETIMEDOUT</td>
<td>SOCKSv5 server connection failure due to TTL expiry.</td>
</tr>
<tr>
<td>S5_EOPNOTSUPP</td>
<td>Command not supported by SOCKSv5 server.</td>
</tr>
<tr>
<td>S5_EAFNOSUPPORT</td>
<td>Address family not supported by SOCKSv5 server.</td>
</tr>
<tr>
<td>S5_ENOSERV</td>
<td>No server found.</td>
</tr>
</tbody>
</table>

Examples

The following program fragment illustrates the use of the **socks5udp_associate** subroutine by a client to request an association on a server.

```c
struct sockaddr_in svr;
struct sockaddr_in6 dst;
.
.
socks5udp_associate(s,(struct sockaddr*)&dst, sizeof(dst), (struct sockaddr *)&svr, sizeof(svr));
```

Related Information

The **socks5udp_sendto** subroutine, **socks5_getserv** subroutine, **/etc/socks5c.conf** file, **connect** subroutine, **accept** subroutine, **bind** subroutine, **getsockopt** subroutine, **send** subroutine, **socket** subroutine.

**Initiating UNIX Stream Connections Example Program**, **Sockets Overview**, and **Understanding Socket Connections** in *AIX 5L Version 5.3 Communications Programming Concepts*.

**SOCKS5C_CONFIG Environment Variable** in *AIX 5L Version 5.3 Files Reference*.

**socks5udp_sendto Subroutine**

**Purpose**

Send UDP packets through a SOCKSv5 server.

**Library**

Standard C Library (*libc.a*)

**Syntax**

```c
#include <stdlib.h>
#include <netinet/in.h>
#include <sys/socket.h>

int socks5udp_sendto (Socket, Message, MsgLen, Flags, Dst, DstLen, Svr, SvrLen) 
int Socket;
void *Message;
size_t MsgLen;
int Flags;
```
struct sockaddr *Dst;
size_t DstLen;
struct sockaddr *Svr;
size_t SrvLen;

Description
The socks5udp_sendto subroutine sends a UDP packet to Svr for relay to Dst. Svr must be the rendezvous address returned from a previous call to socks5udp_associate.

Socket must be an open socket descriptor of type SOCK_DGRAM; Dst and Svr may be either IPv4 or IPv6 addresses.

The socket applications can be compiled with COMPAT_43 defined. This will make the sockaddr structure BSD 4.3 compatible. For more details refer to socket.h.

Parameters
Socket Specifies the unique name of the socket.
Message Specifies the address containing the message to be sent.
MsgLen Specifies the size of the message in bytes.
Flags Allows the sender to control the message transmission. See the description in the sendto subroutine for more specific details.
Dst Specifies the external address to which the SOCKSv5 server will attempt to relay the UDP packet.
DstLength Specifies the length of the address structure in Dst.
Svr Specifies the address of the SOCKSv5 server to send the UDP packet for relay.
SvrLength Specifies the length of the address structure in Svr.

Return Values
Upon successful completion, the socks5udp_sendto subroutine returns a value of 0.

If the socks5udp_sendto subroutine is unsuccessful, the system handler performs the following functions:
• Returns a value of -1 to the calling program.
• Moves an error code, indicating the generic system error, into the errno global variable.
• Moves an error code, indicating the specific SOCKSv5 error, into the socks5_errno global variable.

Error Codes (placed in errno; inherited from underlying call to sendto())
The socks5tcp_connect subroutine is unsuccessful if any of the following errors occurs:

<table>
<thead>
<tr>
<th>Error</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EBADF</td>
<td>The Socket parameter is not valid.</td>
</tr>
<tr>
<td>ENOTSOCK</td>
<td>The Socket parameter refers to a file, not a socket.</td>
</tr>
<tr>
<td>EAFNOSUPPORT</td>
<td>The addresses in the specified address family cannot be used with this socket.</td>
</tr>
<tr>
<td>ENETUNREACH</td>
<td>No route to the network or host is present.</td>
</tr>
<tr>
<td>EINVAL</td>
<td>One or more of the specified arguments is invalid.</td>
</tr>
<tr>
<td>ENETDOWN</td>
<td>The specified physical network is down.</td>
</tr>
<tr>
<td>ENOSPC</td>
<td>There is no space left on a device or system table.</td>
</tr>
</tbody>
</table>
Error Codes (placed in socks5_errno; SOCKSv5-specific errors)
The `socks5tcp_connect` subroutine is unsuccessful if any of the following errors occurs:

<table>
<thead>
<tr>
<th>Error</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>S5_ESRVFAIL</td>
<td>General SOCKSv5 server failure.</td>
</tr>
<tr>
<td>S5_EPERM</td>
<td>SOCKSv5 server ruleset rejection.</td>
</tr>
<tr>
<td>S5_ENETUNREACH</td>
<td>SOCKSv5 server could not reach target network.</td>
</tr>
<tr>
<td>S5_EHOSTUNREACH</td>
<td>SOCKSv5 server could not reach target host.</td>
</tr>
<tr>
<td>S5_ECONNREFUSED</td>
<td>SOCKSv5 server connection request refused by target host.</td>
</tr>
<tr>
<td>S5_ETIMEDOUT</td>
<td>SOCKSv5 server connection failure due to TTL expiry.</td>
</tr>
<tr>
<td>S5_EOPNOTSUPP</td>
<td>Command not supported by SOCKSv5 server.</td>
</tr>
<tr>
<td>S5_EAFNOSUPPORT</td>
<td>Address family not supported by SOCKSv5 server.</td>
</tr>
<tr>
<td>S5_ENOSERV</td>
<td>No server found.</td>
</tr>
</tbody>
</table>

Examples
The following program fragment illustrates the use of the `socks5udp_sendto` subroutine by a client to request a connection from a server’s socket.

```c
void *message;
size_t msglen;
int flags;
struct sockaddr_in svr;
struct sockaddr_in6 dst;
.
.
socks5udp_associate(s,(struct sockaddr*)&dst,
sizeof(dst), (struct sockaddr *)&svr,
sizeof(svr));
.
.
socks5udp_sendto(s, message, msglen, flags (struct sockaddr*)&dst,
sizeof(dst), (struct sockaddr *)&svr, sizeof(svr));
```

Related Information
The [socks5udp_associate](#) subroutine, [socks5_getserv](#) subroutine, `/etc/socks5c.conf` file, `bind` subroutine, [getsockname](#) subroutine, `sendto` subroutine, `socket` subroutine.

Initiating UNIX Stream Connections Example Program, Sockets Overview and Understanding Socket Connections in AIX 5L Version 5.3 Communications Programming Concepts.

**SOCKS5C_CONFIG Environment Variable** in AIX 5L Version 5.3 Files Reference.

splice Subroutine

**Purpose**
Let the protocol stack manage two sockets that use TCP.

**Syntax**

```c
#include <sys/types.h>
#include <sys/socket.h>

int splice(socket1, socket2, flags)
int socket1, socket2
int flags
```
Description
The **splice** subroutine will let TCP manage two sockets that are in connected state thus relieving the caller from moving data from one socket to another. After the **splice** subroutine returns successfully, the caller needs to close the two sockets.

The two sockets should be of type **SOCK_STREAM** and protocol **IPPROTO_TCP**. Specifying a protocol of zero will also work.

Parameters

- **socket1**, **socket2**
  Specifies a socket that had gone through a successful connect() or accept().

- **flags**
  Set to zero. Currently ignored.

Return Values

- **0**
  Indicates a successful completion.

- **-1**
  Indicates an error. The specific error is indicated by errno.

Error Codes

- **EBADF**
  **socket1** or **socket2** is not valid.

- **ENOTSOCK**
  **socket1** or **socket2** refers to a file, not a socket.

- **EOPNOTSUPP**
  **socket1** or **socket2** is not of type **SOCK_STREAM**.

- **EINVAL**
  The parameters are invalid.

- **EEXIST**
  **socket1** or **socket2** is already spliced.

- **ENOTCONN**
  **socket1** or **socket2** is not in connected state.

- **EAFNOSUPPORT**
  **socket1** or **socket2** address family is not supported for this subroutine.

WriteFile Subroutine

Purpose

Writes data to a socket.

Syntax

```c
#include <iocp.h>
boolean_t WriteFile (HANDLE FileDescriptor, LPVOID Buffer, DWORD WriteCount, LPDWORD AmountWritten, LPOVERLAPPED Overlapped);
```

Description

The **WriteFile** subroutine writes the number of bytes specified by the **WriteCount** parameter from the buffer indicated by the **Buffer** parameter to the **FileDescriptor** parameter. The number of bytes written is saved in the **AmountWritten** parameter. The **Overlapped** parameter indicates whether or not the operation can be handled asynchronously.

The **WriteFile** subroutine returns a boolean (an integer) indicating whether or not the request has been completed.

The **WriteFile** subroutine is part of the I/O Completion Port (IOCP) kernel extension.
Note: This subroutine only works to a socket file descriptor. It does not work with files or other file descriptors.

Parameters

FileDescriptor Specifies a valid file descriptor obtained from a call to the socket or accept subroutines.

Buffer Specifies the buffer from which the data will be written.

WriteCount Specifies the maximum number of bytes to write.

AmountWritten Specifies the number of bytes written. The parameter is set by the subroutine.

Overlapped Specifies an overlapped structure indicating whether or not the request can be handled asynchronously.

Return Values

Upon successful completion, the WriteFile subroutine returns a boolean indicating the request has been completed.

If the WriteFile subroutine is unsuccessful, the subroutine handler performs the following functions:

- Returns a value of 0 to the calling program.
- Moves an error code, indicating the specific error, into the errno global variable. For further explanation of the errno variable, see the link in the Related Information section of this document.

Error Codes

EINPROGRESS The write request can not be immediately satisfied and will be handled asynchronously. A completion packet will be sent to the associated completion port upon completion.

EAGAIN The write request cannot be immediately satisfied and cannot be handled asynchronously.

EINVAL The FileDescriptor is invalid.

Examples

The following program fragment illustrates the use of the WriteFile subroutine to synchronously write data to a socket:

```c
void buffer;
int amount_written;
b=WriteFile (34, &buffer, 128, &amount_written, NULL);
```

The following program fragment illustrates the use of the WriteFile subroutine to asynchronously write data to a socket:

```c
void buffer;
int amount_written;
LPOVERLAPPED overlapped;
b = ReadFile (34, &buffer, 128, &amount_written, overlapped);
```

Note: The request will only be handled asynchronously if it cannot be immediately satisfied.

Related Information

The "socket Subroutine" on page 226, "accept Subroutine" on page 29, "CreateIoCompletionPort Subroutine" on page 36, "ReadFile Subroutine" on page 155, "GetQueuedCompletionStatus Subroutine" on page 91, and "PostQueuedCompletionStatus Subroutine" on page 150.

For further explanation of the errno variable, see Error Notification Object Class in AIX 5L Version 5.3 General Programming Concepts: Writing and Debugging Programs
Chapter 3. Streams

adjmsg Utility

Purpose
Trims bytes in a message.

Syntax
```c
int adjmsg(mp, len)
    mblk_t *mp;
    register int len;
```

Description
The `adjmsg` utility trims bytes from either the head or tail of the message specified by the `mp` parameter. It only trims bytes across `message blocks` of the same type. The `adjmsg` utility is unsuccessful if the `mp` parameter points to a message containing fewer than `len` bytes of similar type at the message position indicated.

This utility is part of STREAMS Kernel Extensions.

Parameters
- `mp` Specifies the message to be trimmed.
- `len` Specifies the number of bytes to remove from the message.

If the value of the `len` parameter is greater than 0, the `adjmsg` utility removes the number of bytes specified by the `len` parameter from the beginning of the `mp` message. If the value of the `len` parameter is less than 0, it removes `len` bytes from the end of the `mp` message. If the value of the `len` parameter is 0, the `adjmsg` utility does nothing.

Return Values
On successful completion, the `adjmsg` utility returns a value of 1. Otherwise, it returns a value of 0.

Related Information
The `msgdsiz` utility.

List of Streams Programming References and Understanding STREAMS Messages in AIX 5L Version 5.3 Communications Programming Concepts.

allocb Utility

Purpose
Allocates message and data blocks.

Syntax
```c
struct msgb *
allocb(size, pri)
    register int size;
    uint pri;
```

© Copyright IBM Corp. 1997, 2006
Description
The allocb utility allocates blocks for a message. When a message is allocated in this manner, the b_band field of the mblk_t structure is initially set to a value of 0. Modules and drivers can set this field.

This utility is part of STREAMS Kernel Extensions.

Parameters

size Specifies the minimum number of bytes needed in the data buffer.
pri Specifies the relative importance of the allocated blocks to the module. The possible values are:
  - BPRI_LO
  - BPRI_MED
  - BPRI_HI

Return Values

The allocb utility returns a pointer to a message block of type M_DATA in which the data buffer contains at least the number of bytes specified by the size parameter. If a block cannot be allocated as requested, the allocb utility returns a null pointer.

Related Information

The esballoc utility.

List of Streams Programming References and Understanding STREAMS Messages in AIX 5L Version 5.3 Communications Programming Concepts.

backq Utility

Purpose

Returns a pointer to the queue behind a given queue.

Syntax

queue_t *
backq(q)

register queue_t * q;

Description

The backq utility returns a pointer to the queue preceding a given queue. If no such queue exists (as when the q parameter points to a stream end), the backq utility returns a null pointer.

This utility is part of STREAMS Kernel Extensions.

Parameters

q Specifies the queue from which to begin.

Return Values

The backq utility returns a pointer to the queue behind a given queue. If no such queue exists, the backq utility returns a null pointer.
bcanput Utility

Purpose
Tests for flow control in the given priority band.

Syntax
```
int bcanput(q, pri)
register queue_t * q;
unsigned char pri;
```

Description
The bcanput utility provides modules and drivers with a way to test flow control in the given priority band.

The bcanput(q, 0) call is equivalent to the canput(q) call.

This utility is part of STREAMS Kernel Extensions.

Parameters
q Specifies the queue from which to begin to test.
pri Specifies the priority band to test.

Return Values
The bcanput utility returns a value of 1 if a message of the specified priority can be placed on the queue. It returns a value of 0 if the priority band is flow-controlled and sets the QWANTW flag to 0 band (the QB_WANTW flag is set to nonzero band). If the band does not yet exist on the queue in question, it returns a value of 1.

Related Information
List of Streams Programming References and Understanding STREAMS Messages in AIX 5L Version 5.3 Communications Programming Concepts.

bufcall Utility

Purpose
Recover from a failure of the allocb utility.

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

int bufcall(size, pri, func, arg)
uint size;
```
The bufcall utility assists in the event of a block-allocation failure. If the allocb utility returns a null, indicating a message block is not currently available, the bufcall utility may be invoked.

The bufcall utility arranges for (*func)(arg) call to be made when a buffer of the number of bytes specified by the size parameter is available. The pri parameter is as described in the allocb utility. When the function specified by the func parameter is called, it has no user context. It cannot reference the u_area and must return without sleeping. The bufcall utility does not guarantee that the desired buffer will be available when the function specified by the func parameter is called since interrupt processing may acquire it.

On an unsuccessful return, the function specified by the func parameter will never be called. A failure indicates a temporary inability to allocate required internal data structures.

On multiprocessor systems, the function specified by the func parameter should be interrupt-safe. Otherwise, the STR_QSAFETY flag must be set when installing the module or driver with the str_install utility.

This utility is part of STREAMS Kernel Extensions.

Note: The stream.h header file must be the last included header file of each source file using the stream library.

Parameters

- **size**  Specifies the number of bytes needed.
- **pri**  Specifies the relative importance of the allocated blocks to the module. The possible values are:
  - BPRI_LO
  - BPRI_MED
  - BPRI_HI
- **func**  Specifies the function to be called.
- **arg**  Specifies an argument passed to the function.

Return Values

The bufcall utility returns a value of 1 when the request is successfully recorded. Otherwise, it returns a value of 0.

Related Information

The allocb utility, unbufcall utility, and mi_bufcall utility.

List of Streams Programming References and Understanding STREAMS Messages in AIX 5L Version 5.3 Communications Programming Concepts.

Understanding STREAMS Synchronization in AIX 5L Version 5.3 Communications Programming Concepts.
canput Utility

Purpose
Tests for available room in a queue.

Syntax
```c
int canput(q)
register queue_t * q;
```

Description
The **canput** utility determines if there is room left in a message queue. If the queue does not have a service procedure, the **canput** utility searches farther in the same direction in the stream until it finds a queue containing a service procedure. This is the first queue on which the passed message can actually be queued. If such a queue cannot be found, the search terminates on the queue at the end of the stream.

The **canput** utility only takes into account normal data flow control.

This utility is part of STREAMS Kernel Extensions.

Parameters
- `q` Specifies the queue at which to begin the search.

Return Values
The **canput** utility tests the queue found by the search. If the message queue in this queue is not full, the **canput** utility returns a value of 1. This return indicates that a message can be put to the queue. If the message queue is full, the **canput** utility returns a value of 0. In this case, the caller is generally referred to as "blocked".

Related Information
[List of Streams Programming References](#) [Understanding STREAMS Messages](#) in *AIX 5L Version 5.3 Communications Programming Concepts*.

clone Device Driver

Purpose
Opens an unused minor device on another STREAMS driver.

Description
The **clone** device driver is a STREAMS software driver that finds and opens an unused minor device on another STREAMS driver. The minor device passed to the **clone** device driver during the open routine is interpreted as the major device number of another STREAMS driver for which an unused minor device is to be obtained. Each such open operation results in a separate stream to a previously unused minor device.

The **clone** device driver consists solely of an open subroutine. This open function performs all of the necessary work so that subsequent subroutine calls (including the close subroutine) require no further involvement of the **clone** device driver.
The clone device driver generates an ENXIO error, without opening the device, if the minor device number provided does not correspond to a valid major device, or if the driver indicated is not a STREAMS driver.

**Note:** Multiple opens of the same minor device cannot be done through the clone interface. Executing the stat subroutine on the file system node for a cloned device yields a different result from executing the fstat subroutine using a file descriptor obtained from opening the node.

**Related Information**
The close subroutine, fstat subroutine, open subroutine, stat subroutine.

Understanding STREAMS Drivers and Modules and Understanding the log Device Driver in AIX 5L Version 5.3 Communications Programming Concepts.

---

**copyb Utility**

**Purpose**
Copies a message block.

**Syntax**

```c
mblk_t *

copyb(bp)
register mblk_t *bp;
```

**Description**
The copyb utility copies the contents of the message block pointed to by the bp parameter into a newly allocated message block of at least the same size. The copyb utility allocates a new block by calling the allocb utility. All data between the b_rptr and b_wptr pointers of a message block are copied to the new block, and these pointers in the new block are given the same offset values they had in the original message block.

This utility is part of STREAMS Kernel Extensions.

**Parameters**

*bp* Contains a pointer to the message block to be copied.

**Return Values**
On successful completion, the copyb utility returns a pointer to the new message block containing the copied data. Otherwise, it returns a null value. The copy is rounded to a fullword boundary.

**Related Information**
The allocb utility, copymsg utility.

List of Streams Programming References and Understanding STREAMS Messages in AIX 5L Version 5.3 Communications Programming Concepts.

---

**copymsg Utility**

**Purpose**
Copies a message.
Syntax

```c
mblk_t *
copymsg(bp)
register mblk_t * bp;
```

Description

The `copymsg` utility uses the `copyb` utility to copy the message blocks contained in the message pointed to by the `bp` parameter to newly allocated message blocks. It then links the new message blocks to form the new message.

This utility is part of STREAMS Kernel Extensions.

Parameters

`bp`  
Contains a pointer to the message to be copied.

Return Values

On successful compilation, the `copymsg` utility returns a pointer to the new message. Otherwise, it returns a null value.

Related Information

The `copyb` utility.

[List of Streams Programming References](#) and [Understanding STREAMS Messages](#) in *AIX 5L Version 5.3 Communications Programming Concepts*.

---

**datamsg Utility**

Purpose

Tests whether message is a data message.

Syntax

```c
#define datamsg( type ) ((type) == M_DATA | (type) == M_PROTO | (type) == M_PCPROTO | (type) == M_DELAY)
```

Description

The `datamsg` utility determines if a message is a data-type message. It returns a value of True if `mp->b_datap->db_type` (where `mp` is declared as `mblk_t *mp`) is a data-type message. The possible data types are `M_DATA`, `M_PROTO`, `M_PCPROTO`, and `M_DELAY`.

This utility is part of STREAMS Kernel Extensions.

Parameters

`type`  
Specifies acceptable data types.

Return Values

The `datamsg` utility returns a value of True if the message is a data-type message. Otherwise, it returns a value of False.
Related Information

List of Streams Programming References [Understanding STREAMS Messages] in AIX 5L Version 5.3 Communications Programming Concepts.

dlpi STREAMS Driver

Purpose
Provides an interface to the data link provider.

Description
The `dlpi` driver is a STREAMS-based pseudo-driver that provides a Data Link Provider Interface (DLPI) style 2 interface to the data link providers in the operating system.

This driver is part of STREAMS Kernel Extensions.

The data link provider interface supports both the connectionless and connection-oriented modes of service, using the DL_UNITDATA_REQ and DL_UNITDATA_IND primitives. See Data Link Provider Interface Information in AIX 5L Version 5.3 Communications Programming Concepts.

Refer to the “STREAMS Overview” in AIX 5L Version 5.3 Communications Programming Concepts for related publications about the DLPI.

File System Name
Each provider supported by the `dlpi` driver has a unique name in the file system. The supported interfaces are:

<table>
<thead>
<tr>
<th>Driver Name</th>
<th>Interface</th>
</tr>
</thead>
<tbody>
<tr>
<td>/dev/dlpi/en</td>
<td>Ethernet</td>
</tr>
<tr>
<td>/dev/dlpi/et</td>
<td>802.3</td>
</tr>
<tr>
<td>/dev/dlpi/tr</td>
<td>802.5</td>
</tr>
<tr>
<td>/dev/dlpi/fi</td>
<td>FDDI</td>
</tr>
</tbody>
</table>

Physical Point of Attachment
The PPA is used to identify one of several of the same type of interface in the system. It should be a nonnegative integer in the range 0 through 99.

The `dlpi` drivers use the network interface drivers to access the communication adapter drivers. For example, the `/dev/dlpi/tr` file uses the network interface driver `if_tr` (interface `tr0`, `tr1`, `tr2`, . . . ) to access the token-ring adapter driver. The PPA value used attaches the device open instance with the corresponding network interface. For example, opening to the `/dev/dlpi/en` device and then performing an attach with PPA value of 1 attaches this open instance to the network interface `en1`. Therefore, choosing a PPA value selects a network interface. The specific network interface should be active before a certain PPA value is used.

Examples of client and server `dlpi` programs are located in the `/usr/samples/dlpi` directory.

Files

<table>
<thead>
<tr>
<th>File Path</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>/dev/dlpi/*</td>
<td>Contains names of supported protocols.</td>
</tr>
<tr>
<td>/usr/samples/dlpi</td>
<td>Contains client and server <code>dlpi</code> sample programs.</td>
</tr>
</tbody>
</table>
Related Information
The `ifconfig` command, `strload` command.

**Understanding STREAMS Drivers and Modules**, **Obtaining Copies of the DLPI and TPI Specification**, **Data Link Provider Interface Information** in *AIX 5L Version 5.3 Communications Programming Concepts*.

---

**dupb Utility**

**Purpose**
Duplicates a message-block descriptor.

**Syntax**
```c
mblk_t *
dupb(bp)

register mblk_t * bp;
```

**Description**
The `dupb` utility duplicates the `mblk_t` message-block descriptor pointed to by the `bp` parameter by copying the descriptor into a newly allocated message-block descriptor. A message block is formed with the new message-block descriptor pointing to the same data block as the original descriptor. The reference count in the data-block descriptor (`dblk_t`) is then incremented. The `dupb` utility does not copy the data buffer, only the message-block descriptor.

Message blocks that exist on different queues can reference the same data block. In general, if the contents of a message block with a reference count greater than 1 are to be modified, the `copymsg` utility should be used to create a new message block. Only the new message block should be modified to ensure that other references to the original message block are not invalidated by unwanted changes.

This utility is part of STREAMS Kernel Extensions.

**Parameters**
- `bp`  
  Contains a pointer to the message-block descriptor to be copied.

**Return Values**
On successful compilation, the `dupb` utility returns a pointer to the new message block. If the `dupb` utility cannot allocate a new message-block descriptor, it returns a null pointer.

**Related Information**
The `copymsg` utility, `dupmsg` utility.

- List of Streams Programming References and [Understanding STREAMS Messages](#) in *AIX 5L Version 5.3 Communications Programming Concepts*.

---

**dupmsg Utility**

**Purpose**
Duplicates a message.
Syntax
```c
mblk_t *
dupmsg(bp)
register mblk_t * bp;
```

Description
The `dupmsg` utility calls the `dupb` utility to duplicate the message pointed to by the `bp` parameter by copying all individual message block descriptors and then linking the new message blocks to form the new message. The `dupmsg` utility does not copy data buffers, only message-block descriptors.

This utility is part of STREAMS Kernel Extensions.

Parameters
- `bp` Specifies the message to be copied.

Return Values
On successful completion, the `dupmsg` utility returns a pointer to the new message. Otherwise, it returns a null pointer.

Related Information
The `dupb` utility.

**enableok Utility**

Purpose
Enables a queue to be scheduled for service.

Syntax
```c
void
enableok(q)
queue_t * q;
```

Description
The `enableok` utility cancels the effect of an earlier `noenable` utility on the same queue. It allows a queue to be scheduled for service that had previously been excluded from queue service by a call to the `noenable` utility.

This utility is part of STREAMS Kernel Extensions.

Parameters
- `q` Specifies the queue to be enabled.

Related Information
The `noenable` utility.
esballoc Utility

Purpose
Allocates message and data blocks.

Syntax
```c
mblk_t *
esballoc(base, size, pri, free_rtn)
unsigned char * base;
int size, pri;
frn_t * free_rtn;
```

Description
The `esballoc` utility allocates message and data blocks that point directly to a client-supplied buffer. The `esballoc` utility sets the `db_base`, `b_rptr`, and `b_wptr` fields to the value specified in the `base` parameter (data buffer size) and the `db_lim` field to the `base` value plus the `size` value. The pointer to the `free_rtn` structure is placed in the `db_freep` field of the data block.

The success of the `esballoc` utility depends on the success of the `allocb` utility and also that the `base`, `size`, and `free_rtn` parameters are not null. If successful, the `esballoc` utility returns a pointer to a message block. If an error occurs, the `esballoc` utility returns a null pointer.

This utility is part of STREAMS Kernel Extensions.

Parameters
- `base` Specifies the data buffer size.
- `size` Specifies the number of bytes.
- `pri` Specifies the relative importance of this block to the module. The possible values are:
  - BPRI_LO
  - BPRI_MED
  - BPRI_HI

  The `pri` parameter is currently unused and is maintained only for compatibility with applications developed prior to UNIX System V Release 4.0.

- `free_rtn` Specifies the function and argument to be called when the message is freed.

Return Values
On successful completion, the `esballoc` utility returns a pointer to a message block. Otherwise, it returns a null pointer.

Related Information
The `allocb` utility.
flushband Utility

Purpose
Flushes the messages in a given priority band.

Syntax
void flushband(q, pri, flag)
register queue_t * q;
unsigned char pri;
ing flag;

Description
The flushband utility provides modules and drivers with the capability to flush the messages associated in a given priority band. The flag parameter is defined the same as in the flushq utility. Otherwise, messages are flushed from the band specified by the pri parameter according to the value of the flag parameter.

This utility is part of STREAMS Kernel Extensions.

Parameters
q Specifies the queue to flush.
pri Specifies the priority band to flush. If the value of the pri parameter is 0, only ordinary messages are flushed.
flag Specifies which messages to flush from the queue. Possible values are:

- FLUSHDATA
  Discards all M_DATA, M_PROTO, M_PCPROTO, and M_DELAY messages, but leaves all other messages on the queue.

- FLUSHALL
  Discards all messages from the queue.

Related Information
The flushq utility.

List of Streams Programming References and Understanding STREAMS Messages in AIX 5L Version 5.3 Communications Programming Concepts.

flushq Utility

Purpose
Flushes a queue.

Syntax
void flushq(q, flag)
register queue_t * q;
ing flag;

Description
The flushq utility removes messages from the message queue specified by the q parameter and then frees them using the freemsg utility.
If a queue behind the `q` parameter is blocked, the `flushq` utility may enable the blocked queue, as described in the `putq` utility.

This utility is part of STREAMS Kernel Extensions.

**Parameters**

- `q`: Specifies the queue to flush.
- `flag`: Specifies the types of messages to flush. Possible values are:
  - `FLUSHDATA`: Discards all `M_DATA`, `M_PROTO`, `M_PCPROTO`, and `M_DELAY` messages, but leaves all other messages on the queue.
  - `FLUSHALL`: Discards all messages from the queue.

**Related Information**

The `freemsg` utility, `putq` utility.

List of Streams Programming References and Understanding STREAMS Messages in AIX 5L Version 5.3

---

**freeb Utility**

**Purpose**

Frees a single message block.

**Syntax**

```c
void freeb(bp)
    register struct msgb *bp;
```

**Description**

The `freeb` utility frees (deallo cate) the message-block descriptor pointed to by the `bp` parameter. It also frees the corresponding data block if the reference count (see the `dupb` utility) in the data-block descriptor (datab structure) is equal to 1. If the reference count is greater than 1, the `freeb` utility does not free the data block, but decrements the reference count instead.

If the reference count is 1 and if the message was allocated by the `esballoc` utility, the function specified by the `db_frtnp->free_func` pointer is called with the parameter specified by the `db_frtnp->free_arg` pointer.

The `freeb` utility cannot be used to free a multiple-block message (see the `freemsg` utility). Results are unpredictable if the `freeb` utility is called with a null argument. Always ensure that the pointer is nonnull before using the `freeb` utility.

This utility is part of STREAMS Kernel Extensions.

**Parameters**

- `bp`: Contains a pointer to the message-block descriptor that is to be freed.
freemsg Utility

Purpose
Frees all message blocks in a message.

Syntax
```c
void freemsg(bp);
register mblk_t *bp;
```

Description
The `freemsg` utility uses the `freeb` utility to free all message blocks and their corresponding data blocks for the message pointed to by the `bp` parameter.

This utility is part of STREAMS Kernel Extensions.

Parameters

- `bp` Contains a pointer to the message that is to be freed.

Related Information
The `freeb` utility.

List of Streams Programming References and Understanding STREAMS Messages in AIX 5L Version 5.3 Communications Programming Concepts.

getadmin Utility

Purpose
Returns a pointer to a module.

Syntax
```c
int (*getadmin(mid))();
unsigned short mid;
```

Description
The `getadmin` utility returns a pointer to the module identified by the `mid` parameter.

This utility is part of STREAMS Kernel Extensions.

Parameters

- `mid` Identifies the module to locate.
Return Values
On successful completion, the getadmin utility returns a pointer to the specified module. Otherwise, it returns a null pointer.

Related Information
List of Streams Programming References, Understanding STREAMS Drivers and Modules in AIX 5L Version 5.3 Communications Programming Concepts.

getmid Utility

Purpose
Returns a module ID.

Syntax
```c
ushort
getmid(name)
char [name];
```

Description
The getmid utility returns the module ID for the module identified by the name parameter.

This utility is part of STREAMS Kernel Extensions.

Parameters
```
name Specify the module to be identified.
```

Return Values
On successful completion, the getmid utility returns the module ID. Otherwise, it returns a value of 0.

Related Information
List of Streams Programming References, Understanding STREAMS Drivers and Modules in AIX 5L Version 5.3 Communications Programming Concepts.

getmsg System Call

Purpose
Gets the next message off a stream.

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

int getmsg (fd, ctlptr, dataptr, flags)
int fd;
struct strbuf *ctlptr;
struct strbuf *dataptr;
int *flags;
```
Description

The `getmsg` system call retrieves from a STREAMS file the contents of a message located at the stream-head read queue, and places the contents into user-specified buffers. The message must contain either a data part, a control part, or both. The data and control parts of the message are placed into separate buffers, as described in the “Parameters” section. The semantics of each part are defined by the STREAMS module that generated the message.

This system call is part of the STREAMS Kernel Extensions.

Parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>fd</code></td>
<td>Specifies a file descriptor referencing an open stream.</td>
</tr>
<tr>
<td><code>ctlptr</code></td>
<td>Holds the control part of the message.</td>
</tr>
<tr>
<td><code>dataptr</code></td>
<td>Holds the data part of the message.</td>
</tr>
<tr>
<td><code>flags</code></td>
<td>Indicates the type of message to be retrieved. Acceptable values are:</td>
</tr>
<tr>
<td></td>
<td>0 Process the next message of any type.</td>
</tr>
<tr>
<td></td>
<td>RS_HIPRI Process the next message only if it is a priority message.</td>
</tr>
</tbody>
</table>

The `ctlptr` and `dataptr` parameters each point to a `strbuf` structure that contains the following members:

```
int maxlen; /* maximum buffer length */
int len;   /* length of data */
char *buf; /* ptr to buffer */
```

In the `strbuf` structure, the `maxlen` field indicates the maximum number of bytes this buffer can hold, the `len` field contains the number of bytes of data or control information received, and the `buf` field points to a buffer in which the data or control information is to be placed.

If the `ctlptr` (or `dataptr`) parameter is null or the `maxlen` field is -1, the following events occur:

- The control part of the message is not processed. Thus, it is left on the stream-head read queue.
- The `len` field is set to -1.

If the `maxlen` field is set to 0 and there is a zero-length control (or data) part, the following events occur:

- The zero-length part is removed from the read queue.
- The `len` field is set to 0.

If the `maxlen` field is set to 0 and there are more than 0 bytes of control (or data) information, the following events occur:

- The information is left on the read queue.
- The `len` field is set to 0.

If the `maxlen` field in the `ctlptr` or `dataptr` parameter is less than, respectively, the control or data part of the message, the following events occur:

- The `maxlen` bytes are retrieved.
- The remainder of the message is left on the stream-head read queue.
- A nonzero return value is provided.

By default, the `getmsg` system call processes the first priority or nonpriority message available on the stream-head read queue. However, a user may choose to retrieve only priority messages by setting the `flags` parameter to `RS_HIPRI`. In this case, the `getmsg` system call processes the next message only if it
is a priority message. When the integer pointed to by flagsp is 0, any message will be retrieved. In this case, on return, the integer pointed to by flagsp will be set to RS_HIPRI if a high-priority message was retrieved, or 0 otherwise.

If the O_NDELAY or O_NONBLOCK flag has not been set, the getmsg system call blocks until a message of the types specified by the flags parameter (priority only or either type) is available on the stream-head read queue. If the O_DELAY or O_NONBLOCK flag has been set and a message of the specified types is not present on the read queue, the getmsg system call fails and sets the errno global variable to EAGAIN.

If a hangup occurs on the stream from which messages are to be retrieved, the getmsg system call continues to operate until the stream-head read queue is empty. Thereafter, it returns 0 in the len fields of both the ctlptr and dataptr parameters.

**Return Values**

Upon successful completion, the getmsg system call returns a nonnegative value. The possible values are:

<table>
<thead>
<tr>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Indicates that a full message was read successfully.</td>
</tr>
<tr>
<td>MORECTL</td>
<td>Indicates that more control information is waiting for retrieval.</td>
</tr>
<tr>
<td>MOREDATA</td>
<td>Indicates that more data is waiting for retrieval.</td>
</tr>
<tr>
<td>MORECTL.MOREDATA</td>
<td>Indicates that both types of information remain. Subsequent getmsg calls retrieve the remainder of the message.</td>
</tr>
</tbody>
</table>

If the high priority control part of the message is consumed, the message will be placed back on the queue as a normal message of band 0. Subsequent getmsg system calls retrieve the remainder of the message. If, however, a priority message arrives or already exists on the STREAM head, the subsequent call to getmsg retrieves the higher-priority message before retrieving the remainder of the message that was put back.

On return, the len field contains one of the following:

* The number of bytes of control information or data actually received
* 0 if there is a zero-length control or data part
* -1 if no control information or data is present in the message.

If information is retrieved from a priority message, the flags parameter is set to RS_HIPRI on return.

Upon failure, getmsg returns -1 and sets errno to indicate the error.

**Error Codes**

The getmsg system call fails if one or more of the following is true:

<table>
<thead>
<tr>
<th>Error</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EAGAIN</td>
<td>The O_NDELAY flag is set, and no messages are available.</td>
</tr>
<tr>
<td>EBADF</td>
<td>The fd parameter is not a valid file descriptor open for reading.</td>
</tr>
<tr>
<td>EBADMSG</td>
<td>Queued message to be read is not valid for the getmsg system call.</td>
</tr>
<tr>
<td>EFAULT</td>
<td>The ctlptr, dataptr, or flags parameter points to a location outside the allocated address space.</td>
</tr>
<tr>
<td>EINVAL</td>
<td>A signal was caught during the getmsg system call.</td>
</tr>
<tr>
<td>ENOERR</td>
<td>An illegal value was specified in the flags parameter or else the stream referenced by the fd parameter is linked under a multiplexer.</td>
</tr>
<tr>
<td>ENOSTR</td>
<td>A stream is not associated with the fd parameter.</td>
</tr>
</tbody>
</table>
The `getmsg` system call can also fail if a STREAMS error message had been received at the stream head before the call to the `getmsg` system call. The error returned is the value contained in the STREAMS error message.

_files_

_/lib/pse.exp_ Contains the STREAMS export symbols.

Related Information

The `poll` subroutine, `read` subroutine, `write` subroutine.

The `getpmsg` system call, `putmsg` system call, `putpmsg` system call.

List of Streams Programming References and STREAMS Overview in _AIX 5L Version 5.3 Communications Programming Concepts_.

getpmsg System Call

Purpose

Gets the next priority message off a stream.

Syntax

```c
#include <stropts.h>

int getpmsg (fd, ctlptr, dataptr, bandp, flags)
int fd;
struct strbuf *ctlptr;
struct strbuf *dataptr;
int *bandp;
int *flags;
```

Description

The `getpmsg` system call is identical to the `getmsg` system call, except that the message priority can be specified.

This system call is part of the STREAMS Kernel Extensions.

Parameters

- **fd** Specifies a file descriptor referencing an open stream.
- **ctlptr** Holds the control part of the message.
- **dataptr** Holds the data part of the message.
- **bandp** Specifies the priority band of the message. If the value of the `bandp` parameter is set to 0, then the priority band is not limited.
**Flags**

Indicates the type of message priority to be retrieved. Acceptable values are:

- **MSG_ANY**
  - Process the next message of any type.

- **MSG_BAND**
  - Process the next message only if it is of the specified priority band.

- **MSG_HIPRI**
  - Process the next message only if it is a priority message.

If the value of the *flags* parameter is **MSG_ANY** or **MSG_HIPRI**, then the *bandp* parameter must be set to 0.

**Related Information**

The [poll](#) subroutine, [read](#) subroutine, [write](#) subroutine.

The [getmsg](#) system call, [putmsg](#) system call, [putpmsg](#) system call.

List of Streams Programming References and [STREAMS Overview](#) in AIX 5L Version 5.3 Communications Programming Concepts.

---

**getq Utility**

**Purpose**

Gets a message from a queue.

**Syntax**

```c
mblk_t *
getq(q)  
register queue_t * q;
```

**Description**

The *getq* utility gets the next available message from the queue pointed to by the *q* parameter. The *getq* utility returns a pointer to the message and removes that message from the queue. If no message is queued, the *getq* utility returns null.

The *getq* utility, and certain other utility routines, affect flow control in the Stream as follows: If the *getq* utility returns null, the queue is marked with the **QWANTR** flag so that the next time a message is placed on it, it will be scheduled for service (that is, enabled - see the [qenable](#) utility). If the data in the enqueued messages in the queue drops below the low-water mark, as specified by the *q_lowat* field, and if a queue behind the current queue has previously attempted to place a message in the queue and failed, (that is, was blocked - see the [canput](#) utility), then the queue behind the current queue is scheduled for service.

The queue count is maintained on a per-band basis. Priority band 0 (normal messages) uses the *q_count* and *q_lowat* fields. Nonzero priority bands use the fields in their respective *qband* structures (the *qb_count* and *qb_lowat* fields). All messages appear on the same list, linked according to their *b_next* pointers.

The *q_count* field does not reflect the size of all messages on the queue; it only reflects those messages in the normal band of flow.

This utility is part of STREAMS Kernel Extensions.
Parameters

q    Specifies the queue from which to get the message.

Return Values

On successful completion, the **getq** utility returns a pointer to the message. Otherwise, it returns a null value.

Related Information

The **canput** utility, **qenable** utility, **rmvq** utility.

List of Streams Programming References and Understanding STREAMS Messages in AIX 5L Version 5.3 Communications Programming Concepts.

**insq Utility**

**Purpose**

Puts a message at a specific place in a queue.

**Syntax**

```c
int
insq(q, emp, mp)
register queue_t * q;
register mblk_t * emp;
register mblk_t * mp;
```

**Description**

The **insq** utility places the message pointed to by the *mp* parameter in the message queue pointed to by the *q* parameter, immediately before the already-queued message pointed to by the *emp* parameter.

If an attempt is made to insert a message out of order in a queue by using the **insq** utility, the message will not be inserted and the routine is not successful.

This utility is part of STREAMS Kernel Extensions.

The queue class of the new message is ignored. However, the priority band of the new message must adhere to the following format:

```c
emp->b_prev->b_band >= mp->b_band && mp->b_band >= emp->b_band.
```

**Parameters**

q    Specifies the queue on which to place the message.

emp   Specifies the existing message before which the new message is to be placed.

If the *emp* parameter has a value of null, the message is placed at the end of the queue. If the *emp* parameter is nonnull, it must point to a message that exists on the queue specified by the *q* parameter, or undesirable results could occur.

mp    Specifies the message that is to be inserted on the queue.

**Return Values**

On successful completion, the **insq** utility returns a value of 1. Otherwise, it returns a value of 0.
Related Information
The `getq` utility.

List of Streams Programming References and Understanding STREAMS Messages in AIX 5L Version 5.3 Communications Programming Concepts.

ioctl Streams Device Driver Operations
(As defined in X/Open Common Application Environment (CAE) Specification: System Interfaces and Headers, Issue 5 (2/97.).)

Purpose
Controls a STREAMS device.

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

int ioctl (fd, request, .../*arg*/);
```

Parameters
- `fd` An open file descriptor that refers to a device.
- `request` Selects the control function to be performed and will depend on the STREAMS device being addressed.
- `.../*arg*/` Represents additional information that is needed by this specific STREAMS device to perform the requested function. The type of `arg` depends on the particular control request, but it is either an integer or a pointer to a device-specific data structure.

The following `ioctl` commands, with error values indicated, are applicable to all STREAMS files:

- **I_PUSH**
  Pushes the module whose name is pointed to by `arg` onto the top of the current STREAM, just below the STREAM head. It then calls the `open` function of the newly-pushed module.

  The `ioctl` function with the `I_PUSH` command will fail if:
  - `[EINVAL]` Invalid module name.
  - `[ENXIO]` Hangup received on `fd`. 

Chapter 3. Streams 265
I_POP

Removes the module just below the STREAM head of the STREAM pointed to by \textit{fd}. The \textit{arg} argument should be 0 in an I_POP request.

The \textit{ioctl} function with the I_POP command will fail if:

- [EINVAL] No module present in the STREAM.
- [ENXIO] Hangup received on \textit{fd}.

I_LOOK

Retrieves the name of the module just below the STREAM head of the STREAM pointed to by \textit{fd}, and places it in a character string pointed to by \textit{arg}. The buffer pointed to by \textit{arg} should be at least FMNAMESZ+1 bytes long, where FMNAMESZ is defined in \texttt{<stropts.h>}.

The \textit{ioctl} function with the I_LOOK command will fail if:

- [EINVAL] No module present in the STREAM.

I_FLUSH

This request flushes read and/or write queues, depending on the value of \textit{arg}. Valid \textit{arg} values are:

- FLUSHR: Flush all read queues.
- FLUSHW: Flush all write queues.
- FLUSHRW: Flush all read and all write queues.

The \textit{ioctl} function with the I_FLUSH command will fail if:

- [EINVAL] Invalid \textit{arg} argument.
- [EAGAIN] or [ENOSR] Unable to allocate buffers for flush messages.
- [ENXIO] Hangup received on \textit{fd}.

I_FLUSHBAND

Flushes a particular band of messages. The \textit{arg} argument points to a bandinfo structure. The \texttt{bi_flag} member may be one of FLUSHR, FLUSHW, OR FLUSHRW as described above. The \texttt{bi_pri} member determines the priority band to be flushed.
Request that the STREAMS implementation send the SIGPOLL signal to the calling process when a particular event has occurred on the STREAM associated with \textit{fd}. \texttt{I_SETSIG} supports an asynchronous processing capability in STREAMS. The value of \textit{arg} is a bitmask that specifies the events for which the process should be signaled. It is the bitwise-OR of an combination of the following constants:

\begin{description}
\item[\texttt{S_RDNORM}] A normal (priority band set to 0) message has arrived at the head of a STREAM head read queue. A signal will be generated even if the message is of zero length.
\item[\texttt{S_RDBAND}] A message with a nonzero priority band has arrived at the head of a STREAM head read queue. A signal will be generated even if the message is of zero length.
\item[\texttt{S_INPUT}] A message, other than a high-priority message, has arrived at the head of a STREAM head read queue. A signal will be generated even if the message is of zero length.
\item[\texttt{S_HIPRI}] A high-priority message is present on a STREAM head read queue. A signal will be generated even if the message is of zero length.
\item[\texttt{S_OUTPUT}] The write queue for normal data (priority band 0) just below the STREAM head is no longer full. This notifies the process that there is room on the queue for sending (or writing) normal data downstream.
\item[\texttt{S_WRNORM}] Same as \texttt{S_OUTPUT}.
\item[\texttt{S_WRBAND}] The write queue for a nonzero priority band just below the STREAM head is no longer full. This notifies the process that there is room on the queue for sending (or writing) priority data downstream.
\item[\texttt{S_MSG}] A STREAMS signal message that contains the SIGPOLL signal has reached the front of the STREAM head read queue.
\item[\texttt{S_ERROR}] Notification of an error condition has reached the STREAM head.
\item[\texttt{S_HANGUP}] Notification of a hangup has reached the STREAM head.
\item[\texttt{S_BANDURG}] When used in conjunction with \texttt{S_RDBAND}, SIGURG is generated instead of SIGPOLL when a priority message reaches the front of the STREAM head read queue.
\end{description}

If \textit{arg} is 0, the calling process will be unregistered and will not receive further SIGPOLL signals for the stream associated with \textit{fd}.

Processes that wish to receive SIGPOLL signals must explicitly register to receive them using \texttt{I_SETSIG}. If several processes register to receive this signal for the same event on the same STREAM, each process will be signaled when the event occurs.

The \texttt{ioctl} function with the \texttt{I_SETSIG} command will fail if:

\begin{description}
\item[\texttt{EINVAL}] The value of \textit{arg} is invalid.
\item[\texttt{EINVAL}] The value of \textit{arg} is 0 and the calling process is not registered to receive the SIGPOLL signal.
\item[\texttt{EAGAIN}] There were insufficient resources to store the signal request.
\end{description}
I_GETSIG

Returns the events for which the calling process is currently registered to be sent a SIGPOLL signal. The events are returned as a bitmask in an int pointed to by arg, where the events are those specified in the description of I_SETSIG above.

The ioctl function with the I_GETSIG command will fail if:

[EINVAL]
Process is not registered to receive the SIGPOLL signal.

I_FIND

The request compares the names of all modules currently present in the STREAM to the name pointed to by arg, and returns 1 if the named module is present in the STREAM, or returns 0 if the named module is not present.

The ioctl function with the I_FIND command will fail if:

[EINVAL]
arg does not contain a valid module name.

I_PEEK

This request allows a process to retrieve the information in the first message on the STREAM head read queue without taking the message off the queue. It is analogous to getmsg except that this command does not remove the message from the queue. The arg argument points to a strpeek structure.

The maxlen member in the ctllbuf and databuf strbuf structures must be set to the number of bytes of control information and/or data information, respectively, to retrieve. The flags member may be marked RS_HIPRI or 0, as described by getmsg. If the process sets flags to RS_HIPRI, for example, I_PEEK will only look for a high-priority message on the STREAM head read queue.

I_PEEK returns 1 if a message was retrieved, and returns 0 if no message was found on the STREAM head read queue, or if the RS_HIPRI flag was set in flags and a high-priority message was not present on the STREAM head read queue. It does not wait for a message to arrive. On return, ctllbuf specifies information in the control buffer, databuf specifies information in the data buffer, and flags contains the value RS_HIPRI or 0.

I_SRDOPT

Sets the read mode using the value of the argument arg. Read modules are described in read. Valid arg flags are:

RNORM
Byte-stream mode, the default.

RMSGD
Message-discard mode.

RMSGN
Message-nondiscard mode.

The bitwise inclusive OR of RMSGD and RMSGN will return [EINVAL]. The bitwise inclusive OR of RNORM and either RMSGD or RMSGN will result in the other flag overriding RNORM which is the default.

In addition, treatment of control messages by the STREAM head may be changed by setting any of the following flags in arg:

RPROTNORM
Fail read with [EBADMSG] if a message containing a control part is at the front of the STREAM head read queue.

RPROTDAT
Deliver the control part of a message as data when a process issues a read.

RPROTDIS
Discard the control part of a message, delivering any data portion, when a process issues a read.

I_GRDOPT

Returns the current read mode setting as, described above, in an int pointed to by the argument arg. Read modes are described in read.
Counts the number of data bytes in the data part of the first message on the STREAM head read queue and places this value in the int pointed to by arg. The return value for the command is the number of messages on the STREAM head read queue. For example, if 0 is returned in arg, but the ioctl return value is greater than 0, this indicates that a zero-length message is next on the queue.

Creates a message from a specified buffer(s), adds information about another STREAM, and sends the message downstream. The message contains a control part and an optional data part. The data and control parts to be sent are distinguished by placement in separate buffers, as described below. The arg argument points to a strfdinsert structure.

The len member in the ctlbuf strbuf structure must be set to the size of a t_uscalar_t plus the number of bytes of control information to be sent with the message. The fd member specifies the file descriptor of the other STREAM, and the offset member, which must be suitably aligned for use as a t_uscalar_t, specifies the offset from the start of the control buffer where I_FDINSERT will store a t_uscalar_t whose interpretation is specific to the STREAM end. The len member in the databuf strbuf structure must be set to the number of bytes of data information to be sent with the message, or to 0 if no data part is to be sent.

The flags member specifies the type of message to be created. A normal message is created if flags is set to 0, and a high-priority message is created if flags is set to RS_HIPRI. For non-priority messages, I_FDINSERT will block if the STREAM write queue is full due to internal flow control conditions. For priority messages, I_FDINSERT does not block on this condition. For non-priority messages, I_FDINSERT does not block when the write queue is full and O_NONBLOCK is set. Instead, it fails and sets errno to [EAGAIN].

I_FDINSERT also blocks, unless prevented by lack of internal resources, waiting for the availability of message blocks in the STREAM, regardless of priority or whether O_NONBLOCK has been specified. No partial message is sent.

The ioctl function with the I_FDINSERT command will fail if:

[EAGAIN]  
A non-priority message is specified, the O_NONBLOCK flag is set, and the STREAM write queue is full due to internal flow control conditions.

[EAGAIN] or [ENOSR]  
Buffers cannot be allocated for the message that is to be created.

[EINVAL]  
One of the following:
  • The fd member of the strfdinsert structure is not a valid, open STREAM file descriptor.
  • The size of a t_uscalar_t plus offset is greater than the len member for the buffer specified through ctlptr.
  • The offset member does not specify a properly-aligned location in the data buffer.
  • An undefined value is stored in flags.

[ENXIO]  
Hangup received on the STREAM identified by either the fd argument or the fd member of the strfdinsert structure.

[ERANGE]  
The len member for the buffer specified through databuf does not fall within the range specified by the maximum and minimum packet sizes of the topmost STREAM module or the len member for the buffer specified through databuf is larger than the maximum configured size of the data part of a message; or the len member for the buffer specified through ctlbuf is larger than the maximum configured size of the control message.
I_STR

Constructs an internal STREAMS ioctl message from the data pointed to by arg, and sends that message downstream.

This mechanism is provided to send ioctl requests to downstream modules and drivers. It allows information to be sent with ioctl, and returns to the process any information sent upstream by the downstream recipient. I_STR blocks until the system responds with either a positive or negative acknowledgment message, or until the request “times out” after some period of time. If the request times out, it fails with errno set to [ETIME].

At most, one I_STR can be active on a STREAM. Further I_STR calls will block until the active I_STR completes at the STREAM head. The default timeout interval for these requests is 15 seconds. The O_NONBLOCK flag has no effect on this call.

To send requests downstream, arg must point to a strioctl structure.

The ic_cmd member is the internal ioctl command intended for a downstream module or driver and ic_timeout is the number of seconds:

-1 = Infinite.
0 = Use implementation-dependent timeout interval.
>0 = As specified.

an I_STR request will wait for acknowledgment before timing out. ic_len is the number of bytes in the data argument, and ic_dp is a pointer to the data argument. The ic_len member has two uses:

- On input, it contains the length of the data argument passed on.
- On return from the command, it contains the number of bytes being returned to the process (the buffer pointed to by ic_dp should be large enough to contain the maximum amount of data that any module or the driver in the STREAM can return).

The ioctl function with the I_STR command will fail if:

[EAGAIN] or [ENOSR]
Unable to allocate buffers for the ioctl message.

[EINVAL]
The ic_len member is less than 0 or larger than the maximum configured size of the data part of a message, or ictimeout is less than -1.

[ENXIO]
Hangup received on fd.

[ETIME]
A downstream ioctl timed out before acknowledgment was received.

An I_STR can also fail while waiting for an acknowledgment if a message indicating an error or a hangup is received at the STREAM head. In addition, an error code can be returned in the positive or negative acknowledgment message, in the event the ioctl command sent downstream fails. For these cases, I_STR fails with errno set to the value in the message.

I_SWROPT

Sets the write mode using the value of the argument arg. Valid bit settings for arg are:

SNDZERO
Send a zero-length message downstream when a write of 0 bytes occurs. To not send a zero-length message when a write of 0 bytes occurs, this bit must not be set in arg (for example, arg would be set to 0).

The ioctl function with the I_SWROPT command will fail if:

[EINVAL]
arg is not the above value.

I_GWROPT

Returns the current write mode setting, as described above, in the int that is pointed to by the argument arg.
I SENDFD

I_SENDFD creates a new reference to the open file description, associated with the file descriptor arg, and writes a message on the STREAMS-based pipe fd containing this reference, together with the user ID and group ID of the calling process.

The ioctl function with the I_SENDFD command will fail if:

[EAGAIN]
   The sending STREAM is unable to allocate a message block to contain the file pointer; or the read queue of the receiving STREAM head is full and cannot accept the message sent by I_SENDFD.

[EBADF]
   The arg argument is not a valid, open file descriptor.

[EINVAL]
   The fd argument is not connected to a STREAM pipe.

[ENXIO]
   Hangup received on fd.

I_RECVFD

Retrieves the reference to an open file description from a message written to a STREAMS-based pipe using the I_SENDFD command, and allocates a new file descriptor in the calling process that refers to this open file description. The arg argument is a pointer to an strrecvfd data structure as defined in stropts.h.

The fd member is a file descriptor. The uid and gid members are the effective user ID and group ID, respectively, of the sending process.

If O_NONBLOCK is not set, I_RECVFD blocks until a message is present at the STREAM head. If O_NONBLOCK is set, I_RECVFD fails with errno set to [EAGAIN] if no message is present at the STREAM head.

If the message at the STREAM head is a message sent by an I_SENDFD, a new file descriptor is allocated for the open file descriptor referenced in the message. The new file descriptor is placed in the fd member of the strrecfd structure pointed to by arg.

The ioctl function with the I_RECVFD command will fail if:

[EAGAIN]
   A message is not present at the STREAM head read queue and the O_NONBLOCK flag is set.

[EBADMSG]
   The message at the STREAM head read queue is not a message containing a passed file descriptor.

[EMFILE]
   The process has the maximum number of file descriptors currently open that is allowed.

[ENXIO]
   Hangup received on fd.
This request allows the process to list all the module names on the STREAM, up to an including the topmost driver name. If arg is a null pointer, the return value is the number of modules, including the driver, that are on the STREAM pointed to by fd. This lets the process allocate enough space for the module names. Otherwise, it should point to an str_list structure.

The sl_nmods member indicates the number of entries that process has allocated in the array. Upon return, the sl_modlist member of the str_list structure contains the list of module names, and the number of entries that have been filled into the sl_modlist array is found in the sl_nmods member (the number includes the number of modules including the driver). The return value from ioctl is 0. The entries are filled in starting at the top of the STREAM and continuing downstream until either the end of the STREAM is reached, or the number of requested modules (sl_nmods) is satisfied.

The ioctl function with the I_LIST command will fail if:

[EINVAL] The sl_nmods member is less than 1.

[EAGAIN] or [ENOSR] Unable to allocate buffers.

This request allows the process to see if the message at the head of the STREAM head read queue is marked by some module downstream. The arg argument determines how the checking is done when there may be multiple marked messages on the STREAM head read queue. It may take on the following values:

ANYMARK
Check if the message is marked.

LASTMARK
Check if the message is the last one marked on the queue.

The bitwise inclusive OR of the flags ANYMARK and LASTMARK is permitted.

The return value is 1 if the mark condition is satisfied and 0 otherwise.

The ioctl function with the I_ATMARK command will fail if:

[EINVAL] Invalid arg value.

Check if the message of given priority band exists on the STREAM head read queue. This returns 1 if a message of the given priority exists, 0 if no such message exists, or -1 on error. arg should be of type int.

The ioctl function with the I_CKBAND command will fail if:

[EINVAL] Invalid arg value.

Return the priority band of the first message on the STREAM head read queue in the integer referenced by arg.

The ioctl function with the I_GETBAND command will fail if:

[ENODATA] No message on the STREAM head read queue.

Check if a certain band is writable. arg is set to the priority band in question. The return value is 0 if the band is flow-controlled, 1 if the band is writable, or -1 on error.

The ioctl function with the I_CANPUT command will fail if:

[EINVAL] Invalid arg value.
This request allows the process to set the time the STREAM head will delay when a STREAM is closing and there is no data on the write queues. Before closing each module or driver, if there is data on its write queue, the STREAM head will delay for the specified amount of time to allow the data to drain. If, after the delay, data is still present, they will be flushed. The arg argument is a pointer to an integer specifying the number of milliseconds to delay, rounded up to the nearest valid value. If I_SETCLTIME is not performed on a STREAM, an implementation-dependent default timeout interval is used.

The ioctl function with the I_SETCLTIME command will fail if:

- [EINVAL] Invalid arg value.

I_GETCLTIME

This request returns the close time delay in the integer pointed to by arg.

Multiplexed STREAMS Configurations

The following four commands are used for connecting and disconnecting multiplexed STREAMS configurations. These commands use an implementation-dependent default timeout interval.

I_LINK

Connects two STREAMS, where fd is the file descriptor of the STREAM connected to the multiplexing driver, and arg is the file descriptor of the STREAM connected to another driver. The STREAM designated by arg gets connected below the multiplexing driver. I_LINK requires the multiplexing driver to send an acknowledgment message to the STREAM head regarding the connection. This call returns a multiplexer ID number (an identifier used to disconnect the multiplexer; see I_UNLINK) on success, and -1 on failure.

The ioctl function with the I_LINK command will fail if:

- [ENXIO] Hangup received on fd.
- [ETIME] Time out before acknowledgment message was received at STREAM head.
- [EAGAIN] or [ENOSR] Unable to allocate STREAMS storage to perform the I_LINK.
- [EINVAL] The fd argument does not support multiplexing; or arg is not a STREAM or is already connected downstream from a multiplexer; or the specified I_LINK operation would connect the STREAM head in more than one place in the multiplexed STREAM.

An I_LINK can also fail while waiting for the multiplexing driver to acknowledge the request, if a message indicating an error or a hangup is received at the STREAM head of fd. In addition, an error code can be returned in the positive or negative acknowledgment message. For these cases, I_LINK fails with errno set to the value in the message.
I_UNLINK

Disconnects the two STREAMS specified by `fd` and `arg`. `fd` is the file descriptor of the STREAM connected to the multiplexing driver. The `arg` argument is the multiplexer ID number that was returned by the I_LINK ioctl command when a STREAM was connected downstream from the multiplexing driver. If `arg` is MUXID_ALL, then all STREAMS that were connected to `fd` are disconnected. As in I_LINK, this command requires acknowledgment.

The ioctl function with the I_UNLINK command will fail if:

- [ENOMEM] Hangup received on `fd`.
- [ETIME] Time out before acknowledgment message was received at STREAM head.
- [EAGAIN] or [ENOSR] Unable to allocate buffers for the acknowledgment message.
- [EINVAL] Invalid multiplexer ID number.

An I_UNLINK can also fail while waiting for the multiplexing driver to acknowledge the request is a message indicating an error or a hangup is received at the STREAM head of `fd`. In addition, an error code can be returned in the positive or negative acknowledgment message. For these cases, I_UNLINK fails with `errno` set to the value in the message.

I_PLINK

Creates a persistent connection between two STREAMS, where `fd` is the file descriptor of the STREAM connected to the multiplexing driver, and `arg` is the file descriptor of the STREAM connected to another driver. This call creates a persistent connection which can exist even if the file descriptor `fd` associated with the upper STREAM to the multiplexing driver is closed. The STREAM designated by `arg` gets connected via a persistent connection below the multiplexing driver. I_PLINK requires the multiplexing driver to send an acknowledgment to the STREAM head. This call returns a multiplexer ID number (an identifier that may be used to disconnect the multiplexer; see I_PUNLINK) on success, and -1 on failure.

The ioctl function with the I_PLINK command will fail if:

- [ENOMEM] Hangup received on `fd`.
- [ETIME] Time out before acknowledgment message was received at STREAM head.
- [EAGAIN] or [ENOSR] Unable to allocate STREAMS storage to perform the I_PLINK.
- [EINVAL] The `fd` argument does not support multiplexing; or `arg` is not a STREAM or is already connected downstream from a multiplexer; or the specified I_PLINK operation would connect the STREAM head in more than one place in the multiplexed STREAM.

An I_PLINK can also fail while waiting for the multiplexing driver to acknowledge the request, if a message indicating an error or a hangup is received at the STREAM head of `fd`. In addition, an error code can be returned in the positive or negative acknowledgment message. For these cases, I_PLINK fails with `errno` set to the value in the message.
Disconnects the two STREAMS specified by \textit{fd} and \textit{arg} from a persistent connection. The \textit{fd} argument is the file descriptor of the STREAM connected to the multiplexing driver. The \textit{arg} argument is the multiplexer ID number that was returned by the I\_PLINK \textit{ioctl} command when a STREAM was connected downstream from the multiplexing driver. If \textit{arg} is MUXID\_ALL than all STREAMS which are persistent conditions to \textit{fd} are disconnected. As in I\_PLINK, this command requires the multiplexing driver to acknowledge the request.

The \textit{ioctl} function with the I\_PUNLINK command will fail if:

- **[ENXIO]**
  - Hangup received on \textit{fd}.
- **[ETIME]**
  - Time out before acknowledgment message was received at STREAM head.
- **[EAGAIN]** or **[ENOSR]**
  - Unable to allocate buffers for the acknowledgment message.
- **[EINVAL]**
  - Invalid multiplexer ID number.

An I\_PUNLINK can also fail while waiting for the multiplexing driver to acknowledge the request if a message indicating an error or a hangup is received at the STREAM head of \textit{fd}. In addition, an error code can be returned in the positive or negative acknowledgment message. For these cases, I\_PUNLINK fails with \textit{errno} set to the value in the message.

**Return Value**

Upon successful completion, \textit{ioctl} returns a value other than -1 that depends upon the STREAMS device control function. Otherwise, it returns -1 and sets \textit{errno} to indicate the error.

**Errors**

Under the following general conditions, \textit{ioctl} will fail if:

- **[EBADF]**
  - The \textit{fd} argument is not a valid open file descriptor.
- **[EINTR]**
  - A signal was caught during the \textit{ioctl} operation.
- **[EINVAL]**
  - The STREAM or a multiplexer referenced by \textit{fd} is linked (directly or indirectly) downstream from a multiplexer.

If an underlying device driver detects an error, then \textit{ioctl} will fail if:

- **[EINVAL]**
  - The \textit{request} or \textit{arg} argument is not valid for this device.
- **[EIO]**
  - Some physical I/O error has occurred.
- **[ENOTTY]**
  - The \textit{fd} argument is not associated with a STREAMS device that accepts control functions. A file descriptor was obtained from a call to the \textit{shm\_open} subroutine.
- **[ENXIO]**
  - The \textit{request} and \textit{arg} arguments are valid for this device driver, but the service requested cannot be performed on this particular sub-device.
- **[ENODEV]**
  - The \textit{fd} argument refers to a valid STREAMS device, but the corresponding device driver does not support the \textit{ioctl} function.

If a STREAM is connected downstream from a multiplexer, and \textit{ioctl} command except I\_UNLINK and I\_PUNLINK will set \textit{errno} to [EINVAL].

**Application Usage**

The implementation-dependent timeout interval for STREAMS has historically been 15 seconds.
I_ATMARK streamio Operation

Purpose
Checks to see if a message is marked.

Description
The I_ATMARK operation shows the user if the current message on the stream-head read queue is marked by a downstream module. The arg parameter determines how the checking is done when there are multiple marked messages on the stream-head read queue. The possible values for the arg parameter are:

<table>
<thead>
<tr>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ANYMARK</td>
<td>Read to determine if the message is marked by a downstream module.</td>
</tr>
<tr>
<td>LASTMARK</td>
<td>Read to determine if the message is the last one marked on the queue by a downstream module.</td>
</tr>
</tbody>
</table>

The I_ATMARK operation returns a value of 1 if the mark condition is satisfied. Otherwise, it returns a value of 0.

This operation is part of STREAMS Kernel Extensions.

Error Codes
If unsuccessful, the errno global variable is set to the following value:

<table>
<thead>
<tr>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EINVAL</td>
<td>The value of the arg parameter could not be used.</td>
</tr>
</tbody>
</table>

I_CANPUT streamio Operation

Purpose
Checks if a given band is writable.

Description
The I_CANPUT operation checks a given priority band to see if it can be written on. The arg parameter contains the priority band to be checked.

Return Values
The return value is set to one of the following:

<table>
<thead>
<tr>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>The band is flow controlled.</td>
</tr>
<tr>
<td>1</td>
<td>The band is writable.</td>
</tr>
<tr>
<td>-1</td>
<td>An error occurred.</td>
</tr>
</tbody>
</table>
Error Codes
If unsuccessful, the errno global variable is set to the following value:

<table>
<thead>
<tr>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EINVAL</td>
<td>The value in the arg parameter is invalid.</td>
</tr>
</tbody>
</table>

I_CKBAND streamio Operation

Purpose
Checks if a message of a particular band is on the stream-head read queue.

Description
The I_CKBAND operation checks to see if a message of a given priority band exists on the stream-head read queue. The arg parameter is an integer containing the value of the priority band being searched for.

The I_CKBAND operation returns a value of 1 if a message of the given band exists. Otherwise, it returns a value of -1.

This operation is part of STREAMS Kernel Extensions.

Error Codes
If unsuccessful, the errno global variable is set to the following value:

<table>
<thead>
<tr>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EINVAL</td>
<td>The value in the arg parameter is not valid.</td>
</tr>
</tbody>
</table>

I_FDINSERT streamio Operation

Purpose
Creates a message from user-specified buffers, adds information about another stream and sends the message downstream.

Description
The I_FDINSERT operation creates a message from user-specified buffers, adds information about another stream, and sends the message downstream. The message contains a control part and an optional data part. The data and control parts transmitted are identified by their placement in separate buffers. The arg parameter points to a strfdinsert structure that contains the following elements:

```c
struct strbuf ctlbuf;
struct strbuf databuf;
long flags;
int fildes;
int offset;
```

The len field in the strbuf structure must be set to the size of a pointer plus the number of bytes of control information sent with the message. The fildes field in the strfdinsert structure specifies the file descriptor of the other stream. The offset field, which must be word-aligned, specifies the number of bytes beyond the beginning of the control buffer to store a pointer. This pointer will be the address of the read queue structure of the driver for the stream corresponding to the fildes field in the strfdinsert structure. The len field in the strbuf structure of the databuf field must be set to the number of bytes of data information sent with the message or to 0 if no data part is sent.
The flags field specifies the type of message created. There are two valid values for the flags field:

<table>
<thead>
<tr>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Creates a nonpriority message.</td>
</tr>
<tr>
<td>RS_HIPRI</td>
<td>Creates a priority message.</td>
</tr>
</tbody>
</table>

For nonpriority messages, the I_FDINSERT operation blocks if the stream write queue is full due to internal flow-control conditions. For priority messages, the I_FDINSERT operation does not block on this condition. For nonpriority messages, the I_FDINSERT operation does not block when the write queue is full and the O_NDELAY flag is set. Instead, the operation fails and sets the errno global variable to EAGAIN.

The I_FDINSERT operation also blocks unless prevented by lack of internal resources, while it is waiting for the availability of message blocks in the stream, regardless of priority or whether the O_NDELAY flag has been specified. No partial message is sent.

This operation is part of STREAMS Kernel Extensions.

Error Codes

If unsuccessful, the errno global variable is set to one of the following values:

<table>
<thead>
<tr>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EAGAIN</td>
<td>A nonpriority message was specified, the O_NDELAY flag is set, and the stream write queue is full due to internal flow-control conditions.</td>
</tr>
<tr>
<td>ENOSR</td>
<td>Buffers could not be allocated for the message that was to be created due to insufficient STREAMS memory resources.</td>
</tr>
<tr>
<td>EFAULT</td>
<td>The arg parameter points to an area outside the allocated address space, or the buffer area specified in the ctlbuf or databuf field is outside this space.</td>
</tr>
<tr>
<td>EINVAL</td>
<td>One of the following conditions has occurred:</td>
</tr>
<tr>
<td></td>
<td>The fildes field in the strfdinsert structure is not a valid, open stream file descriptor.</td>
</tr>
<tr>
<td></td>
<td>The size of a pointer plus the value of the offset field is greater than the len field for the buffer specified through the ctlptr field.</td>
</tr>
<tr>
<td></td>
<td>The offset parameter does not specify a properly aligned location in the data buffer.</td>
</tr>
<tr>
<td></td>
<td>An undefined value is stored in the flags parameter.</td>
</tr>
<tr>
<td>ENXIO</td>
<td>Hangup received on the fildes parameter of the ioctl call or the fildes field in the strfdinsert structure.</td>
</tr>
<tr>
<td>ERANGE</td>
<td>The len field for the buffer specified through the databuf field does not fall within the range specified by the maximum and minimum packet sizes of the topmost stream module; or the len field for the buffer specified through the databuf field is larger than the maximum configured size of the data part of a message; or the len field for the buffer specified through the ctlbuf field is larger than the maximum configured size of the control part of a message.</td>
</tr>
</tbody>
</table>

The I_FDINSERT operation is also unsuccessful if an error message is received by the stream head corresponding to the fildes field in the strfdinsert structure. In this case, the errno global variable is set to the value in the message.

I_FIND streamio Operation

Purpose

Compares the names of all modules currently present in the stream to a specified name.
Description
The **I_FIND** operation compares the names of all currently present in the stream to the name pointed to by the `arg` parameter, and returns a value of 1 if the named module is present in the stream. It returns a value of 0 if the named module is not present.

This operation is part of STREAMS Kernel Extensions.

**Error Codes**
If unsuccessful, the `errno` global variable is set to one of the following values:

<table>
<thead>
<tr>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EFAULT</td>
<td>The <code>arg</code> parameter points outside the allocated address space.</td>
</tr>
<tr>
<td>EINVAL</td>
<td>The <code>arg</code> parameter does not contain a valid module name.</td>
</tr>
</tbody>
</table>

---

**I_FLUSH streamio Operation**

**Purpose**
Flushes all input or output queues.

**Description**
The **I_FLUSH** operation flushes all input or output queues depending on the value of the `arg` parameter.

Legal values for the `arg` parameter are:

<table>
<thead>
<tr>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>FLUSHR</td>
<td>Flush read queues.</td>
</tr>
<tr>
<td>FLUSHW</td>
<td>Flush write queues.</td>
</tr>
<tr>
<td>FLUSHRW</td>
<td>Flush read and write queues.</td>
</tr>
</tbody>
</table>

This operation is part of STREAMS Kernel Extensions.

**Error Codes**
If unsuccessful, the `errno` global variable is set to one of the following values:

<table>
<thead>
<tr>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ENOSR</td>
<td>Unable to allocate buffers for flush message due to insufficient STREAMS memory resources.</td>
</tr>
<tr>
<td>EINVAL</td>
<td>Invalid value for the <code>arg</code> parameter.</td>
</tr>
<tr>
<td>ENXIO</td>
<td>Hangup received on the <code>fildes</code> parameter.</td>
</tr>
</tbody>
</table>

---

**I_FLUSHBAND streamio Operation**

**Purpose**
Flushes all messages from a particular band.

**Description**
The **I_FLUSHBAND** operation flushes all messages of a given priority band from all input or output queues. The `arg` parameter points to a `bandinfo` structure that contains the following elements:

```c
unsigned char bi_pri;
int bi_flag;
```
The elements are defined as follows:

<table>
<thead>
<tr>
<th>Element</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>bi_pri</td>
<td>Specifies the band to be flushed.</td>
</tr>
<tr>
<td>bi_flag</td>
<td>Specifies the queues to be pushed. Legal values for the bi_flag field are:</td>
</tr>
<tr>
<td>FLUSHR</td>
<td>Flush read queues.</td>
</tr>
<tr>
<td>FLUSHW</td>
<td>Flush write queues.</td>
</tr>
<tr>
<td>FLUSHRW</td>
<td>Flush read and write queues.</td>
</tr>
</tbody>
</table>

This operation is part of STREAMS Kernel Extensions.

**Error Codes**
If unsuccessful, the errno global variable is set to one of the following values:

<table>
<thead>
<tr>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ENOSR</td>
<td>Unable to allocate buffers for flush message due to insufficient STREAMS memory resources.</td>
</tr>
<tr>
<td>EINVAL</td>
<td>Invalid value for the arg parameter.</td>
</tr>
<tr>
<td>ENXIO</td>
<td>Hangup received on the fildes parameter.</td>
</tr>
</tbody>
</table>

---

### I_GETBAND streamio Operation

**Purpose**

Gets the band of the first message on the stream-head read queue.

**Description**

The I_GETBAND operation returns the priority band of the first message on the stream-head read queue in the integer referenced by the arg parameter.

This operation is part of STREAMS Kernel Extensions.

**Error Codes**
If unsuccessful, the errno global variable is set to the following value:

<table>
<thead>
<tr>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ENODATA</td>
<td>No message is on the stream-head read queue.</td>
</tr>
</tbody>
</table>

### I_GETCLTIME streamio Operation

**Purpose**

Returns the delay time.

**Description**

The I_GETCLTIME operation returns the delay time, in milliseconds, that is pointed to by the arg parameter.
I_GETSIG streamio Operation

Purpose
Returns the events for which the calling process is currently registered to be sent a SIGPOLL signal.

Description
The I_GETSIG operation returns the events for which the calling process is currently registered to be sent a SIGPOLL signal. The events are returned as a bitmask pointed to by the arg parameter, where the events are those specified in the description of the I_SETSIG operation.

Error Codes
If unsuccessful, the errno global variable is set to one of the following values:

<table>
<thead>
<tr>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EINVAL</td>
<td>Process not registered to receive the SIGPOLL signal.</td>
</tr>
<tr>
<td>EFAULT</td>
<td>The arg parameter points outside the allocated address space.</td>
</tr>
</tbody>
</table>

I_GRDOPT streamio Operation

Purpose
Returns the current read mode setting.

Description
The I_GRDOPT operation returns the current read mode setting in an int parameter pointed to by the arg parameter. Read modes are described in the read subroutine description.

Error Codes
If unsuccessful, the errno global variable is set to the following value:

<table>
<thead>
<tr>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EFAULT</td>
<td>The arg parameter points outside the allocated address space.</td>
</tr>
</tbody>
</table>

I_LINK streamio Operation

Purpose
Connects two specified streams.

Description
The I_LINK operation is used for connecting multiplexed STREAMS configurations.

The I_LINK operation connects two streams, where the fildes parameter is the file descriptor of the stream connected to the multiplexing driver, and the arg parameter is the file descriptor of the stream connected to another driver. The stream designated by the arg parameter gets connected below the multiplexing
The I_LINK operation requires the multiplexing driver to send an acknowledgment message to the stream head regarding the linking operation. This call returns a multiplexer ID number (an identifier used to disconnect the multiplexer; see the I_UNLINK operation) on success, and a value of -1 on failure.

This operation is part of STREAMS Kernel Extensions.

**Error Codes**

If unsuccessful, the errno global variable is set to one of the following values:

<table>
<thead>
<tr>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ENXIO</td>
<td>Hangup received on the fildes field.</td>
</tr>
<tr>
<td>ETIME</td>
<td>Time out before acknowledgment message was received at stream head.</td>
</tr>
<tr>
<td>EAGAIN</td>
<td>Temporarily unable to allocate storage to perform the I_LINK operation.</td>
</tr>
<tr>
<td>ENOSR</td>
<td>Unable to allocate storage to perform the I_LINK operation due to insufficient STREAMS memory resources.</td>
</tr>
<tr>
<td>EBADF</td>
<td>The arg parameter is not a valid, open file descriptor.</td>
</tr>
<tr>
<td>EINVAL</td>
<td>The specified link operation would cause a cycle in the resulting configuration; that is, if a given stream head is linked into a multiplexing configuration in more than one place.</td>
</tr>
</tbody>
</table>

An I_LINK operation can also fail while waiting for the multiplexing driver to acknowledge the link request, if a message indicating an error or a hangup is received at the stream head of the fildes parameter. In addition, an error code can be returned in the positive or negative acknowledgment message. For these cases, the I_LINK operation fails with the errno global variable set to the value in the message.

**I_LIST streamio Operation**

**Purpose**

Lists all the module names on a stream.

**Description**

The I_LIST operation lists all of the modules present on a stream, including the topmost driver name. If the value of the arg parameter is null, the I_LIST operation returns the number of modules on the stream pointed to by the fildes parameter. If the value of the arg parameter is nonnull, it points to an str_list structure that contains the following elements:

```c
int sl_nmods;
struct str_mlist *sl_modlist;
```

The str_mlist structure contains the following element:

```c
char l_name[FMNAMESZ+1];
```

The fields are defined as follows:

<table>
<thead>
<tr>
<th>Field</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>sl_nmods</td>
<td>Specifies the number of entries the user has allocated in the array.</td>
</tr>
<tr>
<td>sl_modlist</td>
<td>Contains the list of module names (on return).</td>
</tr>
</tbody>
</table>

This operation is part of STREAMS Kernel Extensions.
Error Codes
If unsuccessful, the errno global variable is set to one of the following values:

<table>
<thead>
<tr>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EAGAIN</td>
<td>Unable to allocate buffers.</td>
</tr>
<tr>
<td>EINVAL</td>
<td>The s1_nmods member is less than 1.</td>
</tr>
</tbody>
</table>

I_LOOK streamio Operation

Purpose
Retrieves the name of the module just below the stream head.

Syntax
```c
#include <sys/conf.h>
#include <stropts.h>

int ioctl(int fildes, command, arg);
int fildes, command;
```

Description
The I_LOOK operation retrieves the name of the module just below the stream head of the stream pointed to by the fildes parameter and places it in a null terminated character string pointed at by the arg parameter. The buffer pointed to by the arg parameter should be at least FMNAMESMZ + 1 bytes long.

This operation is part of STREAMS Kernel Extensions.

Error Codes
If unsuccessful, the errno global variable is set to one of the following values:

<table>
<thead>
<tr>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EFAULT</td>
<td>The arg parameter points outside the allocated address space.</td>
</tr>
<tr>
<td>EINVAL</td>
<td>No module is present in stream.</td>
</tr>
</tbody>
</table>

I_NREAD streamio Operation

Purpose
Counts the number of data bytes in data blocks in the first message on the stream-head read queue, and places this value in a specified location.

Description
The I_NREAD operation counts the number of data bytes in data blocks in the first message on the stream-head read queue, and places this value in the location pointed to by the arg parameter.

This operation is part of STREAMS Kernel Extensions.

Return Values
The return value for the operation is the number of messages on the stream-head read queue. For example, if a value of 0 is returned in the arg parameter, but the ioctl operation return value is greater than 0, this indicates that a zero-length message is next on the queue.
### Error Codes
If unsuccessful, the `errno` global variable is set to the following value:

<table>
<thead>
<tr>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EFAULT</td>
<td>The <code>arg</code> parameter points outside the allocated address space.</td>
</tr>
</tbody>
</table>

### I_PEEK streamio Operation

**Purpose**

Allows a user to retrieve the information in the first message on the stream-head read queue without taking the message off the queue.

**Description**

The `I_PEEK` operation allows a user to retrieve the information in the first message on the stream-head read queue without taking the message off the queue. The `arg` parameter points to a `strpeek` structure that contains the following elements:

```c
struct strbuf ctlbuf;
struct strbuf databuf;
long flags;
```

The `maxlen` field in the `strbuf` structures of the `ctlbuf` and `databuf` fields must be set to the number of bytes of control information or data information, respectively, to retrieve. If the user sets the `flags` field to `RS_HIPRI`, the `I_PEEK` operation looks for a priority message only on the stream-head read queue.

The `I_PEEK` operation returns a value of 1 if a message was retrieved, and returns a value of 0 if no message was found on the stream-head read queue, or if the `RS_HIPRI` flag was set in the `flags` field and a priority message was not present on the stream-head read queue. It does not wait for a message to arrive.

On return, the fields contain the following data:

<table>
<thead>
<tr>
<th>Data</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>ctlbuf</code></td>
<td>Specifies information in the control buffer.</td>
</tr>
<tr>
<td><code>databuf</code></td>
<td>Specifies information in the data buffer.</td>
</tr>
<tr>
<td><code>flags</code></td>
<td>Contains the value of 0 or <code>RS_HIPRI</code>.</td>
</tr>
</tbody>
</table>

This operation is part of STREAMS Kernel Extensions.

### Error Codes
If unsuccessful, the `errno` global variable is set to one of the following values:

<table>
<thead>
<tr>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EFAULT</td>
<td>The <code>arg</code> parameter points, or the buffer area specified in the <code>ctlbuf</code> or <code>databuf</code> field is outside the allocated address space.</td>
</tr>
<tr>
<td>EBADMSG</td>
<td>Queued message is not valid for the <code>I_PEEK</code> operation.</td>
</tr>
</tbody>
</table>

### I_PLINK streamio Operation

**Purpose**

Connects two specified streams.
Description

The **I_PLINK** operation is used for connecting multiplexed STREAMS configurations with a permanent link. This operation is part of STREAMS Kernel Extensions.

The **I_PLINK** operation connects two streams, where the `fildes` parameter is the file descriptor of the stream connected to the multiplexing driver, and the `arg` parameter is the file descriptor of the stream connected to another driver. The stream designated by the `arg` parameter gets connected by a permanent link below the multiplexing driver. The **I_PLINK** operation requires the multiplexing driver to send an acknowledgment message to the stream head regarding the linking operation. This call creates a permanent link which can exist even if the file descriptor associated with the upper stream to the multiplexing driver is closed. This call returns a multiplexer ID number (an identifier used to disconnect the multiplexer; see the **I_PUNLINK** operation) on success, and a value of -1 on failure.

Error Codes

If unsuccessful, the `errno` global variable is set to one of the following values:

<table>
<thead>
<tr>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ENXIO</td>
<td>Hangup received on the <code>fildes</code> field.</td>
</tr>
<tr>
<td>ETIME</td>
<td>Time out occurred before acknowledgment message was received at stream head.</td>
</tr>
<tr>
<td>EAGAIN</td>
<td>Unable to allocate storage to perform the <strong>I_PLINK</strong> operation.</td>
</tr>
<tr>
<td>EBADF</td>
<td>The <code>arg</code> parameter is not a valid, open file descriptor.</td>
</tr>
<tr>
<td>EINVAL</td>
<td>The <code>fildes</code> parameter does not support multiplexing. OR The <code>fildes</code> parameter is the file descriptor of a pipe or FIFO. OR The <code>arg</code> parameter is not a stream or is already linked under a multiplexer. OR The specified link operation would cause a cycle in the resulting configuration; that is, if a given stream head is linked into a multiplexing configuration in more than one place.</td>
</tr>
</tbody>
</table>

An **I_PLINK** operation can also be unsuccessful while waiting for the multiplexing driver to acknowledge the link request, if a message indicating an error or a hangup is received at the stream head of the `fildes` parameter. In addition, an error code can be returned in the positive or negative acknowledgment message. For these cases, the **I_PLINK** operation is unsuccessful with the `errno` global variable set to the value in the message.

**I_POP** streamio Operation

Purpose

Removes the module just below the stream head.

Description

The **I_POP** operation removes the module just below the stream head of the stream pointed to by the `fildes` parameter. The value of the `arg` parameter should be 0 in an **I_POP** request.

This operation is part of STREAMS Kernel Extensions.
Error Codes
If unsuccessful, the errno global variable is set to one of the following values:

<table>
<thead>
<tr>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EINVAL</td>
<td>No module is present in the stream.</td>
</tr>
<tr>
<td>ENXIO</td>
<td>Hangup received on the fildes parameter.</td>
</tr>
</tbody>
</table>

I_PUNLINK streamio Operation

Purpose
Disconnects the two specified streams.

Description
The I_PUNLINK operation is used for disconnecting Multiplexed STREAMS configurations connected by a permanent link.

The I_PUNLINK operation disconnects the two streams specified by the fildes parameter and the arg parameter that are connected with a permanent link. The fildes parameter is the file descriptor of the stream connected to the multiplexing driver. The arg parameter is the multiplexer ID number that was returned by the I_PLINK operation. If the value of the arg parameter is MUXID_ALL, then all streams which are permanently linked to the stream specified by the fildes parameter are disconnected. As in the I_PLINK operation, this operation requires the multiplexing driver to acknowledge the unlink.

This operation is part of STREAMS Kernel Extensions.

Error Codes
If unsuccessful, the errno global variable is set to one of the following values:

<table>
<thead>
<tr>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ENXIO</td>
<td>Hangup received on the fildes parameter.</td>
</tr>
<tr>
<td>ETIME</td>
<td>Time out occurred before acknowledgment message was received at stream head.</td>
</tr>
<tr>
<td>EINVAL</td>
<td>The arg parameter is an invalid multiplexer ID number.</td>
</tr>
</tbody>
</table>

OR

The fildes parameter is the file descriptor of a pipe or FIFO.

An I_PUNLINK operation can also be unsuccessful while waiting for the multiplexing driver to acknowledge the link request, if a message indicating an error or a hangup is received at the stream head of the fildes parameter. In addition, an error code can be returned in the positive or negative acknowledgment message. For these cases, the I_PUNLINK operation is unsuccessful and the errno global variable is set to the value in the message.

I_PUSH streamio Operation

Purpose
Pushes a module onto the top of the current stream.

Description
The I_PUSH operation pushes the module whose name is pointed to by the arg parameter onto the top of the current stream, just below the stream head. It then calls the open routine of the newly-pushed module.
This operation is part of STREAMS Kernel Extensions.

**Error Codes**

If unsuccessful, the `errno` global variable is set to one of the following values:

<table>
<thead>
<tr>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EINVAL</td>
<td>Incorrect module name.</td>
</tr>
<tr>
<td>EFAULT</td>
<td>The <code>arg</code> parameter points outside the allocated address space.</td>
</tr>
<tr>
<td>ENXIO</td>
<td>Open routine of new module failed.</td>
</tr>
<tr>
<td>ENXIO</td>
<td>Hangup received on the <code>fildes</code> parameter.</td>
</tr>
</tbody>
</table>

---

### I_RECVFD streamio Operation

**Purpose**

Retrieves the file descriptor associated with the message sent by an `I_SENDFD` operation over a stream pipe.

**Description**

The `I_RECVFD` operation retrieves the file descriptor associated with the message sent by an `I_SENDFD` operation over a stream pipe. The `arg` parameter is a pointer to a data buffer large enough to hold an `strrecvfd` data structure containing the following elements:

```c
int fd;
unsigned short uid;
unsigned short gid;
char fill[8];
```

The fields of the `strrecvfd` structure are defined as follows:

<table>
<thead>
<tr>
<th>Field</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>fd</td>
<td>Specifies an integer file descriptor.</td>
</tr>
<tr>
<td>uid</td>
<td>Specifies the user ID of the sending stream.</td>
</tr>
<tr>
<td>gid</td>
<td>Specifies the group ID of the sending stream.</td>
</tr>
</tbody>
</table>

If the `O_NDELAY` flag is not set, the `I_RECVFD` operation blocks until a message is present at the stream head. If the `O_NDELAY` flag is set, the `I_RECVFD` operation fails with the `errno` global variable set to `EAGAIN` if no message is present at the stream head.

If the message at the stream head is a message sent by an `I_SENDFD` operation, a new user file descriptor is allocated for the file pointer contained in the message. The new file descriptor is place in the `fd` field of the `strrecvfd` structure. The structure is copied into the user data buffer pointed to by the `arg` parameter.

This operation is part of STREAMS Kernel Extensions.

**Error Codes**

If unsuccessful, the `errno` global variable is set to one of the following values:

<table>
<thead>
<tr>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EAGAIN</td>
<td>A message was not present at the stream head read queue, and the <code>O_NDELAY</code> flag is set.</td>
</tr>
<tr>
<td>EBADMSG</td>
<td>The message at the stream head read queue was not a message containing a passed file descriptor.</td>
</tr>
<tr>
<td>EFAULT</td>
<td>The <code>arg</code> parameter points outside the allocated address space.</td>
</tr>
<tr>
<td>EMFILE</td>
<td>The <code>NOFILES</code> file descriptor is currently open.</td>
</tr>
<tr>
<td>Value</td>
<td>Description</td>
</tr>
<tr>
<td>--------</td>
<td>--------------------------------------------------</td>
</tr>
<tr>
<td>ENXIO</td>
<td>Hangup received on the fildes parameter.</td>
</tr>
</tbody>
</table>

**I_SENDFD streamio Operation**

**Purpose**
Requests a stream to send a message to the stream head at the other end of a stream pipe.

**Description**
The `I_SENDFD` operation requests the stream associated with the `fildes` field to send a message containing a file pointer, to the stream head at the other end of a stream pipe. The file pointer corresponds to the `arg` parameter, which must be an integer file descriptor.

The `I_SENDFD` operation converts the `arg` parameter into the corresponding system file pointer. It allocates a message block and inserts the file pointer in the block. The user ID and group ID associated with the sending process are also inserted. This message is placed directly on the read queue of the stream head at the other end of the stream pipe to which it is connected.

This operation is part of STREAMS Kernel Extensions.

**Error Codes**
If unsuccessful, the `errno` global variable is set to one of the following values:

<table>
<thead>
<tr>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EAGAIN</td>
<td>The sending stream is unable to allocate a message block to contain the file pointer.</td>
</tr>
<tr>
<td>EAGAIN</td>
<td>The read queue of the receiving stream head is full and cannot accept the message sent by the <code>I_SENDFD</code> operation.</td>
</tr>
<tr>
<td>EBADF</td>
<td>The <code>arg</code> parameter is not a valid, open file descriptor.</td>
</tr>
<tr>
<td>EINVAL</td>
<td>The <code>fildes</code> parameter is not connected to a stream pipe.</td>
</tr>
<tr>
<td>ENXIO</td>
<td>Hangup received on the <code>fildes</code> parameter.</td>
</tr>
</tbody>
</table>

**I_SETCLTIME streamio Operation**

**Purpose**
Sets the time that the stream head delays when a stream is closing.

**Description**
The `I_SETCLTIME` operation sets the time that the stream head delays when a stream is closing and there is data on the write queues. Before closing each module and driver, the stream head delays closing for the specified length of time to allow the data to be written. Any data left after the delay is flushed.

The `arg` parameter contains a pointer to the number of milliseconds to delay. This number is rounded up to the nearest legal value on the system. The default delay time is 15 seconds.

This operation is part of STREAMS Kernel Extensions.
Error Codes

If unsuccessful, the `errno` global variable is set to the following value:

<table>
<thead>
<tr>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EINVAL</td>
<td>The value in the <code>arg</code> parameter is invalid.</td>
</tr>
</tbody>
</table>

I_SETSIG streamio Operation

Purpose

Informs the stream head that the user wishes the kernel to issue the `SIGPOLL` signal when a particular event occurs on the stream.

Description

The `I_SETSIG` operation informs the stream head that the user wishes the kernel to issue the `SIGPOLL` signal (see the `signal` and `sigset` subroutines) when a particular event has occurred on the stream associated with the `fildes` parameter. The `I_SETSIG` operation supports an asynchronous processing capability in STREAMS. The value of the `arg` parameter is a bit mask that specifies the events for which the user should be signaled. It is the bitwise-OR of any combination of the following constants:

<table>
<thead>
<tr>
<th>Constant</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>S_INPUT</td>
<td>A nonpriority message has arrived on a stream-head read queue, and no other messages existed on that queue before this message was placed there. This is set even if the message is of zero length.</td>
</tr>
<tr>
<td>S_HIPRI</td>
<td>A priority message is present on the stream-head read queue. This is set even if the message is of zero length.</td>
</tr>
<tr>
<td>S_OUTPUT</td>
<td>The write queue just below the stream head is no longer full. This notifies the user that there is room on the queue for sending (or writing) data downstream.</td>
</tr>
<tr>
<td>S_MSG</td>
<td>A STREAMS signal message that contains the <code>SIGPOLL</code> signal has reached the front of the stream-head read queue.</td>
</tr>
</tbody>
</table>

A user process may choose to be signaled only by priority messages by setting the `arg` bit mask to the value `S_HIRPI`.

Processes that wish to receive `SIGPOLL` signals must explicitly register to receive them using `I_SETSIG`. If several processes register to receive this signal for the same event on the same stream, each process will be signaled when the event occurs.

If the value of the `arg` parameter is 0, the calling process is unregistered and does not receive further `SIGPOLL` signals.

This operation is part of STREAMS Kernel Extensions.

Error Codes

If unsuccessful, the `errno` global variable is set to one of the following values:

<table>
<thead>
<tr>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EINVAL</td>
<td>The value for the <code>arg</code> parameter is invalid or 0 and process is not registered to receive the <code>SIGPOLL</code> signal.</td>
</tr>
<tr>
<td>EAGAIN</td>
<td>The allocation of a data structure to store the signal request is unsuccessful.</td>
</tr>
</tbody>
</table>
I_SRDOPT streamio Operation

Purpose
Sets the read mode.

Description
The I_SRDOPT operation sets the read mode using the value of the arg parameter. Legal values for the arg parameter are:

<table>
<thead>
<tr>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>RNORM</td>
<td>Byte-stream mode. This is the default mode.</td>
</tr>
<tr>
<td>RMSGD</td>
<td>Message-discard mode.</td>
</tr>
<tr>
<td>RMSGN</td>
<td>Message-nondiscard mode.</td>
</tr>
<tr>
<td>RFILL</td>
<td>Read mode. This mode prevents completion of any read request until one of three conditions occurs:</td>
</tr>
<tr>
<td></td>
<td>• The entire user buffer is filled.</td>
</tr>
<tr>
<td></td>
<td>• An end of file occurs.</td>
</tr>
<tr>
<td></td>
<td>• The stream head receives an M_MI_READ_END message.</td>
</tr>
</tbody>
</table>

Several control messages support the RFILL mode. They are used by modules to manipulate data being placed in user buffers at the stream head. These messages are multiplexed under a single M_MI message type. The message subtype, pointed to by the b_rptr parameter, is one of the following:

M_MI_READ_SEEK
Provides random access data retrieval. An application and a cooperating module can gather large data blocks from a slow, high-latency, or unreliable link, while minimizing the number of system calls required, and relieving the protocol modules of large buffering requirements.

The M_MI_READ_SEEK message subtype is followed by two long words, as in a standard seek call. The first word is an origin indicator as follows:

0  Start of buffer
1  Current position
2  End of buffer

The second word is a signed offset from the specified origin.

M_MI_READ_RESET
Discards any data previously delivered to partially satisfy an RFILL mode read request.

M_MI_READ_END
Completes the current RFILL mode read request with whatever data has already been delivered.

In addition, treatment of control messages by the stream head can be changed by setting the following flags in the arg parameter:

<table>
<thead>
<tr>
<th>Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>RPROTNORM</td>
<td>Causes the read routine to be unsuccessful if a control message is at the front of the stream-head read queue.</td>
</tr>
<tr>
<td>RPROTDAT</td>
<td>Delivers the control portion of a message as data.</td>
</tr>
<tr>
<td>RPROTDIS</td>
<td>Discards the control portion of a message, delivering any data portion.</td>
</tr>
</tbody>
</table>

This operation is part of STREAMS Kernel Extensions.
Error Codes
If unsuccessful, the *errno* global variable is set to the following value:

<table>
<thead>
<tr>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EINVAL</td>
<td>The value of the <em>arg</em> parameter is not one of the above legal values.</td>
</tr>
</tbody>
</table>

I_STR streamio Operation

Purpose
Constructs an internal STREAMS ioctl message.

Description
The I_STR operation constructs an internal STREAMS ioctl message from the data pointed to by the *arg* parameter and sends that message downstream.

This mechanism is provided to send user ioctl requests to downstream modules and drivers. It allows information to be sent with the ioctl and returns to the user any information sent upstream by the downstream recipient. The I_STR operation blocks until the system responds with either a positive or negative acknowledgment message or until the request times out after some period of time. If the request times out, it fails with the *errno* global variable set to ETIME.

At most, one I_STR operation can be active on a stream. Further I_STR operation calls block until the active I_STR operation completes at the stream head. The default timeout interval for this request is 15 seconds. The O_NDELAY flag has no effect on this call.

To send a request downstream, the *arg* parameter must point to a strioctl structure that contains the following elements:

```c
int ic_cmd;   /* downstream operation */
int ic_timeout; /* ACK/NAK timeout */
int ic_len;   /* length of data arg */
char *ic_dp;  /* ptr to data arg */
```

The elements of the strioctl structure are described as follows:

<table>
<thead>
<tr>
<th>Element</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ic_cmd</td>
<td>The internal ioctl operation intended for a downstream module or driver.</td>
</tr>
<tr>
<td>ic_timeout</td>
<td>The number of seconds an I_STR request waits for acknowledgment before timing out:</td>
</tr>
<tr>
<td></td>
<td>-1   waits an infinite number of seconds.</td>
</tr>
<tr>
<td></td>
<td>0    uses default value.</td>
</tr>
<tr>
<td></td>
<td>&gt; 0  waits the specified number of seconds.</td>
</tr>
<tr>
<td>ic_len</td>
<td>The number of bytes in the data argument. The ic_len field has two uses:</td>
</tr>
<tr>
<td></td>
<td>• On input, it contains the length of the data argument passed in.</td>
</tr>
<tr>
<td></td>
<td>• On return from the operation, it contains the number of bytes being returned to the user (the buffer pointed to by the ic_dp field should be large enough to contain the maximum amount of data that any module or the driver in the stream can return).</td>
</tr>
<tr>
<td>ic_dp</td>
<td>A pointer to the data parameter.</td>
</tr>
</tbody>
</table>

The stream head converts the information pointed to by the strioctl structure to an internal ioctl operation message and sends it downstream.

This operation is part of STREAMS Kernel Extensions.
### Error Codes

If unsuccessful, the `errno` global variable is set to one of the following values:

<table>
<thead>
<tr>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EAGAIN</td>
<td>The value of <code>ic_len</code> is greater than the maximum size of a message block returned by the STREAMS <code>allocb</code> utility, or there is insufficient memory for a message block.</td>
</tr>
<tr>
<td>ENOSR</td>
<td>Unable to allocate buffers for the ioctl message due to insufficient STREAMS memory resources.</td>
</tr>
<tr>
<td>EFAULT</td>
<td>The area pointed to by the <code>arg</code> parameter or the buffer area specified by the <code>ic_dp</code> and <code>ic_len</code> fields (for data sent and data returned, respectively) is outside of the allocated address space.</td>
</tr>
<tr>
<td>EINVAL</td>
<td>The value of the <code>ic_len</code> field is less than 0 or greater than the maximum configured size of the data part of a message, or the value of the <code>ic_timeout</code> field is less than -1.</td>
</tr>
<tr>
<td>ENXIO</td>
<td>Hangup received on the <code>fildes</code> field.</td>
</tr>
<tr>
<td>ETIME</td>
<td>A downstream <code>streamio</code> operation timed out before acknowledgment was received.</td>
</tr>
</tbody>
</table>

An **I_STR** operation can also be unsuccessful while waiting for an acknowledgment if a message indicating an error or a hangup is received at the stream head. In addition, an error code can be returned in the positive or negative acknowledgment messages, in the event that the `streamio` operation sent downstream fails. For these cases, the **I_STR** operation is unsuccessful and the `errno` global variable is set to the value in the message.

### I_UNLINK streamio Operation

#### Purpose

Disconnects the two specified streams.

#### Description

The **I_UNLINK** operation is used for disconnecting multiplexed STREAMS configurations.

This operation is part of STREAMS Kernel Extensions.

The **I_UNLINK** operation disconnects the two streams specified by the `fildes` parameter and the `arg` parameter. The `fildes` parameter is the file descriptor of the stream connected to the multiplexing driver. The `fildes` parameter must correspond to the stream on which the `ioctl` **I_LINK** operation was issued to link the stream below the multiplexing driver. The `arg` parameter is the multiplexer ID number that was returned by the **I_LINK** operation. If the value of the `arg` parameter is -1, then all streams that were linked to the `fildes` parameter are disconnected. As in the **I_LINK** operation, this operation requires the multiplexing driver to acknowledge the unlink.

#### Error Codes

If unsuccessful, the `errno` global variable is set to one of the following values:

<table>
<thead>
<tr>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ENXIO</td>
<td>Hangup received on the <code>fildes</code> parameter.</td>
</tr>
<tr>
<td>ETIME</td>
<td>Time out before acknowledgment message was received at stream head.</td>
</tr>
<tr>
<td>ENOSR</td>
<td>Unable to allocate storage to perform the <strong>I_UNLINK</strong> operation due to insufficient STREAMS memory resources.</td>
</tr>
<tr>
<td>EINVAL</td>
<td>The <code>arg</code> parameter is an invalid multiplexer ID number or the <code>fildes</code> parameter is not the stream on which the <strong>I_LINK</strong> operation that returned the <code>arg</code> parameter was performed.</td>
</tr>
</tbody>
</table>

An **I_UNLINK** operation can also fail while waiting for the multiplexing driver to acknowledge the link request, if a message indicating an error or a hangup is received at the stream head of the `fildes`
parameter. In addition, an error code can be returned in the positive or negative acknowledgment message. For these cases, the `I_UNLINK` operation fails and the `errno` global variable is set to the value in the message.

**Related Information**
The `I_LINK` streamio operation, `I_PUNLINK` streamio operation.

List of Streams Programming References
- Understanding streamio (STREAMS ioctl) Operations
- Understanding STREAMS Drivers and Modules
- Understanding STREAMS Messages

List of Streams Programming References in AIX 5L Version 5.3 Communications Programming Concepts.

---

**isastream Function**

**Purpose**
Tests a file descriptor.

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

**Syntax**
```
int isastream(int fildes);
```

**Description**
The `isastream` subroutine determines if a file descriptor represents a STREAMS file.

**Parameters**
- `fildes` Specifies which open file to check.

**Return Values**
On successful completion, the `isastream` subroutine returns a value of 1 if the `fildes` parameter represents a STREAMS file, or a value of 0 if not. Otherwise, it returns a value of -1 and sets the `errno` global variable to indicate the error.

**Error Codes**
If unsuccessful, the `errno` global variable is set to the following value:

<table>
<thead>
<tr>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EBADF</td>
<td>The <code>fildes</code> parameter does not specify a valid open file.</td>
</tr>
</tbody>
</table>

**Related Information**
- `streamio` operations.

List of Streams Programming References in AIX 5L Version 5.3 Communications Programming Concepts.

---

**linkb Utility**

**Purpose**
Concatenates two messages into one.
Syntax

```c
void link(mp, bp)
register mblk_t *mp;
register mblk_t *bp;
```

Description

The `linkb` utility puts the message pointed to by the `bp` parameter at the tail of the message pointed to by the `mp` parameter. This results in a single message.

This utility is part of STREAMS Kernel Extensions.

Parameters

- `mp` Specifies the message to which the second message is to be linked.
- `bp` Specifies the message that is to be linked to the end of first message.

Related Information

The `unlinkb` utility.

[List of Streams Programming References](#) and [Understanding STREAMS Messages](#) in *AIX 5L Version 5.3 Communications Programming Concepts*.

---

**mi_bufcall Utility**

Purpose

Provides a reliable alternative to the `bufcall` utility.

Syntax

```c
#include <pse/mi.h>
#include <sys/stream.h>

void mi_bufcall(Queue, Size, Priority)
queue_t *Queue;
int Size;
int Priority;
```

Description

The `mi_bufcall` utility provides a reliable alternative to the `bufcall` utility. The standard STREAMS `bufcall` utility is intended to be called when the `allocb` utility is unable to allocate a block for a message, and invokes a specified callback function (typically the `qenable` utility) with a given queue when a large enough block becomes available. This can cause system problems if the stream closes so that the queue becomes invalid before the callback function is invoked.

The `mi_bufcall` utility is a reliable alternative, as the queue is not deallocated until the call is complete. This utility uses the standard `bufcall` mechanism with its own internal callback routine. The callback routine either invokes the `qenable` utility with the specified `Queue` parameter, or simply deallocates the instance data associated with the stream if the queue has already been closed.

The `mi_bufcall` utility is part of STREAMS kernel extensions.

**Note:** The `stream.h` header file must be the last included header file of each source file using the stream library.
Parameters

**Queue**
Specifies the queue which is to be passed to the *qenable* utility.

**Size**
Specifies the required buffer size.

**Priority**
Specifies the priority as used by the standard STREAMS *bufcall* mechanism.

Related Information

- [List of Streams Programming References](#) in *AIX 5L Version 5.3 Communications Programming Concepts* .
- [STREAMS Overview](#) in *AIX 5L Version 5.3 Communications Programming Concepts* .

The *bufcall* utility, **mi_close_comm** utility, **mi_next_ptr** utility, **mi_open_comm** utility.

---

### mi_close_comm Utility

#### Purpose
Performs housekeeping during STREAMS driver or module close operations.

#### Syntax

```c
#include <pses/mi.h>
#include <sys/stream.h>

int mi_close_comm (StaticPointer, Queue)
caddr_t *StaticPointer;
queue_t *Queue;
```

#### Description

The **mi_close_comm** utility performs housekeeping during STREAMS driver or module close operations. It is intended to be called by the driver or module *close* routine. It releases the memory allocated by the corresponding call to the **mi_open_comm** utility, and frees the minor number for reuse.

If an **mi_bufcall** operation is outstanding, module resources are not freed until the **mi_bufcall** operation is complete.

The **mi_close_comm** utility is part of STREAMS kernel extensions.

#### Notes:

1. Each call to the **mi_close_comm** utility must have a corresponding call to the **mi_open_comm** utility. Executing one of these utilities without making a corresponding call to the other will lead to unpredictable results.

2. The **stream.h** header file must be the last included header file of each source file using the stream library.

#### Parameters

**StaticPointer**
Specifies the address of the static pointer which was passed to the corresponding call to the **mi_open_comm** utility to store the address of the module’s list of open streams.

**Queue**
Specifies the **Queue** parameter which was passed to the corresponding call to the **mi_open_comm** utility.
Return Values
The `mi_close_comm` utility always returns a value of zero.

Related Information
- List of Streams Programming References in *AIX 5L Version 5.3 Communications Programming Concepts*.
- STREAMS Overview in *AIX 5L Version 5.3 Communications Programming Concepts*.

The `mi_open_comm` utility, `mi_next_ptr` utility, `mi_bufcall` utility.

### mi_next_ptr Utility

**Purpose**
Traverses a STREAMS module's linked list of open streams.

**Syntax**
```c
#include <pse/mi.h>
#include <sys/stream.h>

caddr_t mi_next_ptr (Origin)
caddr_t Origin;
```

**Description**
The `mi_next_ptr` utility traverses a module's linked list of open streams. The `Origin` argument specifies the address of a per-instance list item, and the return value indicates the address of the next item. The first time the `mi_next_ptr` utility is called, the `Origin` parameter should be initialized with the value of the static pointer which was passed to the `mi_open_comm` utility. Subsequent calls to the `mi_next_ptr` utility should pass the address which was returned by the previous call, until a `NULL` address is returned, indicating that the end of the queue has been reached.

The `mi_next_ptr` utility is part of STREAMS kernel extensions.

**Note:** The `stream.h` header file must be the last included header file of each source file using the stream library.

**Parameter**
- `Origin` Specifies the address of the current list item being examined.

**Return Values**
The `mi_next_ptr` utility returns the address of the next list item, or `NULL` if the end of the list has been reached.

**Related Information**
- List of Streams Programming References in *AIX 5L Version 5.3 Communications Programming Concepts*.
- STREAMS Overview in *AIX 5L Version 5.3 Communications Programming Concepts*.

The `mi_close_comm` utility, `mi_open_comm` utility, `mi_bufcall` utility.
mi_open_comm Utility

Purpose
Performs housekeeping during STREAMS driver or module open operations.

Syntax
```
#include <pse/mi.h>
#include <sys/stream.h>

int mi_open_comm ( StaticPointer, Size, Queue, Device, Flag, SFlag, credp);
```

Description
The mi_open_comm subroutine performs housekeeping during STREAMS driver or module open operations. It is intended to be called by the driver or module open routine. It assigns a minor device number to the stream (as specified by the SFlag parameter), allocates the requested per-stream data, and sets the q_ptr fields of the stream being opened.

The mi_open_comm subroutine is part of STREAMS kernel extensions.

Notes:
1. Each call to the mi_open_comm subroutine must have a corresponding call to the mi_close_comm subroutine. Executing one of these utilities without making a corresponding call to the other will lead to unpredictable results.
2. The stream.h header file must be the last included header file of each source file using the stream library.

Parameters
- **StaticPointer**: Specifies the address of a static pointer which will be used internally by the mi_open_comm and related utilities to store the address of the module’s list of open streams. This pointer should be initialized to NULL.
- **Size**: Specifies the amount of memory the module needs for its per-stream data. It is usually the size of the local structure which contains the module’s instance data.
- **Queue**: Specifies the address of a queue_t structure. The q_ptr field of the of this structure, and of the corresponding read queue structure (if Queue points to a write queue) or write queue structure (if Queue points to a read queue), are filled in with the address of the queue_t structure being initialized.
- **Device**: Specifies the address of a dev_t structure. The use of this parameter depends on the value of the SFlag parameter.
- **Flag**: Unused.
Specifies how the Device parameter is to be used. The SFlag parameter may take one of the following values:

**DEVOPEN**
The minor device number specified by the Device argument is used.

**MODOPEN**
The Device parameter is NULL. This value should be used if the mi_open_com subroutine is called from the open routine of a STREAMS module rather than a STREAMS driver.

**CLONEOPEN**
A unique minor device number above 5 is assigned (minor numbers 0-5 are reserved as special access codes).

Unused

**Return Values**
On successful completion, the mi_open_com subroutine returns a value of zero, otherwise one of the following codes is returned:

<table>
<thead>
<tr>
<th>Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ENXIO</td>
<td>Indicates an invalid parameter.</td>
</tr>
<tr>
<td>EAGAIN</td>
<td>Indicates that an internal structure could not be allocated, and that the call should be retried.</td>
</tr>
</tbody>
</table>

**Related Information**

- List of Streams Programming References in *AIX 5L Version 5.3 Communications Programming Concepts*.
- STREAMS Overview in *AIX 5L Version 5.3 Communications Programming Concepts*.

The mi_close_com subroutine, mi_next_ptr subroutine, mi_bufcall subroutine.

**msgdsize Utility**

**Purpose**
Gets the number of data bytes in a message.

**Syntax**

```c
int msgdsize(bp)
register mblk_t *bp;
```

**Description**
The msgdsize utility returns the number of bytes of data in the message pointed to by the bp parameter. Only bytes included in data blocks of type M_DATA are included in the total.

This utility is part of STREAMS Kernel Extensions.

**Parameters**

- **bp** Specifies the message from which to get the number of bytes.
Return Values
The `msgdsize` utility returns the number of bytes of data in a message.

Related Information
List of Streams Programming References | Understanding STREAMS Messages in AIX 5L Version 5.3 Communications Programming Concepts.

noenable Utility

Purpose
Prevents a queue from being scheduled.

Syntax
```c
void noenable(q)
queue_t * q;
```

Description
The `noenable` utility prevents the queue specified by the `q` parameter from being scheduled for service either by the `putq` or `putbq` utility, when these routines queue an ordinary priority message, or by the `insq` utility when it queues any message. The `noenable` utility does not prevent the scheduling of queues when a high-priority message is queued, unless the message is queued by the `insq` utility.

This utility is part of STREAMS Kernel Extensions.

Parameters
- `q` Specifies the queue to disable.

Related Information
The `enableok` utility, `insq` utility, `putbq` utility, `putq` utility.

OTHERQ Utility

Purpose
Returns the pointer to the mate queue.

Syntax
```c
#define OTHERQ((q) -> flag&QREADER? (q)+1: (q)-1)
```

Description
The `OTHERQ` utility returns a pointer to the mate queue of the `q` parameter.

This utility is part of STREAMS Kernel Extensions.
Parameters

$q$  Specifies that queue whose mate is to be returned.

Return Values

If the $q$ parameter specifies the read queue for the module, the OTHERQ utility returns a pointer to the module's write queue. If the $q$ parameter specifies the write queue for the module, this utility returns a pointer to the read queue.

Related Information

The RD utility, WR utility.

List of Streams Programming References and Understanding STREAMS Messages in AIX 5L Version 5.3 Communications Programming Concepts.

pfmod Packet Filter Module

Purpose

Selectively removes upstream data messages on a Stream.

Synopsis

```c
#include <stropts.h>
#include <sys/pfmod.h>

ioctl(fd, I_PUSH, "pfmod");
```

Description

The pfmod module implements a programmable packet filter facility that may be pushed over any stream. Every data message that pfmod receives on its read side is subjected to a filter program. If the filter program accepts a message, it will be passed along upstream, and will otherwise be freed. If no filter program has been set (as is the case when pfmod is first pushed), all messages are accepted. Non-data messages (for example, M_FLUSH, M_PCPROTO, M_IOCACK) are never examined and always accepted. The write side is not filtered.

Data messages are defined as either M_PROTO or M_DATA. If an M_PROTO message is received, pfmod will skip over all the leading blocks until it finds an M_DATA block. If none is found, the message is accepted. The M_DATA portion of the message is then made contiguous with pullupmsg(), if necessary, to ensure the data area referenced by the filter program can be accessed in a single mblk_t.

IOCTLS

The following ioctls are defined for this module. All other ioctls are passed downstream without examination.

PFIOCSETF

Install a new filter program, replacing any previous program. It uses the following data structure:

```c
typedef struct packetfilt {
  uchar  Pf_Priority;
  uchar  Pf_FilterLen;
  ushort Pf_Filter[MAXFILTERS];
} pfilter_t;
```
Pf_Priority is currently ignored, and should be set to zero. Pf_FilterLen indicates the number of shortwords in the Pf_Filter array. Pf_Filter is an array of shortwords that comprise the filter program. See [Filters](#) for details on how to write filter programs.

This ioctl may be issued either transparently or as an I_STR. It will return 0 on success, or -1 on failure, and set errno to one of:

<table>
<thead>
<tr>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ERANGE</td>
<td>The length of the M_IOCTL message data was not exactly size of (pfilt_t). The data structure is not variable length, although the filter program is.</td>
</tr>
<tr>
<td>EFAULT</td>
<td>The ioctl argument points out of bounds.</td>
</tr>
</tbody>
</table>

## Filters

A filter program consists of a linear array of shortword instructions. These instructions operate upon a stack of shortwords. Flow of control is strictly linear; there are no branches or loops. When the filter program completes, the top of the stack is examined. If it is non-zero, or if the stack is empty, the packet being examined is passed upstream (accepted), otherwise the packet is freed (rejected).

Instructions are composed of three portions: push command PF_CMD(), argument PF_ARG(), and operation PF_OP(). Each instruction optionally pushes a shortword onto the stack, then optionally performs a binary operation on the top two elements on the stack, leaving its result on the stack. If there are not at least two elements on the stack, the operation will immediately fail and the packet will be rejected. The argument portion is used only by certain push commands, as documented below.

The following push commands are defined:

<table>
<thead>
<tr>
<th>Command</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>PF_NOPUSH</td>
<td>Nothing is pushed onto the stack.</td>
</tr>
<tr>
<td>PF_PUSHZERO</td>
<td>Pushes 0x0000.</td>
</tr>
<tr>
<td>PF_PUSHONE</td>
<td>Pushes 0x0001.</td>
</tr>
<tr>
<td>PF_PUSHFFFF</td>
<td>Pushes 0xffff.</td>
</tr>
<tr>
<td>PF_PUSHFF00</td>
<td>Pushes 0xff00.</td>
</tr>
<tr>
<td>PF_PUSH00FF</td>
<td>Pushes 0x00ff.</td>
</tr>
<tr>
<td>PF_PUSHLIT</td>
<td>Pushes the next shortword in the filter program as literal data. Execution resumes with the next shortword after the literal data.</td>
</tr>
<tr>
<td>PF_PUSHWORD+N</td>
<td>Pushes shortword N of the message onto the data stack. N must be in the range 0-255, as enforced by the macro PF_ARG().</td>
</tr>
</tbody>
</table>

The following operations are defined. Each operation pops the top two elements from the stack, and pushes the result of the operation onto the stack. The operations below are described in terms of v1 and v2. The top of stack is popped into v2, then the new top of stack is popped into v1. The result of v1 op v2 is then pushed onto the stack.

<table>
<thead>
<tr>
<th>Operation</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>PF_NOP</td>
<td>The stack is unchanged; nothing is popped.</td>
</tr>
<tr>
<td>PF_EQ</td>
<td>v1 == v2</td>
</tr>
<tr>
<td>PF_NEQ</td>
<td>v1 != v2</td>
</tr>
<tr>
<td>PF_LT</td>
<td>v1 &lt; v2</td>
</tr>
<tr>
<td>PF_LE</td>
<td>v1 &lt;= v2</td>
</tr>
<tr>
<td>PF_GT</td>
<td>v1 &gt; v2</td>
</tr>
<tr>
<td>PF_GE</td>
<td>v1 &gt;= v2</td>
</tr>
<tr>
<td>PF_AND</td>
<td>v1 &amp; v2; bitwise</td>
</tr>
<tr>
<td>PF_OR</td>
<td>v1</td>
</tr>
<tr>
<td>PF_XOR</td>
<td>v1 ^ v2; bitwise</td>
</tr>
</tbody>
</table>
The remaining operations are “short-circuit” operations. If the condition checked for is found, then the filter program terminates immediately, either accepting or rejecting the packet as specified, without examining the top of stack. If the condition is not found, the filter program continues. These operators do not push any result onto the stack.

<table>
<thead>
<tr>
<th>Operation</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>PF_COR</td>
<td>If v1 == v2, accept.</td>
</tr>
<tr>
<td>PF_CNOR</td>
<td>If v1 == v2, reject.</td>
</tr>
<tr>
<td>PF_CAND</td>
<td>If v1 != v2, reject.</td>
</tr>
<tr>
<td>PF_CNAND</td>
<td>If v1 != v2, accept.</td>
</tr>
</tbody>
</table>

If an unknown push command or operation is specified, the filter program terminates immediately and the packet is rejected.

**Configuration**

Before using pfmod, it must be loaded into the kernel. This may be accomplished with the **strload** command, using the following syntax:

```
strload -m pfmod
```

This command will load the pfmod into the kernel and make it available to **I_PUSH**. Note that attempting to **I_PUSH** pfmod before loading it will result in an **EINVAL** error code.

**Example**

The following program fragment will push pfmod on a stream, then program it to only accept messages with an Ethertype of 0x8137. This example assumes the stream is a promiscuous DLPI ethernet stream (see [dlpi](#) for details).

```
#include <stddef.h>
#include <sys/types.h>
#include <netinet/if_ether.h>
#define scale(x) ((x)/sizeof(ushort))

setfilter(int fd)
{
    pfilter_t filter;
    ushort *fp, offset;
    if (ioctl(fd, I_PUSH, "pfmod"))
        return -1;
    offset = scale(offsetof(struct ether_header, ether_type));
    fp = filter.Pf_Filter;

    /* the filter program */
    *fp++ = PF_PUSHLIT;
    *fp++ = 0x8137;
    *fp++ = PF_PUSHWORD + offset;
    *fp++ = PF_EQ;

    filter.Pf_FilterLen = fp - filter.Pf_Filter;

    if (ioctl(fd, PFIOCSETF, &filter))
        return -1;
    return 0;
}
```

This program may be shortened by combining the operation with the push command:

```
*fp++ = PF_PUSHLIT;
*fp++ = 0x8137;
*fp++ = (PF_PUSHWORD + offset) | PF_EQ;
```
The following filter will accept 802.3 frames addressed to either the Netware raw sap 0xff or the 802.2 sap 0xe0:

```c
offset = scale(offsetof(struct ie3_hdr, llc));
*fp++ = PF_PUSHWORD + offset; /* get ssap, dsap */
*fp++ = PF_PUSH00FF | PF_AND; /* keep only dsap */
*fp++ = PF_PUSH00FF | PF_COR; /* is dsap == 0xff? */
*fp++ = PF_PUSHWORD + offset; /* get ssap, dsap again */
*fp++ = PF_PUSH00FF | PF_AND; /* keep only dsap */
*fp++ = PF_PUSH00FF | PF_CAND; /* is dsap == 0xe0? */
*fp++ = 0x00e0;
```

Note the use of PF_COR in this example. If the dsap is 0xff, then the frame is accepted immediately, without continuing the filter program.

**pullupmsg Utility**

**Purpose**

Concatenates and aligns bytes in a message.

**Syntax**

```c
int pullupmsg(mp, len)
    register struct msgb * mp;
    register int len;
```

**Description**

The **pullupmsg** utility concatenates and aligns the number of data bytes specified by the `len` parameter of the passed message into a single, contiguous message block. Proper alignment is hardware-dependent. The **pullupmsg** utility only concatenates across message blocks of similar type. It fails if the `mp` parameter points to a message of less than `len` bytes of similar type. If the `len` parameter contains a value of -1, the **pullupmsg** utility concatenates all blocks of the same type at the beginning of the message pointed to by the `mp` parameter.

As a result of the concatenation, the contents of the message pointed to by the `mp` parameter may be altered.

This utility is part of STREAMS Kernel Extensions.

**Parameters**

- `mp` Specifies the message that is to be aligned.
- `len` Specifies the number of bytes to align.

**Return Values**

On success, the **pullupmsg** utility returns a value of 1. On failure, it returns a value of 0.

**Related Information**

- List of Streams Programming References
- Understanding STREAMS Messages
- Communications Programming Concepts

in *AIX 5L Version 5.3* Communications Programming Concepts.
putbq Utility

Purpose
Returns a message to the beginning of a queue.

Syntax
```c
int
putbq(q, bp)
register queue_t * q;
register mblk_t * bp;
```

Description
The `putbq` utility puts the message pointed to by the `bp` parameter at the beginning of the queue pointed to by the `q` parameter, in a position in accordance with the message type. High-priority messages are placed at the head of the queue, followed by priority-band messages and ordinary messages. Ordinary messages are placed after all high-priority and priority-band messages, but before all other ordinary messages already on the queue. The queue is scheduled in accordance with the rules described in the `putq` utility. This utility is typically used to replace a message on the queue from which it was just removed.

This utility is part of STREAMS Kernel Extensions.

Note: A service procedure must never put a high-priority message back on its own queue, as this would result in an infinite loop.

Parameters
- `q` Specifies the queue on which to place the message.
- `bp` Specifies the message to place on the queue.

Return Values
The `putbq` utility returns a value of 1 on success. Otherwise, it returns a value of 0.

Related Information
The `putq` utility.

List of Streams Programming References and Understanding STREAMS Messages in AIX 5L Version 5.3 Communications Programming Concepts.

putctl1 Utility

Purpose
Passes a control message with a one-byte parameter.

Syntax
```c
int
putctl1(q, type, param)
queue_t *q;
```
Description
The putctl1 utility creates a control message of the type specified by the `type` parameter with a one-byte parameter specified by the `param` parameter, and calls the put procedure of the queue pointed to by the `q` parameter, with a pointer to the created message as an argument.

The putctl1 utility allocates new blocks by calling the allocb utility.

This utility is part of STREAMS Kernel Extensions.

Parameters

- **q**: Specifies the queue.
- **type**: Specifies the type of control message.
- **param**: Specifies the one-byte parameter.

Return Values
On successful completion, the putctl1 utility returns a value of 1. It returns a value of 0 if it cannot allocate a message block, or if the value of the `type` parameter is `M_DATA`, `M_PROTO`, or `M_PCPROTO`. The `M_DELAY` type is allowed.

Related Information
The allocb utility, putctl utility.

List of Streams Programming References and Understanding STREAMS Messages in AIX 5L Version 5.3 Communications Programming Concepts.

putctl Utility

Purpose
Passes a control message.

Syntax

```
int putctl1(queue_t *q, type)
```

Description
The putctl utility creates a control message of the type specified by the `type` parameter, and calls the put procedure of the queue pointed to by the `q` parameter. The argument of the put procedure is a pointer to the created message. The putctl utility allocates new blocks by calling the allocb utility.

This utility is part of STREAMS Kernel Extensions.

Parameters

- **q**: Specifies the queue that contains the desired put procedure.
- **type**: Specifies the type of control message to create.
Return Values
On successful completion, the `putctl` utility returns a value of 1. It returns a value of 0 if it cannot allocate a message block, or if the value of the `type` parameter is `M_DATA`, `MPROTO`, `M_PCPROTO`, or `M_DELAY`.

Related Information
The `allocb` utility, `putctl` utility.

List of Streams Programming References and Understanding STREAMS Messages in AIX 5L Version 5.3 Communications Programming Concepts.

putmsg System Call

Purpose
Sends a message on a stream.

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

int putmsg (fd, ctlptr, dataptr, flags)

int fd;
struct strbuf *ctlptr;
struct strbuf *dataptr;
int flags;
```

Description
The `putmsg` system call creates a message from user-specified buffers and sends the message to a STREAMS file. The message may contain either a data part, a control part or both. The data and control parts to be sent are distinguished by placement in separate buffers. The semantics of each part is defined by the STREAMS module that receives the message.

This system call is part of STREAMS Kernel Extensions.

Parameters
- `fd` Specifies a file descriptor referencing an open stream.
- `ctlptr` Holds the control part of the message.
- `dataptr` Holds the data part of the message.
- `flags` Indicates the type of message to be sent. Acceptable values are:
  - 0 Sends a nonpriority message.
  - RS_HIPRI Sends a priority message.

The `ctlptr` and `dataptr` parameters each point to a `strbuf` structure that contains the following members:
```c
int maxlen; /* not used */
int len;    /* length of data */
char *buf;  /* ptr to buffer */
```
The len field in the strbuf structure indicates the number of bytes to be sent, and the buf field points to the buffer where the control information or data resides. The maxlen field is not used in the putmsg system call.

To send the data part of a message, the dataptr parameter must be nonnull and the len field of the dataptr parameter must have a value of 0 or greater. To send the control part of a message, the corresponding values must be set for the ctlptr parameter. No data (control) part will be sent if either the dataptr (ctlptr) parameter is null or the len field of the dataptr (ctlptr) parameter is set to -1.

If a control part is specified, and the flags parameter is set to RS_HIPRI, a priority message is sent. If the flags parameter is set to 0, a nonpriority message is sent. If no control part is specified and the flags parameter is set to RS_HIPRI, the putmsg system call fails and sets the errno global variable to EINVAL. If neither a control part nor a data part is specified and the flags parameter is set to 0, no message is sent and 0 is returned.

For nonpriority messages, the putmsg system call blocks if the stream write queue is full due to internal flow-control conditions. For priority messages, the putmsg system call does not block on this condition. For nonpriority messages, the putmsg system call does not block when the write queue is full and the O_NDELAY or O_NONBLOCK flag is set. Instead, the system call fails and sets the errno global variable to EAGAIN.

The putmsg system call also blocks, unless prevented by lack of internal resources, while waiting for the availability of message blocks in the stream, regardless of priority or whether the O_NDELAY or O_NONBLOCK flag has been specified. No partial message is sent.

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 putmsg system call fails if one of the following is true:

<table>
<thead>
<tr>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EAGAIN</td>
<td>A nonpriority message was specified, the O_NONBLOCK flag is set, and the stream write queue is full due to internal flow-control conditions.</td>
</tr>
<tr>
<td>EAGAIN</td>
<td>Buffers could not be allocated for the message that was to be created.</td>
</tr>
<tr>
<td>EBADF</td>
<td>The value of the fd parameter is not a valid file descriptor open for writing.</td>
</tr>
<tr>
<td>EFAULT</td>
<td>The ctlptr or dataptr parameter points outside the allocated address space.</td>
</tr>
<tr>
<td>EINTR</td>
<td>A signal was caught during the putmsg system call.</td>
</tr>
<tr>
<td>EINVAL</td>
<td>An undefined value was specified in the flags parameter, or the flags parameter is set to RS_HIPRI and no control part was supplied.</td>
</tr>
<tr>
<td>EINVAL</td>
<td>The stream referenced by the fd parameter is linked below a multiplexer.</td>
</tr>
<tr>
<td>ENOSR</td>
<td>Buffers could not be allocated for the message that was to be created due to insufficient STREAMS memory resources.</td>
</tr>
<tr>
<td>ENOSTR</td>
<td>A stream is not associated with the fd parameter.</td>
</tr>
<tr>
<td>ENXIO</td>
<td>A hangup condition was generated downstream for the specified stream.</td>
</tr>
<tr>
<td>EPIPE or EIO</td>
<td>The fd parameter refers to a STREAM-based pipe and the other end of the pipe is closed. A SIGPIPE signal is generated for the calling thread.</td>
</tr>
<tr>
<td>ERANGE</td>
<td>The size of the data part of the message does not fall within the range specified by the maximum and minimum packet sizes of the topmost STREAMS module. OR The control part of the message is larger than the maximum configured size of the control part of a message.</td>
</tr>
</tbody>
</table>

Chapter 3. Streams 307
<table>
<thead>
<tr>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>The data part of a message is larger than the maximum configured size of the data part of a message.</td>
</tr>
</tbody>
</table>

The `putmsg` system call also fails if a STREAMS error message was processed by the stream head before the call. The error returned is the value contained in the STREAMS error message.

**Files**

/lib/pse.exp  
Contains the STREAMS export symbols.

**Related Information**

The `getmsg` system call, `getpmsg` system call, `putpmsg` system call.

The `read` subroutine, `poll` subroutine, `write` subroutine.

[Related Information]

List of Streams Programming References and STREAMS Overview in AIX 5L Version 5.3 Communications Programming Concepts.

---

**putnext Utility**

**Purpose**

Passes a message to the next queue.

**Syntax**

```c
#define putnext( q, mp ) (((q)->q_next->q_qinfo->qi_putp)((q)-q_next, (mp)))
```

**Description**

The `putnext` utility calls the `put procedure` of the next queue in a stream and passes to the procedure a message pointer as an argument. The `putnext` utility is the typical means of passing messages to the next queue in a stream.

This utility is part of STREAMS Kernel Extensions.

**Parameters**

- `q` Specifies the calling queue.
- `mp` Specifies the message that is to be passed.

**Related Information**

[Related Information]

List of Streams Programming References and Understanding STREAMS Messages in AIX 5L Version 5.3 Communications Programming Concepts.

---

**putpmsg System Call**

**Purpose**

Sends a priority message on a stream.
Syntax

```c
#include <stropts.h>

int putpmsg (fd, ctlptr, dataptr, band, flags)
int fd;
struct strbuf *ctlptr;
struct strbuf *dataptr;
int band;
int flags;
```

Description

The `putpmsg` system call is identical to the `putmsg` system call except that it sends a priority message. All information except for flag settings are found in the description for the `putmsg` system call. The differences in the flag settings are noted in the error codes section.

This system call is part of STREAMS Kernel Extensions.

Parameters

- `fd` Specifies a file descriptor referencing an open stream.
- `ctlptr` Holds the control part of the message.
- `dataptr` Holds the data part of the message.
- `band` Indicates the priority band.
- `flags` Indicates the priority type of message to be sent. Acceptable values are:
  - `MSG_BAND` Sends a non-priority message.
  - `MSG_HIPRI` Sends a priority message.

Error Codes

The `putpmsg` system call is unsuccessful under the following conditions:

- The `flags` parameter is set to a value of 0.
- The `flags` parameter is set to `MSG_HIPRI` and the `band` parameter is set to a nonzero value.
- The `flags` parameter is set to `MSG_HIPRI` and no control part is specified.

Related Information

- The `poll` subroutine, `read` subroutine, `write` subroutine.
- The `getmsg` system call, `getpmsg` system call, `putmsg` system call.
- List of Streams Programming References and STREAMS Overview in AIX 5L Version 5.3 Communications Programming Concepts.

putq Utility

Purpose

Puts a message on a queue.
Syntax

```c
int putq(q, bp)
register queue_t * q;
register mblk_t * bp;
```

Description

The `putq` utility puts the message pointed to by the `bp` parameter on the message queue pointed to by the `q` parameter, and then enables that queue. The `putq` utility queues messages based on message-queuing priority.

The priority classes are:

<table>
<thead>
<tr>
<th>Class</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>type &gt;= QPCTL</td>
<td>High-priority</td>
</tr>
<tr>
<td>type &lt; QPCTL &amp;&amp; band &gt; 0</td>
<td>Priority band</td>
</tr>
<tr>
<td>type &lt; QPCTL &amp;&amp; band == 0</td>
<td>Normal</td>
</tr>
</tbody>
</table>

When a high-priority message is queued, the `putq` utility always enables the queue. For a priority-band message, the `putq` utility is allowed to enable the queue (the QNOENAB flag is not set). Otherwise, the QWANTR flag is set, indicating that the service procedure is ready to read the queue. When an ordinary message is queued, the `putq` utility enables the queue if the following condition is set and if enabling is not inhibited by the noenable utility: the module has just been pushed, or else no message was queued on the last getq call and no message has been queued since.

The `putq` utility looks only at the priority band in the first message block of a message. If a high-priority message is passed to the `putq` utility with a nonzero `b_band` field value, the `b_band` field is reset to 0 before the message is placed on the queue. If the message passed to the `putq` utility has a `b_band` field value greater than the number of `qband` structures associated with the queue, the `putq` utility tries to allocate a new `qband` structure for each band up to and including the band of the message.

The `putq` utility should be used in the put procedure for the same queue in which the message is queued. A module should not call the `putq` utility directly in order to pass messages to a neighboring module. Instead, the `putq` utility itself can be used as the value of the `qi_putp` field in the put procedure for either or both of the module `qinit` structures. Doing so effectively bypasses any put-procedure processing and uses only the module service procedures.

This utility is part of STREAMS Kernel Extensions.

**Note:** The service procedure must never put a priority message back on its own queue, as this would result in an infinite loop.

Parameters

- `q` Specifies the queue on which to place the message.
- `bp` Specifies the message to put on the queue.

Return Values

On successful completion, the `putq` utility returns a value of 1. Otherwise, it returns a value of 0.

Related Information

The `getq` utility.
qenable Utility

Purpose
Enables a queue.

Syntax
void qenable (q)
register queue_t * q;

Description
The qenable utility places the queue pointed to by the q parameter on the linked list of queues ready to be called by the STREAMS scheduler.

This utility is part of STREAMS Kernel Extensions.

Parameters

q Specifies the queue to be enabled.

Related Information
List of Streams Programming References Understanding STREAMS Messages in AIX 5L Version 5.3 Communications Programming Concepts.

qreply Utility

Purpose
Sends a message on a stream in the reverse direction.

Syntax
void qreply (q, bp)
register queue_t * q;
register mblk_t * bp;

Description
The qreply utility sends the message pointed to by the bp parameter either up or down the stream-in the reverse direction from the queue pointed to by the q parameter. The utility does this by locating the partner of the queue specified by the q parameter (see the OTHERQ utility), and then calling the put procedure of that queue’s neighbor (as in the putnext utility). The qreply utility is typically used to send back a response (M_IOCACK or M_IOCNAK message) to an M_IOCTL message.

This utility is part of STREAMS Kernel Extensions.

Parameters

q Specifies which queue to send the message up or down.
bp Specifies the message to send.
Related Information
The OTHERQ utility, putnext utility.

List of Streams Programming References and Understanding STREAMS Messages in AIX 5L Version 5.3 Communications Programming Concepts.

qsize Utility

Purpose
Finds the number of messages on a queue.

Syntax
```c
int qsize(qp)
    register queue_t * qp;
```

Description
The qsize utility returns the number of messages present in the queue specified by the qp parameter. If there are no messages on the queue, the qsize parameter returns a value of 0.

This utility is part of STREAMS Kernel Extensions.

Parameters

qp Specifies the queue on which to count the messages.

Related Information
List of Streams Programming References and Understanding STREAMS Messages in AIX 5L Version 5.3 Communications Programming Concepts.

RD Utility

Purpose
Gets the pointer to the read queue.

Syntax
```c
#define RD(q) ((q)-1)
```

Description
The RD utility accepts a write-queue pointer, specified by the q parameter, as an argument and returns a pointer to the read queue for the same module.

This utility is part of STREAMS Kernel Extensions.

Parameters

q Specifies the write queue.
rmvb Utility

Purpose
Removes a message block from a message.

Syntax
```c
mblk_t *
rmvb(mp, bp)
```
```
register mblk_t * mp;
register mblk_t * bp;
```

Description
The `rmvb` utility removes the message block pointed to by the `bp` parameter from the message pointed to by the `mp` parameter, and then restores the linkage of the message blocks remaining in the message. The `rmvb` utility does not free the removed message block, but returns a pointer to the head of the resulting message. If the message block specified by the `bp` parameter is not contained in the message specified by the `mp` parameter, the `rmvb` utility returns a -1. If there are no message blocks in the resulting message, the `rmvb` utility returns a null pointer.

This utility is part of STREAMS Kernel Extensions.

Parameters
- **bp**: Specifies the message block to be removed.
- **mp**: Specifies the message from which to remove the message block.

Related Information
List of Streams Programming References and Understanding STREAMS Messages in AIX 5L Version 5.3 Communications Programming Concepts.

rmvq Utility

Purpose
Removes a message from a queue.

Syntax
```c
void rmvq (q, mp)
```
```
register queue_t * q;
register mblk_t * mp;
```

Description
Attention: If the `mp` parameter does not point to a message that is present on the specified queue, a system panic could result.
The `rmvq` utility removes the message pointed to by the `mp` parameter from the message queue pointed to by the `q` parameter, and then restores the linkage of the messages remaining on the queue.

This utility is part of STREAMS Kernel Extensions.

**Parameters**

- `q` Specifies the queue from which to remove the message.
- `mp` Specifies the message to be removed.

**Related Information**

List of Streams Programming References, Understanding STREAMS Messages in AIX 5L Version 5.3 Communications Programming Concepts.

---

**sad Device Driver**

**Purpose**

Provides an interface for administrative operations.

**Syntax**

```c
#include <sys/types.h>
#include <sys/conf.h>
#include <sys/sad.h>
#include <sys/stropts.h>

int ioctl (fildes, command, arg)
int fildes, command;
int arg;
```

**Description**

The STREAMS Administrative Driver (sad) provides an interface for applications to perform administrative operations on STREAMS modules and drivers. The interface is provided through `ioctl` operations. Privileged operation can access the `sad` device driver in the `/dev/sad/user` directory.

**Parameters**

- `fildes` Specifies an open file descriptor that refers to the `sad` device driver.
- `command` Determines the control function to be performed.
- `arg` Supplies additional information for the given control function.

**Values for the command Parameter**

The `autopush` command allows a user to configure a list of modules to be automatically pushed on a stream when a driver is first opened. The `autopush` command is controlled by the following commands.
Command | Description
--- | ---
SAD_SAP | Allows the person performing administrative duties to configure the information for the given device, which is used by the autopush command. The arg parameter points to a strapush structure containing the following elements:

```c
uint sap_cmd;
long sap_major;
long sap_minor;
long sap_lastminor;
long sap_npush;
uint sap_list[MAXAPUSH] [FMNAMESZ + 1];
```

The elements are described as follows:

- **sap_cmd**
  - Indicates the type of configuration being done. Acceptable values are:
    - **SAP_ONE**
      - Configures one minor device of a driver.
    - **SAP_RANGE**
      - Configures a range of minor devices of a driver.
    - **SAP_ALL**
      - Configures all minor devices of a driver.
    - **SAP_CLEAR**
      - Undoes configuration information for a driver.

- **sap_major**
  - Specifies the major device number of the device to be configured.

- **sap_minor**
  - Specifies the minor device number of the device to be configured.

- **sap_lastminor**
  - Specifies the last minor device number in a range of devices to be configured. This field is used only with the **SAP_RANGE** value in the sap_cmd field.

- **sap_npush**
  - Indicates the number of modules to be automatically pushed when the device is opened. The value of this field must be less than or equal to MAXAPUSH, which is defined in the sad.h file. It must also be less than or equal to NSTRPUSH, which is defined in the kernel master file.

- **sap_list**
  - Specifies an array of module names to be pushed in the order in which they appear in the list.

When using the **SAP_CLEAR** value, the user sets only the **sap_major** and **sap_minor** fields. This undoes the configuration information for any of the other values. If a previous entry was configured with the **SAP_ALL** value, the **sap_minor** field is set to 0. If a previous entry was configured with the **SAP_RANGE** value, the **sap_minor** field is set to the lowest minor device number in the range configured.

On successful completion, the return value from the ioctl operation is 0. Otherwise, the return value is -1.

SAD_GAP | Allows any user to query the sad device driver to get the autopush configuration information for a given device. The arg parameter points to a strapush structure as described under the SAD_SAP value.

The user sets the **sap_major** and **sap_minor** fields to the major and minor device numbers, respectively, of the device in question. On return, the strapush structure is filled with the entire information used to configure the device. Unused entries are filled with zeros.

On successful completion, the return value from the ioctl operation is 0. Otherwise, the return value is -1.
Command  Description
SAD_VML  Allows any user to validate a list of modules; that is, to see if they are installed on the system. The arg parameter is a pointer to a *str_list* structure containing the following elements:

```c
int sl_nmods;
struct str_mlist *sl_modlist;
```

The *str_mlist* structure contains the following element:

```c
char l_name[FMNAMESZ+1];
```

The fields are defined as follows:

- `sl_nmods` Indicates the number of entries the user has allocated in the array.
- `sl_modlist` Points to the array of module names.

**Return Values**

On successful completion, the return value from the *ioctl* operation is 0 if the list is valid or 1 if the list contains an invalid module name. Otherwise the return value is -1.

**Error Codes**

On failure, the *errno* global variable is set to one of the following values:

<table>
<thead>
<tr>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EFAULT</td>
<td>The arg parameter points outside the allocated address space.</td>
</tr>
<tr>
<td>EINVAL</td>
<td>The major device number is not valid, the number of modules is not valid.</td>
</tr>
<tr>
<td></td>
<td>OR</td>
</tr>
<tr>
<td></td>
<td>The list of module names is not valid.</td>
</tr>
<tr>
<td>ENOSTR</td>
<td>The major device number does not represent a STREAMS driver.</td>
</tr>
<tr>
<td>EEXIST</td>
<td>The major-minor device pair is already configured.</td>
</tr>
<tr>
<td>ERANGE</td>
<td>The value of the <em>command</em> parameter is SAP_RANGE and the value in the sap_lastminor field is not greater than the value in the sap_minor field.</td>
</tr>
<tr>
<td></td>
<td>OR</td>
</tr>
<tr>
<td></td>
<td>The value of the <em>command</em> parameter is SAP_CLEAR and the value in the sap_minor field is not equal to the first minor in the range.</td>
</tr>
<tr>
<td>ENODEV</td>
<td>The value in the <em>command</em> parameter is SAP_CLEAR and the device is not configured for the autopush command.</td>
</tr>
<tr>
<td>ENOSR</td>
<td>An internal autopush data structure cannot be allocated.</td>
</tr>
</tbody>
</table>

**Related Information**

The *autopush* command.

The *close* subroutine, *fstat* subroutine, *open* subroutine, *stat* subroutine.

Understanding streamio (STREAMS ioctl) Operations  Understanding STREMS Drivers and Modules  Understanding the log Device Driver in AIX 5L Version 5.3 Communications Programming Concepts.
**splstr Utility**

**Purpose**
Sets the processor level.

**Syntax**
```c
int splstr()
```

**Description**
The `splstr` utility increases the system processor level in order to block interrupts at a level appropriate for STREAMS modules and drivers when they are executing critical portions of their code. The `splstr` utility returns the processor level at the time of its invocation. Module developers are expected to use the standard `splx(s)` utility, where `s` is the integer value returned by the `splstr` operation, to restore the processor level to its previous value after the critical portions of code are passed.

This utility is part of STREAMS Kernel Extensions.

**Related Information**
The [splx utility](#).

[Link to List of Streams Programming References](#) and [STREAMS Overview](#) in *AIX 5L Version 5.3 Communications Programming Concepts*.

**splx Utility**

**Purpose**
Terminates a section of code.

**Syntax**
```c
int splx(x)
int x;
```

**Description**
The `splx` utility terminates a section of protected critical code. This utility restores the interrupt level to the previous level specified by the `x` parameter.

This utility is part of STREAMS Kernel Extensions.

**Related Information**
The [splstr utility](#).

[Link to List of Streams Programming References](#) and [Understanding STREAMS Drivers and Modules](#) in *AIX 5L Version 5.3 Communications Programming Concepts*.

**srv Utility**

**Purpose**
Services queued messages for STREAMS modules or drivers.
Syntax

```c
#include <sys/types.h>
#include <sys/stream.h>
#include <sys/stropts.h>

int <prefix>rsrv(queue_t *q);  /* read side */
int <prefix>wsrv(queue_t *q);  /* write side */
```

Parameters

- **q**: Pointer to the queue structure.

Description

The optional service (\texttt{<prefix>srv}) routine can be included in a STREAMS module or driver for one or more of the following reasons:
- To provide greater control over the flow of messages in a stream
- To make it possible to defer the processing of some messages to avoid depleting system resources
- To combine small messages into larger ones, or break large messages into smaller ones
- To recover from resource allocation failure. A module's or driver's \texttt{put} routine can test for the availability of a resource, and if it is not available, enqueue the message for later processing by the \texttt{srv} routine.

A message is first passed to a module’s or driver’s \texttt{put} routine, which may or may not do some processing. It must then either:
- Pass the message to the next stream component with \texttt{putnext}
- If a \texttt{srv} routine has been included, it may call \texttt{putq} to place the message on the queue

Once a message has been enqueued, the STREAMS scheduler controls the invocation of the service routine. Service routines are called in FIFO order by the scheduler. No guarantees can be made about how long it will take for a \texttt{srv} routine to be called except that it will happen before any user level process is run.

Every stream component (stream head, module or driver) has limit values it uses to implement flow control. Tunable high and low water marks should be checked to stop and restart the flow of message processing. Flow control limits apply only between two adjacent components with \texttt{srv} routines.

STREAMS messages can be defined to have up to 256 different priorities to support requirements for multiple bands of data flow. At a minimum, a stream must distinguish between normal (priority zero) messages and high priority messages (such as \texttt{M_IOCACK}). High priority messages are always placed at the head of the \texttt{srv} routine’s queue, after any other enqueued high priority messages. Next are messages from all included priority bands, which are enqueued in decreasing order of priority. Each priority band has its own flow control limits. If a flow controlled band is stopped, all lower priority bands are also stopped.

Once the STREAMS scheduler calls a \texttt{srv} routine, it must process all messages on its queue. The following steps are general guidelines for processing messages. Keep in mind that many of the details of how a \texttt{srv} routine should be written depend on the implementation, the direction of flow (upstream or downstream), and whether it is for a module or a driver.
1. Use \texttt{getq} to get the next enqueued message.
2. If the message is high priority, process it (if appropriate) and pass it to the next stream component with \texttt{putnext}
3. If it is not a high priority message (and therefore subject to flow control), attempt to send it to the next stream component with a \texttt{srv} routine. Use \texttt{canput} or \texttt{bcanput} to determine if this can be done.
4. If the message cannot be passed, put it back on the queue with \texttt{putbq}. If it can be passed, process it (if appropriate) and pass it with \texttt{putnext}

Rules for service routines:
1. Service routines must not call any kernel services that sleep or are not interrupt safe.
2. Service routines are called by the STREAMS scheduler with most interrupts enabled.

\textbf{Note:} Each stream module must specify a read and a write service (\texttt{srv}) routine. If a service routine is not needed (because the \texttt{put} routine processes all messages), a NULL pointer should be placed in module’s \texttt{qinit} structure. Do not use \texttt{nuldev} instead of the NULL pointer. Use of \texttt{nuldev} for a \texttt{srv} routine may result in flow control errors.

Prior to AIX 4.1, STREAMS service routines were permitted which were not coded to specification (that is, the service routine called sleep or called kernel services that slept, other possibilities). In AIX 4.1, this behavior will cause a system failure because the STREAMS scheduler is executed with some interrupts disabled. Modules or drivers can force the old style scheduling by setting the \texttt{sc_flags} field of the \texttt{kstrconf_t} structure to STR\_Q\_NOTTOSPEC. This structure is passed to the system when the module or driver calls the \texttt{str_install} STREAMS service. This flag will cause STREAMS to schedule the module’s or driver’s service routines with all interrupts enabled. There is a severe performance penalty for this type of STREAMS scheduling and future releases may not support STR\_Q\_NOTTOSPEC.

\textbf{Return Values}  
Ignored.

\textbf{Related Information}  
\texttt{put}, \texttt{bcanput} \texttt{canput}, \texttt{getq}, \texttt{putbq}, \texttt{putnext}, \texttt{putq} utilities.

The \texttt{queue} structure in /\texttt{usr/include/sys/stream.h}.

---

\textbf{str\_install Utility}  

\textbf{Purpose}  
Installs streams modules and drivers.

\textbf{Syntax}  
\texttt{#include <sys/strconf.h>}

\begin{verbatim}
int str_install(cmd, conf)
int cmd;
strconf_t *conf;
\end{verbatim}

\textbf{Description}  
The \texttt{str\_install} utility adds or removes Portable Streams Environment (PSE) drivers and modules from the internal tables of PSE. The extension is pinned when added and unpinned when removed (see the \texttt{pincode} kernel service). It uses a configuration structure to provide sufficient information to perform the specified command.

This utility is part of STREAMS Kernel Extensions.

The configuration structure, \texttt{strconf\_t}, is defined as follows:
typedef struct {
    char *sc_name;
    struct streamtab *sc_str;
    int sc_open_stylesc_flags;
    int sc_major;
    int sc_sqlevel;
    caddr_t sc_sqinfo;
} strconf_t;

The elements of the strconf_t structure are defined as follows:

<table>
<thead>
<tr>
<th>Element</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>sc_name</td>
<td>Specifies the name of the extension in the internal tables of PSE. For modules, this name is installed in the fmodsw table and is used for I_PUSH operations. For drivers, this name is used only for reporting with the scis and strinfo commands.</td>
</tr>
<tr>
<td>sc_str</td>
<td>Points to a streamtab structure.</td>
</tr>
</tbody>
</table>
### sc_open_style

**sc_flags**

Specifies the style of the driver or module open routine. The acceptable values are:

<table>
<thead>
<tr>
<th>Element</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>STR_NEW_OPEN</td>
<td>Specifies the open syntax and semantics used in System V Release 4.</td>
</tr>
<tr>
<td>STR_OLD_OPEN</td>
<td>Specifies the open syntax and semantics used in System V Release 3.</td>
</tr>
</tbody>
</table>

If the module is multiprocessor-safe, the following flag should be added by using the bitwise OR operator:

| STR_MPSAFE    | Specifies that the extension was designed to run on a multiprocessor system. |

If the module uses callback functions that need to be protected against interrupts (non-interrupt-safe callback functions) for the `timeout` or `bufcall` utilities, the following flag should be added by using the bitwise OR operator:

| STR_QSAFETY   | Specifies that the extension uses non-interrupt-safe callback functions for the `timeout` or `bufcall` utilities. |

This flag is automatically set by STREAMS if the module is not multiprocessor-safe.

| STR_PERSTREAM | Specifies that the module accepts to run at perstream synchronization level. |

| STR_Q_NOTTOSPEC | Specifies that the extension is designed to run it's service routine under process context. |

By default STREAMS service routine runs under interrupt context (INTOFFL3). If Streams drivers or modules want to execute their service routine under process context (INTBASE), they need to set this flag.

| STR_64BIT     | Specifies that the extension is capable to support 64-bit data types. |

| STR_NEWCLONING | Specifies the driver open uses new-style cloning. Under this style, the driver open() is not checking for CLONEOPEN flag and returns new device number. |

### sc_major

Specifies the major number of the device.
### sc_sqlevel

<table>
<thead>
<tr>
<th>Description</th>
<th>Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>Reserved for future use. Specifies the synchronization level to be used by PSE. There are seven levels of synchronization:</td>
<td></td>
</tr>
<tr>
<td><strong>SQLVL_NOP</strong> No synchronization</td>
<td>Specifies that each queue can be accessed by more than one thread at the same time. The protection of internal data and of put and service routines against the timeout or buffcall utilities is done by the module or driver itself. This synchronization level should be used essentially for multiprocessor-efficient modules.</td>
</tr>
<tr>
<td><strong>SQLVL_QUEUE</strong> Queue Level</td>
<td>Specifies that each queue can be accessed by only one thread at the same time. This is the finest synchronization level, and should only be used when the two sides of a queue pair do not share common data.</td>
</tr>
<tr>
<td><strong>SQLVL_QUEUEPAIR</strong> Queue Pair Level</td>
<td>Specifies that each queue pair can be accessed by only one thread at the same time.</td>
</tr>
<tr>
<td><strong>SQLVL_MODULE</strong> Module Level</td>
<td>Specifies that all instances of a module can be accessed by only one thread at the same time. This is the default value.</td>
</tr>
<tr>
<td><strong>SQLVL_ELSEWHERE</strong> Arbitrary Level</td>
<td>Specifies that a group of modules can be accessed by only one thread at the same time. Usually, the group of modules is a set of cooperating modules, such as a protocol family. The group is defined by using the same name in the sc_sqinfo field for each module in the group.</td>
</tr>
<tr>
<td><strong>SQLVL_GLOBAL</strong> Global Level</td>
<td>Specifies that all of PSE can be accessed by only one thread at the same time. This option should normally be used only for debugging.</td>
</tr>
<tr>
<td><strong>SQLVL_DEFAULT</strong> Default Level</td>
<td>Specifies the default level, set to SQLVL_MODULE.</td>
</tr>
</tbody>
</table>

### sc_sqinfo

Specifies an optional group name. This field is only used when the **SQLVL_ELSEWHERE** arbitrary synchronization level is set; all modules having the same name belong to one group. The name size is limited to eight characters.

### Parameters

- **cmd** Specifies which operation to perform. Acceptable values are:
  - **STR_LOAD_DEV** Adds a device into PSE internal tables.
  - **STR_UNLOAD_DEV** Removes a device from PSE internal tables.
  - **STR_LOAD_MOD** Adds a module into PSE internal tables.
  - **STR_UNLOAD_MOD** Removes a module from PSE internal tables.

- **conf** Points to a strconf_t structure, which contains all the necessary information to successfully load and unload a PSE kernel extension.
Return Values
On successful completion, the str_install utility returns a value of 0. Otherwise, it returns an error code.

Error Codes
On failure, the str_install utility returns one of the following error codes:

<table>
<thead>
<tr>
<th>Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EBUSY</td>
<td>The PSE kernel extension is already in use and cannot be unloaded.</td>
</tr>
<tr>
<td>EEXIST</td>
<td>The PSE kernel extension already exists in the system.</td>
</tr>
<tr>
<td>EINVAL</td>
<td>A parameter contains an unacceptable value.</td>
</tr>
<tr>
<td>ENODEV</td>
<td>The PSE kernel extension could not be loaded.</td>
</tr>
<tr>
<td>ENOENT</td>
<td>The PSE kernel is not present and could not be unloaded.</td>
</tr>
<tr>
<td>ENOMEM</td>
<td>Not enough memory for the extension could be allocated and pinned.</td>
</tr>
<tr>
<td>ENXIO</td>
<td>PSE is currently locked for use.</td>
</tr>
</tbody>
</table>

Related Information
The pincode kernel service, unpincode kernel service.

The streamio operations.

Configure_drivers_and_modules_in_the_portable_streams_environment_(PSE) and List_of_streams_programming REFERENCES in AIX 5L Version 5.3 Communications Programming Concepts.

streamio Operations

Purpose
Perform a variety of control functions on streams.

Syntax
```c
#include <stropts.h>
int ioctl (fildes, command, arg)
int fildes, command;
```

Description
See individual streamio operations for a description of each one.

This operation is part of STREAMS Kernel Extensions.

Parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>fildes</td>
<td>Specifies an open file descriptor that refers to a stream.</td>
</tr>
<tr>
<td>command</td>
<td>Determines the control function to be performed.</td>
</tr>
<tr>
<td>arg</td>
<td>Represents additional information that is needed by this operation.</td>
</tr>
</tbody>
</table>

The type of the arg parameter depends upon the operation, but it is generally an integer or a pointer to a command-specific data structure.

The command and arg parameters are passed to the file designated by the fildes parameter and are interpreted by the stream head. Certain combinations of these arguments can be passed to a module or driver in the stream.
## Values of the command Parameter

The following ioctl operations are applicable to all STREAMS files:

<table>
<thead>
<tr>
<th>Operation</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>I_ATMARK</td>
<td>Checks if the current message on the stream-head read queue is marked. Checks if a given band is writable. Checks if a message of a particular band is on the stream-head queue.</td>
</tr>
<tr>
<td>I_CANPUT</td>
<td>Creates a message from user specified buffers, adds information about another stream and sends the message downstream. Compares the names of all modules currently present in the stream to a specified name.</td>
</tr>
<tr>
<td>I_CKBAND</td>
<td>Flushes all input or output queues.</td>
</tr>
<tr>
<td>I_FDINSERT</td>
<td>Flushes all message of a particular band.</td>
</tr>
<tr>
<td>I_FIND</td>
<td>Gets the band of the first message on the stream-head read queue.</td>
</tr>
<tr>
<td>I_FLUSH</td>
<td>Returns the delay time.</td>
</tr>
<tr>
<td>I_FLUSHBAND</td>
<td>Returns the events for which the calling process is currently registered to be sent a SIGPOLL signal.</td>
</tr>
<tr>
<td>I_GETBAND</td>
<td>Returns the current read mode setting. Connects two specified streams. Lists all the module names on the stream.</td>
</tr>
<tr>
<td>I_GETCLTIME</td>
<td>Retrieves the name of the module just below the stream head.</td>
</tr>
<tr>
<td>I_GETSIG</td>
<td>Counts the number of data bytes in data blocks in the first message on the stream-head read queue, and places this value in a specified location. Allows a user to retrieve the information in the first message on the stream-head read queue without taking the message off the queue. Connects two specified streams.</td>
</tr>
<tr>
<td>I_GRDOPT</td>
<td>Removes the module just below the stream head.</td>
</tr>
<tr>
<td>I_LINK</td>
<td>Disconnects the two specified streams.</td>
</tr>
<tr>
<td>I_LIST</td>
<td>Pushes a module onto the top of the current stream. Retrieves the file descriptor associated with the message sent by an I_SENDFD operation over a stream pipe.</td>
</tr>
<tr>
<td>I_LOOK</td>
<td>Requests a stream to send a message to the stream head at the other end of a stream pipe.</td>
</tr>
<tr>
<td>I_NREAD</td>
<td>Sets the time that the stream head delays when a stream is closing.</td>
</tr>
<tr>
<td>I_PEEK</td>
<td>Informs the stream head that the user wishes the kernel to issue the SIGPOLL signal when a particular event occurs on the stream. Sets the read mode. Constructs an internal STREAMS ioctl message.</td>
</tr>
<tr>
<td>I_PLINK</td>
<td></td>
</tr>
<tr>
<td>I_POP</td>
<td></td>
</tr>
<tr>
<td>I_PUNLINK</td>
<td></td>
</tr>
<tr>
<td>I_PUSH</td>
<td></td>
</tr>
<tr>
<td>I_RECVFD</td>
<td></td>
</tr>
<tr>
<td>I_SENDFD</td>
<td></td>
</tr>
<tr>
<td>I_SETCLTIME</td>
<td></td>
</tr>
<tr>
<td>I_SETSIG</td>
<td></td>
</tr>
<tr>
<td>I_SRDOPT</td>
<td></td>
</tr>
<tr>
<td>I_STR</td>
<td></td>
</tr>
</tbody>
</table>
## I_UNLINK

**Description**

Disconnects the two specified streams.

## Return Values

Unless specified otherwise, the return value from the `ioctl` subroutine is 0 upon success and -1 if unsuccessful with the `errno` global variable set as indicated.

## Related Information

List of Streams Programming References | Understanding streamio (STREAMS ioctl) Operations in AIX 5L Version 5.3 Communications Programming Concepts.

## strlog Utility

### Purpose

Generates STREAMS error-logging and event-tracing messages.

### Syntax

```c
int strlog(mid, sid, level, flags, fmt, arg1, ...)
short mid, sid;
char level;
ushort flags;
char *fmt;
unsigned arg1;
```

### Description

The `strlog` utility generates log messages within the kernel. Required definitions are contained in the `sys/strlog.h` file.

This utility is part of STREAMS Kernel Extensions.

### Parameters

- **mid** Specifies the STREAMS module ID number for the module or driver submitting the log message.
- **sid** Specifies an internal sub-ID number usually used to identify a particular minor device of a driver.
- **level** Specifies a tracing level that allows for selective screening of low-priority messages from the tracer.
flags Specifies the destination of the message. This can be any combination of:

**SL_ERROR**
The message is for the error logger.

**SL_TRACE**
The message is for the tracer.

**SL_CONSOLE**
Log the message to the console.

**SL_FATAL**
Advisory notification of a fatal error.

**SL_WARN**
Advisory notification of a nonfatal error.

**SL_NOTE**
Advisory message.

**SL_NOTIFY**
Request that a copy of the message be mailed to the system administrator.

fmt Specifies a print style-format string, except that %f, %e, %E, %g, and %G conversion specifications are not handled.

arg1 Specifies numeric or character arguments. Up to NLOGARGS (currently 4) numeric or character arguments can be provided. (The NLOGARGS variable specifies the maximum number of arguments allowed. It is defined in the sys/strlog.h file.)

Related Information
The streamid operations.

**clone Device Driver** in AIX 5L Version 5.3 Communications Programming Concepts.

List of Streams Programming References | Understanding the log Device Driver | Understanding STREEMS Error and Trace Logging in AIX 5L Version 5.3 Communications Programming Concepts.

strqget Utility

Purpose
Obtains information about a queue or band of the queue.

Syntax
```c
int strqget(q, what, pri, valp)
register queue_t * q;
qfields_t what;
register unsigned char pri;
long * valp;
```

Description
The strqget utility allows modules and drivers to get information about a queue or particular band of the queue. The information is returned in the valp parameter. The fields that can be obtained are defined as follows:

This utility is part of STREEMS Kernel Extensions.
typedef enum qfields {
    QHIWAT = 0,
    QLOWAT = 1,
    QMAXPSZ = 2,
    QMINPSZ = 3,
    QCOUNT = 4,
    QFIRST = 5,
    QLAST = 6,
    QFLAG = 7,
    QBAD = 8
} qfields_t;

Parameters

q Specifies the queue about which to get information.
what Specifies the information to get from the queue.
pri Specifies the priority band about which to get information.
valp Contains the requested information on return.

Return Values

On success, the `strqget` utility returns a value of 0. Otherwise, it returns an error number.

Related Information

[List of Streams Programming References][1] Understanding STREAMS Messages in AIX 5L Version 5.3 Communications Programming Concepts.

---

### t_accept Subroutine for Transport Layer Interface

**Purpose**

Accepts a connect request.

**Library**

Transport Layer Interface Library (`libtli.a`)

**Syntax**

```c
#include <tuser.h>

int t_accept (fd, resfd, call)

int fd;
int resfd;
struct t_call * call;
```

**Description**

The `t_accept` subroutine is issued by a transport user to accept a connect request. A transport user can accept a connection on either the same local transport end point or on an end point different from the one on which the connect indication arrived.

**Parameters**

`fd` Identifies the local transport end point where the connect indication arrived.
`resfd` Specifies the local transport end point where the connection is to be established.

Contains information required by the transport provider to complete the connection. The call parameter points to a t_call structure, which contains the following members:

```c
struct netbuf addr;
struct netbuf opt;
struct netbuf udata;
int sequence;
```

The netbuf structure is described in the tiuser.h file. In the call parameter, the addr field is the address of the caller, the opt field indicates any protocol-specific parameters associated with the connection, the udata field points to any user data to be returned to the caller, and the sequence field is the value returned by the t_listen subroutine which uniquely associates the response with a previously received connect indication.

If the same end point is specified (that is, the resfd value equals the fd value), the connection can be accepted unless the following condition is true: the user has received other indications on that end point, but has not responded to them (with either the t_accept or t_snddis subroutine). For this condition, the t_accept subroutine fails and sets the t_errno variable to TBAFD.

If a different transport end point is specified (that is, the resfd value does not equal the fd value), the end point must be bound to a protocol address and must be in the T_IDLE state (see the t_getstate subroutine) before the t_accept subroutine is issued.

For both types of end points, the t_accept subroutine fails and sets the t_errno variable to TLOOK if there are indications (for example, a connect or disconnect) waiting to be received on that end point.

The values of parameters specified by the opt field and the syntax of those values are protocol-specific. The udata field enables the called transport user to send user data to the caller, the amount of user data must not exceed the limits supported by the transport provider as returned by the t_open or t_getinfo subroutine. If the value in the len field of the udata field is 0, no data will be sent to the caller.

Return Values
On successful completion, the t_connect subroutine returns a value of 0. Otherwise, it returns a value of -1, and the t_errno variable is set to indicate the error.

Error Codes
If unsuccessful, the t_errno variable is set to one of the following:

<table>
<thead>
<tr>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>TACCESS</td>
<td>The user does not have permission to accept a connection on the responding transport end point or use the specified options.</td>
</tr>
<tr>
<td>TBADDATA</td>
<td>The amount of user data specified was not within the bounds allowed by the transport provider.</td>
</tr>
<tr>
<td>TBADDR</td>
<td>The specified protocol address was in an incorrect format or contained illegal information.</td>
</tr>
<tr>
<td>TBADF</td>
<td>The specified file descriptor does not refer to a transport end point; or the user is illegally accepting a connection on the same transport end point on which the connect indication arrived.</td>
</tr>
<tr>
<td>TBADOPT</td>
<td>The specified options were in an incorrect format or contained illegal information.</td>
</tr>
<tr>
<td>TBADSEQ</td>
<td>An incorrect sequence number was specified.</td>
</tr>
<tr>
<td>TLOOK</td>
<td>An asynchronous event has occurred on the transport end point referenced by the fd parameter and requires immediate attention.</td>
</tr>
<tr>
<td>TNOTSUPPORT</td>
<td>This function is not supported by the underlying transport provider.</td>
</tr>
<tr>
<td>TOUTSTATE</td>
<td>The function was issued in the wrong sequence on the transport end point referenced by the fd parameter, or the transport end point referred to by the resfd parameter is not in the T_IDLE state.</td>
</tr>
<tr>
<td>TSYSERR</td>
<td>A system error has occurred during execution of this function.</td>
</tr>
</tbody>
</table>
Related Information
The `t_alloc` subroutine, `t_connect` subroutine, `t_getinfo` subroutine, `t_getstate` subroutine, `t_listen` subroutine, `t_open` subroutine, `t_rcvconnect` subroutine and `t_snddis` subroutine.

List of Streams Programming References and STREAMS Overview in AIX 5L Version 5.3 Communications Programming Concepts.

---

t_alloc Subroutine for Transport Layer Interface

**Purpose**
Allocates a library structure.

**Library**
Transport Layer Interface Library (`libtli.a`)

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

char *t_alloc (fd, struct_type, fields)
int fd;
int struct_type;
int fields;
```

**Description**
The `t_alloc` subroutine dynamically assigns memory for the various transport-function argument structures. This subroutine allocates memory for the specified structure, and also allocates memory for buffers referenced by the structure.

Use of the `t_alloc` subroutine to allocate structures will help ensure the compatibility of user programs with future releases of the transport interface.

**Parameters**
- `fd` Specifies the transport end point through which the newly allocated structure will be passed.
struct_type

Specifies the structure to be allocated. The structure to allocate is specified by the struct_type parameter, and can be one of the following:

T_BIND
    struct t_bind
T_CALL
    struct t_call
T_OPTMGMT
    struct t_optmgmt
T_DIS
    struct t_discon
T_UNITDATA
    struct t_unitdata
T_UDERROR
    struct t_uderr
T_INFO
    struct t_info

Each of these structures may subsequently be used as a parameter to one or more transport functions.

Each of the above structures, except T_INFO, contains at least one field of the struct netbuf type. The netbuf structure is described in the tiuser.h file. For each field of this type, the user may specify that the buffer for that field should be allocated as well. The fields parameter specifies this option, where the parameter is the bitwise-OR of any of the following:

T_ADDR
    The addr field of the t_bind, t_call, t_unitdata, or t_uderr structure.
T_OPT
    The opt field of the t_optmgmt, t_call, t_unitdata, or t_uderr structure.
T_UDATA
    The udata field of the t_call, t_discon, or t_unitdata structure.
T_ALL
    All relevant fields of the given structure.

fields

Specifies whether the buffer should be allocated for each field type. For each field specified in the fields parameter, the t_alloc subroutine allocates memory for the buffer associated with the field, initializes the len field to zero, and initializes the buf pointer and the maxlen field accordingly. The length of the buffer allocated is based on the same size information returned to the user from the t_open and t_getinfo subroutines. Thus, the fd parameter must refer to the transport end point through which the newly allocated structure will be passed, so that the appropriate size information can be accessed. If the size value associated with any specified field is -1 or -2, the t_alloc subroutine will be unable to determine the size of the buffer to allocate; it then fails, setting the t_errno variable to T_SYSERR and the errno global variable to EINVAL. For any field not specified in the fields parameter, the buf field is set to null and the maxlen field is set to 0.

Return Values

On successful completion, the t_alloc subroutine returns a pointer to the newly allocated structure. Otherwise, it returns a null pointer.

Error Codes

On failure, the t_errno variable is set to one of the following:

<table>
<thead>
<tr>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>TBADF</td>
<td>The specified file descriptor does not refer to a transport end point.</td>
</tr>
</tbody>
</table>
Value  Description
TNOSTRUCTYPE  Unsupported structure type requested. This can include a request for a structure type which is inconsistent with the transport provider type specified, for example, connection-oriented or connectionless.
TSYSERR  A system error has occurred during execution of this function.

Related Information
The [t_free][1] subroutine,  [t_getinfo][2] subroutine,  [t_open][3] subroutine.

The [List of Streams Programming References][4] and [STREAMS Overview][5] in AIX 5L Version 5.3 Communications Programming Concepts.

---

### t_bind Subroutine for Transport Layer Interface

#### Purpose
Binds an address to a transport end point.

#### Library
Transport Layer Interface Library (libtli.a)

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

int t_bind(fd, req, ret)
int fd;
struct t_bind * req;
struct t_bind * ret;
```

#### Description
The **t_bind** subroutine associates a protocol address with the transport end point specified by the **fd** parameter and activates that transport end point. In connection mode, the transport provider may begin accepting or requesting connections on the transport end point. In connectionless mode, the transport user may send or receive data units through the transport end point.

#### Parameters
- **fd** Specifies the transport end point.
- **req** Specifies the address to be bound to the given transport end point.
- **ret** Specifies the maximum size of the address buffer.

The **req** and **ret** parameters point to a **t_bind** structure containing the following members:
```
struct netbuf addr;
unsigned qlen;
```

The **netbuf** structure is described in the **tiuser.h** file. The **addr** field of the **t_bind** structure specifies a protocol address and the **qlen** field is used to indicate the maximum number of outstanding connect indications.

The **req** parameter is used to request that the address represented by the **netbuf** structure be bound to the given transport end point. In the **req** parameter, the **netbuf** structure fields have the following meanings:
Field | Description
--- | ---
len | Specifies the number of bytes in the address.
buf | Points to the address buffer.
maxlen | Has no meaning for the req parameter.

On return, the ret parameter contains the address that the transport provider actually bound to the transport end point; this may be different from the address specified by the user in the req parameter. In the ret parameter, the netbuf structure fields have the following meanings:

Field | Description
--- | ---
maxlen | Specifies the maximum size of the address buffer.
buf | Points to the buffer where the address is to be placed. (On return, this field points to the bound address.)
len | Specifies the number of bytes in the bound address.

If the value of the maxlen field is not large enough to hold the returned address, an error will result.

If the requested address is not available or if no address is specified in the req parameter (that is, the len field of the addr field in the req parameter is 0) the transport provider assigns an appropriate address to be bound and returns that address in the addr field of the ret parameter. The user can compare the addresses in the req parameter to those in the ret parameter to determine whether the transport provider has bound the transport end point to a different address than that requested. If the transport provider could not allocate an address, the t_bind subroutine fails and t_errno is set to TNOADDR.

The req parameter may be null if the user does not wish to specify an address to be bound. Here, the value of the qlen field is assumed to be 0, and the transport provider must assign an address to the transport end point. Similarly, the ret parameter may be null if the user does not care which address was bound by the provider and is not interested in the negotiated value of the qlen field. It is valid to set the req and ret parameters to null for the same call, in which case the provider chooses the address to bind to the transport end point and does not return that information to the user.

The qlen field has meaning only when initializing a connection-mode service. It specifies the number of outstanding connect indications the transport provider should support for the given transport end point. An outstanding connect indication is one that has been passed to the transport user by the transport provider. A value of the qlen field greater than 0 is only meaningful when issued by a passive transport user that expects other users to call it. The value of the qlen field is negotiated by the transport provider and can be changed if the transport provider cannot support the specified number of outstanding connect indications. On return, the qlen field in the ret parameter contains the negotiated value.

This subroutine allows more than one transport end point to be bound to the same protocol address as long as the transport provider also supports this capability. However, it is not allowable to bind more than one protocol address to the same transport end point. If a user binds more than one transport end point to the same protocol address, only one end point can be used to listen for connect indications associated with that protocol address. In other words, only one t_bind subroutine for a given protocol address may specify a value greater than 0 for the qlen field. In this way, the transport provider can identify which transport end point should be notified of an incoming connect indication. If a user attempts to bind a protocol address to a second transport end point having a qlen value greater than 0, the transport provider instead assigns another address to be bound to that end point. If a user accepts a connection on the transport end point that is being used as the listening end point, the bound protocol address is found to be busy for the duration of that connection. No other transport end points may be bound for listening while that initial listening end point is in the data-transfer phase. This prevents more than one transport end point bound to the same protocol address from accepting connect indications.
Return Values
On successful completion, the \texttt{t_connect} subroutine returns a value of 0. Otherwise, it returns a value of -1, and the \texttt{t_errno} variable is set to indicate the error.

Error Codes
If unsuccessful, the \texttt{t_errno} variable is set to one of the following:

<table>
<thead>
<tr>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>TACCES</td>
<td>The user does not have permission to use the specified address.</td>
</tr>
<tr>
<td>TADDRBUSY</td>
<td>The requested address is in use.</td>
</tr>
<tr>
<td>TBADADDR</td>
<td>The specified protocol address was in an incorrect format or contained illegal information.</td>
</tr>
<tr>
<td>TBADF</td>
<td>The specified file descriptor does not refer to a transport end point.</td>
</tr>
<tr>
<td>TBUFOVFLW</td>
<td>The number of bytes allowed for an incoming argument is not sufficient to store the value of that argument. The provider's state changes to \texttt{T_IDLE} and the information to be returned in the \texttt{ret} parameter is discarded.</td>
</tr>
<tr>
<td>TNOADDR</td>
<td>The transport provider could not allocate an address.</td>
</tr>
<tr>
<td>TOUTSTATE</td>
<td>The function was issued in the wrong sequence.</td>
</tr>
<tr>
<td>TSYSERR</td>
<td>A system error has occurred during execution of this function.</td>
</tr>
</tbody>
</table>

Related Information
The \texttt{t_open} subroutine, \texttt{t_optmgmt} subroutine, \texttt{t_unbind} subroutine.

List of Streams Programming References and STREAMS Overview in AIX 5L Version 5.3 Communications Programming Concepts.

\textbf{t_close Subroutine for Transport Layer Interface}

\textbf{Purpose}
Closes a transport end point.

\textbf{Library}
Transport Layer Interface Library (\texttt{libtli.a})

\textbf{Syntax}
```c
#include <tiuser.h>

int t_close(fd)
    int fd;
```

\textbf{Description}
The \texttt{t_close} subroutine informs the transport provider that the user is finished with the transport end point specified by the \texttt{fd} parameter and frees any local library resources associated with the end point. In addition, the \texttt{t_close} subroutine closes the file associated with the transport end point.

The \texttt{t_close} subroutine should be called from the \texttt{T_UNBND} state (see the \texttt{t_getstate} subroutine). However, this subroutine does not check state information, so it may be called from any state to close a transport end point. If this occurs, the local library resources associated with the end point are freed automatically. In addition, the \texttt{close} subroutine is issued for that file descriptor. The \texttt{close} subroutine is abortive if no other process has that file open, and will break any transport connection that may be associated with that end point.
Parameter

\(fd\) Specifies the transport end point to be closed.

Return Values
On successful completion, the \texttt{t_connect} subroutine returns a value of 0. Otherwise, it returns a value of -1, and the \texttt{t_errno} variable is set to indicate the error.

Error Code
If unsuccessful, the \texttt{t_errno} variable is set to the following:

<table>
<thead>
<tr>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>TBADF</td>
<td>The specified file descriptor does not refer to a transport end point.</td>
</tr>
</tbody>
</table>

Related Information
The \texttt{close} subroutine, \texttt{t_getstate} subroutine, \texttt{t_open} subroutine, \texttt{t_unbind} subroutine.

List of Streams Programming References and STREAMS Overview in AIX 5L Version 5.3 Communications Programming Concepts.

\textbf{t_connect Subroutine for Transport Layer Interface}

Purpose
Establishes a connection with another transport user.

Library
Transport Layer Interface Library (\texttt{libtli.a})

Syntax

\begin{verbatim}
#include <tiuser.h>

int t_connect(fd, sndcall, rcvcall)

int fd;
struct t_call *sndcall;
struct t_call *rcvcall;
\end{verbatim}

Description
The \texttt{t_connect} subroutine enables a transport user to request a connection to the specified destination transport user.

Parameters

\begin{itemize}
  \item \texttt{fd} Identifies the local transport end point where communication will be established.
  \item \texttt{sndcall} Specifies information needed by the transport provider to establish a connection.
  \item \texttt{rcvcall} Specifies information associated with the newly established connection.
\end{itemize}

The \texttt{sndcall} and \texttt{rcvcall} parameters point to a \texttt{t_call} structure that contains the following members:
struct netbuf addr;
struct netbuf opt;
struct netbuf udata;
int sequence;

The netbuf structure is described in the tiuser.h file. In the sndcall parameter, the addr field specifies the protocol address of the destination transport user, the opt field presents any protocol-specific information that might be needed by the transport provider, the udata field points to optional user data that may be passed to the destination transport user during connection establishment, and the sequence field has no meaning for this function.

On return to the rcvcall parameter, the addr field returns the protocol address associated with the responding transport end point, the opt field presents any protocol-specific information associated with the connection, the udata field points to optional user data that may be returned by the destination transport user during connection establishment; and the sequence field has no meaning for this function.

The opt field implies no structure on the options that may be passed to the transport provider. The transport provider is free to specify the structure of any options passed to it. These options are specific to the underlying protocol of the transport provider. The user can choose not to negotiate protocol options by setting the len field of the opt field to 0. In this case, the provider may use default options.

The udata field enables the caller to pass user data to the destination transport user and receive user data from the destination user during connection establishment. However, the amount of user data must not exceed the limits supported by the transport provider as returned by the t_open or t_getinfo subroutine. If the len field of the udata field in the sndcall parameter is 0, no data is sent to the destination transport user.

On return, the addr, opt, and udata fields of the rcvcall parameter are updated to reflect values associated with the connection. Thus, the maxlen field of each parameter must be set before issuing this function to indicate the maximum size of the buffer for each. However, the rcvcall parameter may be null, in which case no information is given to the user on return from the t_connect subroutine.

By default, the t_connect subroutine executes in synchronous mode, and waits for the destination user’s response before returning control to the local user. A successful return (that is, a return value of 0) indicates that the requested connection has been established. However, if the O_NDELAY flag is set (with the t_open subroutine or the fcntl command), the t_connect subroutine executes in asynchronous mode. In this case, the call does not wait for the remote user’s response, but returns control immediately to the local user and returns -1 with the t_errno variable set to TNODATA to indicate that the connection has not yet been established. In this way, the function simply initiates the connection establishment procedure by sending a connect request to the destination transport user.

Return Values
On successful completion, the t_connect subroutine returns a value of 0. Otherwise, it returns a value of -1, and the t_errno variable is set to indicate the error.

Error Codes
If unsuccessful, the t_errno variable is set to one of the following:

<table>
<thead>
<tr>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>TACCESS</td>
<td>The user does not have permission to use the specified address or options.</td>
</tr>
<tr>
<td>TBADADDR</td>
<td>The specified protocol address was in an incorrect format or contained illegal information.</td>
</tr>
<tr>
<td>TBADDATA</td>
<td>The amount of user data specified was not within the bounds allowed by the transport provider.</td>
</tr>
<tr>
<td>TBADF</td>
<td>The specified file descriptor does not refer to a transport end point.</td>
</tr>
<tr>
<td>TBADOPT</td>
<td>The specified protocol options were in an incorrect format or contained illegal information.</td>
</tr>
<tr>
<td>Value</td>
<td>Description</td>
</tr>
<tr>
<td>--------------</td>
<td>------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>TBUFOVFLW</td>
<td>The number of bytes allocated for an incoming argument is not sufficient to</td>
</tr>
<tr>
<td></td>
<td>store the value of that argument. If executed in synchronous mode, the</td>
</tr>
<tr>
<td></td>
<td>provider’s state, as seen by the user, changes to T_DATAxFER, and the</td>
</tr>
<tr>
<td></td>
<td>connect indication information to be returned in the rcvcall parameter is</td>
</tr>
<tr>
<td></td>
<td>discarded.</td>
</tr>
<tr>
<td>TLOOK</td>
<td>An asynchronous event has occurred on this transport end point and requires</td>
</tr>
<tr>
<td></td>
<td>immediate attention.</td>
</tr>
<tr>
<td>TNODATA</td>
<td>The O_NDELAY or O_NONBLOCK flag was set, so the function successfully</td>
</tr>
<tr>
<td></td>
<td>initiated the connection establishment procedure, but did not wait for a</td>
</tr>
<tr>
<td></td>
<td>response from the remote user.</td>
</tr>
<tr>
<td>TNOTSUPPORT</td>
<td>This function is not supported by the underlying transport provider.</td>
</tr>
<tr>
<td>TOUTSTATE</td>
<td>The function was issued in the wrong sequence.</td>
</tr>
<tr>
<td>TSYERR</td>
<td>A system error has occurred during execution of this function.</td>
</tr>
</tbody>
</table>

**Related Information**

The `fcntl` command.

The `t_accept` subroutine, `t_getinfo` subroutine, `t_listen` subroutine, `t_open` subroutine, `t_optmgmt` subroutine, `t_rcvconnect` subroutine.

List of Streams Programming References and STREAMS Overview in AIX 5L Version 5.3 Communications Programming Concepts.

---

### t_error Subroutine for Transport Layer Interface

**Purpose**

Produces an error message.

**Library**

Transport Layer Interface Library (libtli.a)

**Syntax**

```c
#include <tiuser.h>

void t_error(errmsg)
char *errmsg;
extern int t_errno;
extern char *t_errno;
extern int t_nerr;
```

**Description**

The `t_error` subroutine produces a message on the standard error output that describes the last error encountered during a call to a transport function.

The `t_error` subroutine prints the user-supplied error message, followed by a colon and the standard transport-function error message for the current value contained in the `t_errno` variable.

**Parameter**

`errmsg` Specifies a user-supplied error message that gives context to the error.
External Variables

t_errno
Specifies which standard transport-function error message to print. If the value of the t_errno variable is TSYSERR, the t_error subroutine also prints the standard error message for the current value contained in the errno global variable.

The t_errno variable is set when an error occurs and is not cleared on subsequent successful calls.

t_nerr
Specifies the maximum index value for the t_errlist array. The t_errlist array is the array of message strings allowing user-message formatting. The t_errno variable can be used as an index into this array to retrieve the error message string (without a terminating new-line character).

Examples

A t_connect subroutine is unsuccessful on transport end point fd2 because a bad address was given, and the following call follows the failure:

    t_error("t_connect failed on fd2")

The diagnostic message would print as:

    t_connect failed on fd2: Incorrect transport address format

In this example, t_connect failed on fd2 tells the user which function was unsuccessful on which transport end point, and Incorrect transport address format identifies the specific error that occurred.

Related Information

List of Streams Programming References [STREAMS Overview] in AIX 5L Version 5.3 Communications Programming Concepts.

---

t_free Subroutine for Transport Layer Interface

Purpose

Frees a library structure.

Library

Transport Layer Interface Library (libtli.a)

Syntax

```c
#include <tiuser.h>

int t_free(ptr, struct_type)
char *ptr;
int struct_type;
```

Description

The t_free subroutine frees memory previously allocated by the t_alloc subroutine. This subroutine frees memory for the specified structure and also frees memory for buffers referenced by the structure.

The t_free subroutine checks the addr, opt, and udata fields of the given structure (as appropriate) and frees the buffers pointed to by the buf field of the netbuf structure. If the buf field is null, the t_free subroutine does not attempt to free memory. After all buffers are freed, the t_free subroutine frees the memory associated with the structure pointed to by the ptr parameter.

Undefined results will occur if the ptr parameter or any of the buf pointers points to a block of memory that was not previously allocated by the t_alloc subroutine.
Parameters

ptr

Points to one of the seven structure types described for the t_alloc subroutine.

struct_type

Identifies the type of that structure. The type can be one of the following:

<table>
<thead>
<tr>
<th>Type</th>
<th>Structure</th>
</tr>
</thead>
<tbody>
<tr>
<td>T_BIND</td>
<td>struct t_bind</td>
</tr>
<tr>
<td>T_CALL</td>
<td>struct t_call</td>
</tr>
<tr>
<td>T_OPTMGMT</td>
<td>struct t_optmgmt</td>
</tr>
<tr>
<td>T_DIS</td>
<td>struct t_disc</td>
</tr>
<tr>
<td>T_UNITDATA</td>
<td>struct t_unitdata</td>
</tr>
<tr>
<td>T_UDERROR</td>
<td>struct t_uderr</td>
</tr>
<tr>
<td>T_INFO</td>
<td>struct t_info</td>
</tr>
</tbody>
</table>

Each of these structure types is used as a parameter to one or more transport subroutines.

Return Values

On successful completion, the t_free subroutine returns a value of 0. Otherwise, it returns a value of -1, and the t_errno variable is set to indicate the error.

Error Codes

If unsuccessful, the t_errno variable is set to the following:

<table>
<thead>
<tr>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>TNOSTRUCTYPE</td>
<td>Unsupported structure type requested.</td>
</tr>
<tr>
<td>TSYSERR</td>
<td>A system error has occurred during execution of this function.</td>
</tr>
</tbody>
</table>

Related Information

The t_alloc subroutine.

[List of Streams Programming References](#) and [STREAMS Overview](#) in AIX 5L Version 5.3 Communications Programming Concepts.

---

**t_getinfo Subroutine for Transport Layer Interface**

**Purpose**

Gets protocol-specific service information.

**Library**

Transport Layer Interface Library (libtli.a)

**Syntax**

```c
#include <tiuser.h>
```
int t_getinfo(fd, info)
int fd;
struct t_info *info;

Description
The t_getinfo subroutine returns the current characteristics of the underlying transport protocol associated with fd file descriptor. The t_info structure is used to return the same information returned by the t_open subroutine. This function enables a transport user to access this information during any phase of communication.

Parameters

fd Specifies the file descriptor.
info  Points to a `t_info` structure that contains the following members:

```c
long addr;
long options;
long tsdu;
long etsdu;
long connect;
long discon;
long servtype;
```

The values of the fields have the following meanings:

**addr**  A value greater than or equal to 0 indicates the maximum size of a transport protocol address; a value of -1 specifies that there is no limit on the address size; and a value of -2 specifies that the transport provider does not provide user access to transport protocol addresses.

**options**  A value greater than or equal to 0 indicates the maximum number of bytes of protocol-specific options supported by the provider; a value of -1 specifies that there is no limit on the option size; and a value of -2 specifies that the transport provider does not support user-settable options.

**tsdu**  A value greater than 0 specifies the maximum size of a transport service data unit (TSDU); a value of 0 specifies that the transport provider does not support the concept of TSDU, although it does support the sending of a data stream having no logical boundaries preserved across a connection; a value of -1 specifies that there is no limit on the size of a TSDU; and a value of -2 specifies that the transfer of normal data is not supported by the transport provider.

**etsdu**  A value greater than 0 specifies the maximum size of an expedited transport service data unit (ETSDU); a value of 0 specifies that the transport provider does not support the concept of ETSDU, although it does support the sending of an expedited data stream having no logical boundaries preserved across a connection; a value of -1 specifies that there is no limit on the size of an ETSDU; and a value of -2 specifies that the transfer of expedited data is not supported by the transport provider.

**connect**  A value greater than or equal to 0 specifies the maximum amount of data that may be associated with connection establishment functions; a value of -1 specifies that there is no limit on the amount of data sent during connection establishment; and a value of -2 specifies that the transport provider does not allow data to be sent with connection establishment functions.

**discon**  A value greater than or equal to 0 specifies the maximum amount of data that may be associated with the `t_snddis` and `t_rcvdis` subroutines; a value of -1 specifies that there is no limit on the amount of data sent with these abortive release functions; and a value of -2 specifies that the transport provider does not allow data to be sent with the abortive release functions.

**servtype**  This field specifies the service type supported by the transport provider.

If a transport user is concerned with protocol independence, the sizes may be accessed to determine how large the buffers must be to hold each piece of information. Alternatively, the `t_alloc` subroutine may be used to allocate these buffers. An error will result if a transport user exceeds the allowed data size on any function. The value of each field can change as a result of option negotiation; the `t_getinfo` subroutine enables a user to retrieve the current characteristics.

**Return Values**

On successful completion, the `t_getinfo` subroutine returns a value of 0. Otherwise, it returns a value of -1, and the `t_errno` variable is set to indicate the error.

The **servtype** field of the `info` parameter may specify one of the following values on return:

<table>
<thead>
<tr>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>T_COTS</td>
<td>The transport provider supports a connection-mode service, but does not support the optional orderly release facility.</td>
</tr>
</tbody>
</table>
Value     Description
T_COTS_ORD  The transport provider supports a connection-mode service with the optional orderly release facility.
T_CLTS     The transport provider supports a connectionless-mode service. For this service type, the \texttt{t\_open} subroutine returns -2 for the values in the \texttt{etsdu}, \texttt{connect}, and \texttt{discon} fields.

Error Codes
In unsuccessful, the \texttt{t\_errno} variable is set to one of the following:

<table>
<thead>
<tr>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>TBADF</td>
<td>The specified file descriptor does not refer to a transport end point.</td>
</tr>
<tr>
<td>TSYSSERR</td>
<td>A system error has occurred during execution of this function.</td>
</tr>
</tbody>
</table>

Related Information
The \texttt{t\_alloc} subroutine, \texttt{t\_open} subroutine, \texttt{t\_rcvdis} subroutine and \texttt{t\_snddis} subroutine.

\textbf{List of Streams Programming Reference} and \textbf{STREAMS Overview} in \textit{AIX 5L Version 5.3 Communications Programming Concepts}.

\textbf{t\_getstate Subroutine for Transport Layer Interface}

\textbf{Purpose}
Gets the current state.

\textbf{Library}
Transport Layer Interface Library (\texttt{libtli.a})

\textbf{Syntax}
\begin{verbatim}
#include <tiuser.h>

int t_getstate(fd)  
int \texttt{fd};
\end{verbatim}

\textbf{Description}
The \texttt{t\_getstate} subroutine returns the current state of the provider associated with the transport end point specified by the \texttt{fd} parameter.

\textbf{Parameter}
\begin{verbatim}
fd    Specifies the transport end point.
\end{verbatim}

\textbf{Return Codes}
On successful completion, the \texttt{t\_getstate} subroutine returns the current state. Otherwise, it returns a value of -1, and the \texttt{t\_errno} variable is set to indicate the error.

If the provider is undergoing a state transition when the \texttt{t\_getstate} subroutine is called, the function will fail. The current state is one of the following.
### Value | Description
--- | ---
T_DATAXFER | Data transfer.
T_IDLE | Idle.
T_INCON | Incoming connection pending.
T_INREL | Incoming orderly release (waiting to send an orderly release indication).
T_OUTCON | Outgoing connection pending.
T_OUTREL | Outgoing orderly release (waiting for an orderly release indication).
T_UNBND | Unbound.

### Error Codes
If unsuccessful, the t_errno variable is set to one of the following:

#### Value | Description
--- | ---
TBADF | The specified file descriptor does not refer to a transport end point.
TSTATECHNG | The transport provider is undergoing a state change.
TSYSERR | A system error has occurred during execution of this function.

### Related Information
- The [t_open](#) subroutine.
- [List of Streams Programming References](#) and [STREAMS Overview](#) in AIX 5L Version 5.3 Communications Programming Concepts.

### t_listen Subroutine for Transport Layer Interface

#### Purpose
Listens for a connect request.

#### Library
Transport Layer Interface Library (libtli.a)

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

int t_listen(fd, call)
int fd;
struct t_call *call;
```

#### Description
The `t_listen` subroutine listens for a connect request from a calling transport user.

**Note:** If a user issues a `t_listen` subroutine call in synchronous mode on a transport end point that was not bound for listening (that is, the qlen field was 0 on the `t_bind` subroutine), the call will never return because no connect indications will arrive on that endpoint.

#### Parameters
- `fd` Identifies the local transport endpoint where connect indications arrive.
call Contains information describing the connect indication.

The call parameter points to a t_call structure that contains the following members:

```c
struct netbuf addr;
struct netbuf opt;
struct netbuf udata;
int sequence;
```

The netbuf structure contains the following fields:

- **addr** Returns the protocol address of the calling transport user.
- **opt** Returns protocol-specific parameters associated with the connect request.
- **udata** Returns any user data sent by the caller on the connect request.
- **sequence** Uniquely identifies the returned connect indication. The value of sequence enables the user to listen for multiple connect indications before responding to any of them.

Since the t_listen subroutine returns values for the addr, opt, and udata fields of the call parameter, the maxlen field of each must be set before issuing the t_listen subroutine to indicate the maximum size of the buffer for each.

By default, the t_listen subroutine executes in synchronous mode and waits for a connect indication to arrive before returning to the user. However, if the O_NDELAY or O_NONBLOCK flag is set (using the t_open subroutine or the fcntl command), the t_listen subroutine executes asynchronously, reducing to a poll for existing connect indications. If none are available, the t_listen subroutine returns -1 and sets the t_errno variable to TNODATA.

**Return Values**

On successful completion, the t_listen subroutine returns a value of 0. Otherwise, it returns a value of -1, and the t_errno variable is set to indicate the error.

**Error Codes**

If unsuccessful, the t_errno variable is set to one of the following:

<table>
<thead>
<tr>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>TBADF</td>
<td>The specified file descriptor does not refer to a transport end point.</td>
</tr>
<tr>
<td>TBADQLEN</td>
<td>The transport end point is not bound for listening. The qlen is zero.</td>
</tr>
<tr>
<td>TBUFOVFLW</td>
<td>The number of bytes allocated for an incoming argument is not sufficient to store the value of that argument. The provider's state, as seen by the user, changes to T_INCON, and the connect-indication information to be returned in the call parameter is discarded.</td>
</tr>
<tr>
<td>TLOOK</td>
<td>An asynchronous event has occurred on this transport end point and requires immediate attention.</td>
</tr>
<tr>
<td>TNODATA</td>
<td>The O_NDELAY or O_NONBLOCK flag was set, but no connect indications had been queued.</td>
</tr>
<tr>
<td>TNOTSUPPORT</td>
<td>This function is not supported by the underlying transport provider.</td>
</tr>
<tr>
<td>TOUTSTATE</td>
<td>The subroutine was issued in the wrong sequence.</td>
</tr>
<tr>
<td>TSYSERR</td>
<td>A system error has occurred during execution of this function.</td>
</tr>
</tbody>
</table>

**Related Information**

The t_accept subroutine, t_alloc subroutine, t_bind subroutine, t_connect subroutine, t_open subroutine, t_rcvconnect subroutine.

List of Streams Programming References and STREAMS Overview in AIX 5L Version 5.3 Communications Programming Concepts.
t_look Subroutine for Transport Layer Interface

Purpose
Looks at the current event on a transport end point.

Library
Transport Layer Interface Library (libtli.a)

Syntax
#include <tiuser.h>

int t_look(fd)
int fd;

Description
The t_look subroutine returns the current event on the transport end point specified by the fd parameter. This subroutine enables a transport provider to notify a transport user of an asynchronous event when the user is issuing functions in synchronous mode. Certain events require immediate notification of the user and are indicated by a specific error, TLOOK, on the current or next subroutine executed.

This subroutine also enables a transport user to poll a transport end point periodically for asynchronous events.

Parameter
fd Specifies the transport end point.

Return Values
On successful completion, the t_look subroutine returns a value that indicates which of the allowable events has occurred, or returns a value of 0 if no event exists. One of the following events is returned:

<table>
<thead>
<tr>
<th>Event</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>T_CONNECT</td>
<td>Indicates connect confirmation received.</td>
</tr>
<tr>
<td>T_DATA</td>
<td>Indicates normal data received.</td>
</tr>
<tr>
<td>T_DISCONNECT</td>
<td>Indicates disconnect received.</td>
</tr>
<tr>
<td>T_ERROR</td>
<td>Indicates fatal error.</td>
</tr>
<tr>
<td>T_EXDATA</td>
<td>Indicates expedited data received.</td>
</tr>
<tr>
<td>T_LISTEN</td>
<td>Indicates connection indication received.</td>
</tr>
<tr>
<td>T_ORDREL</td>
<td>Indicates orderly release.</td>
</tr>
<tr>
<td>T_UDERR</td>
<td>Indicates datagram error.</td>
</tr>
</tbody>
</table>

If the t_look subroutine is unsuccessful, a value of -1 is returned, and the t_errno variable is set to indicate the error.

Error Codes
If unsuccessful, the t_errno variable is set to one of the following:

<table>
<thead>
<tr>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>TBADF</td>
<td>The specified file descriptor does not refer to a transport end point.</td>
</tr>
<tr>
<td>T_SYSSERR</td>
<td>A system error has occurred during execution of this function.</td>
</tr>
</tbody>
</table>
Related Information

The `t_open` subroutine.

List of Streams Programming References and STREAMS Overview in AIX 5L Version 5.3 Communications Programming Concepts.

---

**t_open Subroutine for Transport Layer Interface**

**Purpose**
Establishes a transport end point.

**Library**
Transport Layer Interface Library (libtli.a)

**Syntax**
```
#include <tiuser.h>

int t_open(path, oflag, info)
char *path;
int oflag;
struct t_info *info;
```

**Description**
The `t_open` subroutine must be called as the first step in the initialization of a transport end point. This subroutine establishes a transport end point, first, by opening a UNIX system file that identifies a particular transport provider (that is, transport protocol) and then returning a file descriptor that identifies that end point. For example, opening the `/dev/dlpi/tr` file identifies an 802.5 data link provider.

**Parameters**
- `path` Points to the path name of the file to open.
- `oflag` Specifies the open routine flags.
info Points to a `t_info` structure.

The `info` parameter points to a `t_info` structure that contains the following elements:

```c
long addr;
long options;
long tsdu;
long etsdu;
long connect;
long discon;
long servtype;
```

The values of the elements have the following meanings:

addr A value greater than or equal to 0 indicates the maximum size of a transport protocol address; a value of -1 specifies that there is no limit on the address size; and a value of -2 specifies that the transport provider does not provide user access to transport protocol addresses.

options A value greater than or equal to 0 indicates the maximum number of bytes of protocol-specific options supported by the provider; a value of -1 specifies that there is no limit on the option size; and a value of -2 specifies that the transport provider does not support user-settable options.

tsdu A value greater than 0 specifies the maximum size of a transport service data unit (TSDU); a value of 0 specifies that the transport provider does not support the concept of TSDU, although it does support the sending of a data stream having no logical boundaries preserved across a connection; a value of -1 specifies that there is no limit on the size of a TSDU; and a value of -2 specifies that the transfer of normal data is not supported by the transport provider.

etsdu A value greater than 0 specifies the maximum size of an expedited transport service data unit (ETSDU); a value of 0 specifies that the transport provider does not support the concept of ETSDU, although it does support the sending of an expedited data stream having no logical boundaries preserved across a connection; a value of -1 specifies that there is no limit on the size of an ETSDU; and a value of -2 specifies that the transfer of expedited data is not supported by the transport provider.

connect A value greater than or equal to 0 specifies the maximum amount of data that may be associated with connection establishment functions; a value of -1 specifies that there is no limit on the amount of data sent during connection establishment; and a value of -2 specifies that the transport provider does not allow data to be sent with connection establishment functions.

discon A value greater than or equal to 0 specifies the maximum amount of data that may be associated with the `t_snddis` and `t_rcvdis` functions; a value of -1 specifies that there is no limit on the amount of data sent with these abortive release functions; and a value of -2 specifies that the transport provider does not allow data to be sent with the abortive release functions.

servtype This field specifies the service type supported by the transport provider, as described in the Return Values section.

If a transport user is concerned with protocol independence, these sizes may be accessed to determine how large the buffers must be to hold each piece of information. Alternatively, the `t_alloc` subroutine can be used to allocate these buffers. An error results if a transport user exceeds the allowed data size on any function.

**Return Values**

On successful completion, the `t_open` subroutine returns a valid file descriptor. Otherwise, it returns a value of -1, and the `t_errno` variable is set to indicate the error.
The `servtype` field of the `info` parameter can specify one of the following values on return:

<table>
<thead>
<tr>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>T_COTS</td>
<td>The transport provider supports a connection-mode service but does not support the optional orderly release facility.</td>
</tr>
<tr>
<td>T_COTS_ORD</td>
<td>The transport provider supports a connection-mode service with the optional orderly release facility.</td>
</tr>
<tr>
<td>T_CLTS</td>
<td>The transport provider supports a connectionless-mode service. For this service type, the <code>t_open</code> subroutine returns -2 for the values in the <code>etsdu</code>, <code>connect</code>, and <code>discon</code> fields.</td>
</tr>
</tbody>
</table>

A single transport end point can support only one of the above services at one time.

If the `info` parameter is set to null by the transport user, no protocol information is returned by the `t_open` subroutine.

**Error Codes**

If unsuccessful, the `t_errno` variable is set to the following:

<table>
<thead>
<tr>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>TSYSERR</td>
<td>A system error has occurred during the startup of this function.</td>
</tr>
</tbody>
</table>

**Related Information**

The `open` subroutine, `t_close` subroutine.

[List of Streams Programming References](#) and [STREAMS Overview](#) in *AIX 5L Version 5.3 Communications Programming Concepts*.

---

### t_optmgmt Subroutine for Transport Layer Interface

**Purpose**

Manages options for a transport end point.

**Library**

Transport Layer Interface Library (libtli.a)

**Syntax**

```
#include <tiuser.h>

int t_optmgmt(fd, req, ret)

int fd;
struct t_optmgmt *req;
struct t_optmgmt *ret;
```

**Description**

The `t_optmgmt` subroutine enables a transport user to retrieve, verify, or negotiate protocol options with the transport provider.

**Parameters**

- `fd` Identifies a bound transport end point.
- `req` Requests a specific action of the provider.
Returns options and flag values to the user.

Both the `req` and `ret` parameters point to a `t_optmgmt` structure containing the following members:

```c
struct netbuf opt;
long flags;
```

The `opt` field identifies protocol options, and the `flags` field specifies the action to take with those options.

The options are represented by a `netbuf` structure in a manner similar to the address in the `t_bind` subroutine. The `req` parameter is used to send options to the provider. This `netbuf` structure contains the following fields:

<table>
<thead>
<tr>
<th>Field</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>len</td>
<td>Specifies the number of bytes in the options.</td>
</tr>
<tr>
<td>buf</td>
<td>Points to the options buffer.</td>
</tr>
<tr>
<td>maxlen</td>
<td>Has no meaning for the <code>req</code> parameter.</td>
</tr>
</tbody>
</table>

The `ret` parameter is used to return information to the user from the transport provider. On return, this `netbuf` structure contains the following fields:

<table>
<thead>
<tr>
<th>Field</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>len</td>
<td>Specifies the number of bytes of options returned.</td>
</tr>
<tr>
<td>buf</td>
<td>Points to the buffer where the options are to be placed.</td>
</tr>
<tr>
<td>maxlen</td>
<td>Specifies the maximum size of the options buffer. The <code>maxlen</code> field has no meaning for the <code>req</code> parameter, but must be set in the <code>ret</code> parameter to specify the maximum number of bytes the options buffer can hold. The actual structure and content of the options is imposed by the transport provider.</td>
</tr>
</tbody>
</table>

The `flags` field of the `req` parameter can specify one of the following actions:

<table>
<thead>
<tr>
<th>Action</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>T_NEGOTIATE</td>
<td>Enables the user to negotiate the values of the options specified in the <code>req</code> parameter with the transport provider. The provider evaluates the requested options and negotiates the values, returning the negotiated values through the <code>ret</code> parameter.</td>
</tr>
<tr>
<td>T_CHECK</td>
<td>Enables the user to verify if the options specified in the <code>req</code> parameter are supported by the transport provider. On return, the <code>flags</code> field of the <code>ret</code> parameter has either <code>T_SUCCESS</code> or <code>T_FAILURE</code> set to indicate to the user whether the options are supported or not. These flags are only meaningful for the <code>T_CHECK</code> request.</td>
</tr>
<tr>
<td>T_DEFAULT</td>
<td>Enables a user to retrieve the default options supported by the transport provider into the <code>opt</code> field of the <code>ret</code> parameter. In the <code>req</code> parameter, the <code>len</code> field of the <code>opt</code> field must be zero, and the <code>buf</code> field can be NULL.</td>
</tr>
</tbody>
</table>

If issued as part of the connectionless-mode service, the `t_optmgmt` subroutine may become blocked due to flow control constraints. The subroutine does not complete until the transport provider has processed all previously sent data units.

**Return Values**

On successful completion, the `t_optmgmt` subroutine returns a value of 0. Otherwise, it returns a value of -1, and the `t_errno` variable is set to indicate the error.
Error Codes

If unsuccessful, the `t_errno` variable is set to one of the following:

<table>
<thead>
<tr>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>TACCES</td>
<td>User does not have permission to negotiate the specified options.</td>
</tr>
<tr>
<td>TBADF</td>
<td>Specified file descriptor does not refer to a transport endpoint.</td>
</tr>
<tr>
<td>TBADFLAG</td>
<td>Unusable flag was specified.</td>
</tr>
<tr>
<td>TBADOPT</td>
<td>Specified protocol options were in an incorrect format or contained unusable information.</td>
</tr>
<tr>
<td>TBUFOVFLW</td>
<td>Number of bytes allowed for an incoming parameter is not sufficient to store the value of that parameter. Information to be returned in the <code>ret</code> parameter will be discarded.</td>
</tr>
<tr>
<td>TOUTSTATE</td>
<td>Function was issued in the wrong sequence.</td>
</tr>
<tr>
<td>TSYSERR</td>
<td>A system error has occurred during operation of this subroutine.</td>
</tr>
</tbody>
</table>

Related Information

The `t_getinfo` subroutine, `t_open` subroutine.

List of Streams Programming References and STREAMS Overview in AIX 5L Version 5.3 Communications Programming Concepts.

**t_rcv Subroutine for Transport Layer Interface**

**Purpose**

Receives normal data or expedited data sent over a connection.

**Library**

Transport Layer Interface Library (`libtli.a`)

**Syntax**

```c
int t_rcv(fd, buf, nbytes, flags)
```

```c
int fd;
char *buf;
unsigned nbytes;
int *flags;
```

**Description**

The `t_rcv` subroutine receives either normal or expedited data. By default, the `t_rcv` subroutine operates in synchronous mode and will wait for data to arrive if none is currently available. However, if the `O_NDELAY` flag is set (using the `t_open` subroutine or the `fcntl` command), the `t_rcv` subroutine runs in asynchronous mode and will stop if no data is available.

On return from the call, if the `T_MORE` flag is set in the `flags` parameter, this indicates that there is more data. This means that the current transport service data unit (TSDU) or expedited transport service data unit (ETSDU) must be received in multiple `t_rcv` subroutine calls. Each `t_rcv` subroutine with the `T_MORE` flag set indicates that another `t_rcv` subroutine must follow immediately to get more data for the current TSDU. The end of the TSDU is identified by the return of a `t_rcv` subroutine call with the `T_MORE` flag not set. If the transport provider does not support the concept of a TSDU as indicated in the `info` parameter on return from a `t_open` or `t_getinfo` subroutine, the `T_MORE` flag is not meaningful and should be ignored.

On return, the data returned is expedited data if the `T_EXPEDITED` flag is set in the `flags` parameter. If the number of bytes of expedited data exceeds the value in the `nbytes` parameter, the `t_rcv` subroutine will set the `T_EXPEDITED` and `T_MORE` flags on return from the initial call. Subsequent calls to retrieve the...
remaining ETSDU not have the T_EXPEDITED flag set on return. The end of the ETSDU is identified by the return of a t_rcv subroutine call with the T_MORE flag not set.

If expedited data arrives after part of a TSDU has been retrieved, receipt of the remainder of the TSDU will be suspended until the ETSDU has been processed. Only after the full ETSDU has been retrieved (the T_MORE flag is not set) will the remainder of the TSDU be available to the user.

**Parameters**

- **fd**
  Identifies the local transport end point through which data will arrive.
- **buf**
  Points to a receive buffer where user data will be placed.
- **nbytes**
  Specifies the size of the receiving buffer.
- **flags**
  Specifies optional flags.

**Return Values**

On successful completion, the t_rcv subroutine returns the number of bytes it received. Otherwise, it returns a value of -1 and sets the t_errno variable to indicate the error.

**Error Codes**

If unsuccessful, the t_errno variable may be set to one of the following:

<table>
<thead>
<tr>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>TBADF</td>
<td>The specified file descriptor does not refer to a transport end point.</td>
</tr>
<tr>
<td>TLOOK</td>
<td>An asynchronous event has occurred on this transport end point and requires immediate attention.</td>
</tr>
<tr>
<td>TNODATA</td>
<td>The O_NDELAY flag was set, but no data is currently available from the transport provider.</td>
</tr>
<tr>
<td>TNOTSUPPORT</td>
<td>This subroutine is not supported by the underlying transport provider.</td>
</tr>
<tr>
<td>TOUTSTATE</td>
<td>The subroutine was issued in the wrong sequence.</td>
</tr>
<tr>
<td>TSYSERR</td>
<td>A system error has occurred during operation of this subroutine.</td>
</tr>
</tbody>
</table>

**Related Information**

The t_getinfo subroutine, t_look subroutine, t_open subroutine, t_snd subroutine.

List of Streams Programming References and STREAMS Overview in AIX 5L Version 5.3 Communications Programming Concepts.

---

**t_rcvconnect Subroutine for Transport Layer Interface**

**Purpose**

Receives the confirmation from a connect request.

**Library**

Transport Layer Interface Library (libtli.a)

**Syntax**

```c
#include <tuser.h>

int t_rcvconnect(fd, call)

int fd;
struct t_call *call;
```

---

350 Technical Reference: Communications, Volume 2
Description

The **t_rcvconnect** subroutine enables a calling transport user to determine the status of a previously sent connect request and is used in conjunction with **t_connect** to establish a connection in asynchronous mode. The connection will be established on successful completion of this function.

Parameters

*fd*  Identifies the local transport end point where communication will be established.

*call*  Contains information associated with the newly established connection.

The *call* parameter points to a **t_call** structure that contains the following elements:

```c
struct netbuf addr;
struct netbuf opt;
struct netbuf udata;
int sequence;
```

The **netbuf** structure contains the following elements:

*addr*  Returns the protocol address associated with the responding transport end point.

*opt*  Presents protocol-specific information associated with the connection.

*udata*  Points to optional user data that may be returned by the destination transport user during connection establishment.

*sequence*  Has no meaning for this function.

The **maxlen** field of each parameter must be set before issuing this function to indicate the maximum size of the buffer for each. However, the *call* parameter may be null, in which case no information is given to the user on return from the **t_rcvconnect** subroutine. By default, the **t_rcvconnect** subroutine runs in synchronous mode and waits for the connection to be established before returning. On return, the *addr*, *opt*, and *udata* fields reflect values associated with the connection.

If the **O_NDELAY** flag is set (using the **t_open** subroutine or **fcntl** command), the **t_rcvconnect** subroutine runs in asynchronous mode and reduces to a poll for existing connect confirmations. If none are available, the **t_rcvconnect** subroutine stops and returns immediately without waiting for the connection to be established. The **t_rcvconnect** subroutine must be re-issued at a later time to complete the connection establishment phase and retrieve the information returned in the *call* parameter.

Return Values

On successful completion, the **t_rcvconnect** subroutine returns a value of 0. Otherwise, it returns a value of -1 and sets the **t_errno** variable to indicate the error.

Error Codes

If unsuccessful, the **t_errno** variable may be set to one of the following:

<table>
<thead>
<tr>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>TBADF</td>
<td>The specified file descriptor does not refer to a transport end point.</td>
</tr>
<tr>
<td>TBUFOVFLW</td>
<td>The number of bytes allocated for an incoming parameter is not sufficient to store the value of that parameter and the connect information to be returned in the <em>call</em> parameter will be discarded. The state of the provider, as seen by the user, will be changed to <strong>DATAXFER</strong>.</td>
</tr>
<tr>
<td>TLOOK</td>
<td>An asynchronous event has occurred on this transport connection and requires immediate attention.</td>
</tr>
<tr>
<td>TNODATA</td>
<td>The <strong>O_NDELAY</strong> flag was set, but a connect confirmation has not yet arrived.</td>
</tr>
<tr>
<td>TNOTSUPPORT</td>
<td>This subroutine is not supported by the underlying transport provider.</td>
</tr>
<tr>
<td>TOUTSTATE</td>
<td>This subroutine was issued in the wrong sequence.</td>
</tr>
</tbody>
</table>
TSYSERR  A system error has occurred during operation of this subroutine.

Related Information
The t_accept subroutine, t_alloc subroutine, t_bind subroutine, t_connect subroutine, t_listen subroutine, t_look subroutine, t_open subroutine.

List of Streams Programming References and STREAMS Overview in AIX 5L Version 5.3 Communications Programming Concepts.

---

**t_rcvdis Subroutine for Transport Layer Interface**

**Purpose**
Retrieves information from disconnect.

**Library**
Transport Layer Interface Library (libtli.a)

**Syntax**
```
#include <tiuser.h>

t_rcvdis(fd, discon)
int fd;
struct t_discon *discon;
```

**Description**
The t_rcvdis subroutine is used to identify the cause of a disconnect, and to retrieve any user data sent with the disconnect.

**Parameters**
- **fd** Identifies the local transport end point where the connection existed.
Points to a `t_discon` structure that contains the reason for the disconnect and contains any user data that was sent with the disconnect.

The `t_discon` structure contains the following members:

```c
struct netbuf udata;
int reason;
int sequence;
```

These fields are defined as follows:

- `<reason>` Specifies the reason for the disconnect through a protocol-dependent reason code.
- `<udata>` Identifies any user data that was sent with the disconnect.
- `<sequence>` Identifies an outstanding connect indication with which the disconnect is associated. The sequence field is only meaningful when the `t_rcvdis` subroutine is issued by a passive transport user that has called one or more `t_listen` subroutines and is processing the resulting connect indications. If a disconnect indication occurs, the sequence field can be used to identify which of the outstanding connect indications is associated with the disconnect.

If a user does not care if there is incoming data and does not need to know the value of the `<reason>` or `<sequence>` fields, the `discon` parameter may be null and any user data associated with the disconnect will be discarded. However, if a user has retrieved more than one outstanding connect indication (using the `t_listen` subroutine) and the `discon` parameter is null, the user will be unable to identify with which connect indication the disconnect is associated.

**Return Values**

On successful completion, the `t_rcvdis` subroutine returns a value of 0. Otherwise, it returns a value of -1 and sets the `t_errno` variable to indicate the error.

**Error Codes**

If unsuccessful, the `t_errno` variable may be set to one of the following:

<table>
<thead>
<tr>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>TBADF</td>
<td>The specified file descriptor does not refer to a transport end point.</td>
</tr>
<tr>
<td>TBUFOVFLW</td>
<td>The number of bytes allocated for incoming data is not sufficient to store the data. (The state of the provider, as seen by the user, will change to T_IDLE, and the disconnect indication information to be returned in the discon parameter will be discarded.)</td>
</tr>
<tr>
<td>TLOOK</td>
<td>An asynchronous event has occurred on this transport endpoint and requires immediate attention.</td>
</tr>
<tr>
<td>TNODIS</td>
<td>No disconnect indication currently exists on the specified transport end point.</td>
</tr>
<tr>
<td>TNOTSUPPORT</td>
<td>This function is not supported by the underlying transport provider.</td>
</tr>
<tr>
<td>TOUTSTATE</td>
<td>This subroutine was issued in the wrong sequence.</td>
</tr>
<tr>
<td>TSYSERR</td>
<td>A system error has occurred during execution of this subroutine.</td>
</tr>
</tbody>
</table>

**Related Information**

The `t_alloc` subroutine, `t_connect` subroutine, `t_listen` subroutine, `t_open` subroutine, `t_snddis` subroutine.

[Link to List of Streams Programming References and STREAMS Overview in AIX 5L Version 5.3 Communications Programming Concepts.]
t_rcvrel Subroutine for Transport Layer Interface

Purpose
Acknowledges receipt of an orderly release indication.

Library
Transport Layer Interface Library (libtli.a)

Syntax
```
#include <tiuser.h>

int t_rcvrel(int fd);
```

Description
The t_rcvrel subroutine is used to acknowledge receipt of an orderly release indication. After receipt of this indication, the user may not attempt to receive more data because such an attempt will block forever. However, the user may continue to send data over the connection if the t_sndrel subroutine has not been issued by the user. The subroutine is an optional service of the transport provider, and is only supported if the transport provider returned service type T_COTS_ORD on the t_open or t_getinfo subroutine.

Parameter

`fd` Identifies the local transport end point where the connection exists.

Return Values
On successful completion, the t_rcvrel subroutine returns a value of 0. Otherwise, it returns a value of -1 and sets the t_errno variable to indicate the error.

Error Codes
If unsuccessful, the t_errno variable is set to one of the following:

<table>
<thead>
<tr>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>TBADF</td>
<td>The specified file descriptor does not refer to a transport end point.</td>
</tr>
<tr>
<td>TLOOK</td>
<td>An asynchronous event has occurred on this transport end point and requires immediate attention.</td>
</tr>
<tr>
<td>TNOREL</td>
<td>No orderly release indication currently exists on the specified transport end point.</td>
</tr>
<tr>
<td>TNOT SUPPORT</td>
<td>This subroutine is not supported by the underlying transport provider.</td>
</tr>
<tr>
<td>TOUTSTATE</td>
<td>This subroutine was issued in the wrong sequence.</td>
</tr>
<tr>
<td>TSYSERRE</td>
<td>A system error has occurred during execution of this function.</td>
</tr>
</tbody>
</table>

Related Information
The t_getinfo subroutine, t_look subroutine, t_open subroutine, t_sndrel subroutine.

List of Streams Programming References and STREAMS Overview in AIX 5L Version 5.3 Communications Programming Concepts.
t_rcvudata Subroutine for Transport Layer Interface

Purpose
Receives a data unit.

Library
Transport Layer Interface Library (libtli.a)

Syntax
#include <tiuser.h>

int t_rcvudata(fd, unitdata, flags)
int fd;
struct t_unitdata * unitdata;
int * flags;

Description
The t_rcvudata subroutine is used in connectionless mode to receive a data unit from another transport user.

Parameters
fd Identifies the local transport end point through which data will be received.
unitdata Holds information associated with the received data unit.

The unitdata parameter points to a t_unitdata structure containing the following members:
struct netbuf addr;
struct netbuf opt;
struct netbuf udata;

On return from this call:
addr Specifies the protocol address of the sending user.
opt Identifies protocol-specific options that were associated with this data unit.
udata Specifies the user data that was received.

Note: The maxlen field of the addr, opt, and udata fields must be set before issuing this function to indicate the maximum size of the buffer for each.

flags Indicates that the complete data unit was not received.

By default, the t_rcvudata subroutine operates in synchronous mode and will wait for a data unit to arrive if none is currently available. However, if the O_NDELAY or O_NONBLOCK flag is set (using the t_open subroutine or fcntl command), the t_rcvudata subroutine will run in asynchronous mode and will stop if no data units are available.

If the buffer defined in the udata field of unitdata is not large enough to hold the current data unit, the buffer will be filled and the T_MORE flag will be set in flags on return to indicate that another t_rcvudata subroutine should be issued to retrieve the rest of the data unit. Subsequent t_rcvudata subroutine calls will return 0 for the length of the address and options until the full data unit has been received.

Return Values
On successful completion, the t_rcvudata subroutine returns a value of 0. Otherwise, it returns a value of -1 and sets the t_errno variable to indicate the error.
Error Codes

If unsuccessful, the t_errno variable is set to one of the following:

<table>
<thead>
<tr>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>TBADF</td>
<td>The specified file descriptor does not refer to a transport end point.</td>
</tr>
<tr>
<td>TBUFOVFLW</td>
<td>The number of bytes allocated for the incoming protocol address of options is not sufficient to store the information. (The unit data information to be returned in the unitdata parameter will be discarded.)</td>
</tr>
<tr>
<td>TLOOK</td>
<td>An asynchronous event has occurred on this transport endpoint and requires immediate attention.</td>
</tr>
<tr>
<td>TNODATA</td>
<td>The O_DELAY or O_NONBLOCK flag was set, but no data units are currently available from the transport provider.</td>
</tr>
<tr>
<td>TNOTSUPPORT</td>
<td>This subroutine is not supported by the underlying transport provider.</td>
</tr>
<tr>
<td>TOUTSTATE</td>
<td>The subroutine was issued in the wrong sequence.</td>
</tr>
<tr>
<td>TSYSERR</td>
<td>A system error has occurred during operation of this subroutine.</td>
</tr>
</tbody>
</table>

Related Information

The t_alloc subroutine, t_open subroutine, t_rcvuderr subroutine, t_sndudata subroutine.

List of Streams Programming References and STREAMS Overview in AIX 5L Version 5.3 Communications Programming Concepts.

**t_rcvuderr Subroutine for Transport Layer Interface**

**Purpose**

Receives a unit data error indication.

**Library**

Transport Layer Interface Library (libtli.a)

**Syntax**

```
#include <tiuser.h>

int t_rcvuderr(fd, udderr)
int fd;
struct t_uderr *udderr;
```

**Description**

The t_rcvuderr subroutine is used in connectionless mode to receive information concerning an error on a previously sent data unit, and should only be issued following a unit data error indication. It informs the transport user that a data unit with a specific destination address and protocol options produced an error.

**Parameters**

- `fd`: Identifies the local transport endpoint through which the error report will be received.
uderr Points to a **t_uderr** structure containing the following members:

```c
struct netbuf addr;
struct netbuf opt;
long error;
```

The `maxlen` field of the `addr` and `opt` fields must be set before issuing this function to indicate the maximum size of the buffer for each.

On return from this call, the **t_uderr** structure contains:

- `addr` Specifies the destination protocol address of the erroneous data unit.
- `opt` Identifies protocol-specific options that were associated with the data unit.
- `error` Specifies a protocol-dependent error code.

If the user decides not to identify the data unit that produced an error, the `uderr` parameter can be set to `null` and the **t_rcvuderr** subroutine will clear the error indication without reporting any information to the user.

**Return Values**

On successful completion, the **t_rcvuderr** subroutine returns a value of 0. Otherwise, it returns a value of -1 and sets the **t_errno** variable to indicate the error.

**Error Codes**

If unsuccessful, the **t_errno** variable is set to one of the following:

<table>
<thead>
<tr>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>TBADF</td>
<td>The specified file descriptor does not refer to a transport end point.</td>
</tr>
<tr>
<td>TNOUDERR</td>
<td>No unit data error indication currently exists on the specified transport end point.</td>
</tr>
<tr>
<td>TBUFOVFLW</td>
<td>The number of bytes allocated for the incoming protocol address or options is not sufficient to store the information. (The unit data error information to be returned in the <code>uderr</code> parameter will be discarded.)</td>
</tr>
<tr>
<td>TNOTSUPPORT</td>
<td>This subroutine is not supported by the underlying transport provider.</td>
</tr>
<tr>
<td>TSYSERR</td>
<td>A system error has occurred during execution of this subroutine.</td>
</tr>
</tbody>
</table>

**Related Information**

The **t_look** subroutine, **t_rcvudata** subroutine, **t_sndudata** subroutine.

[List of Streams Programming References](AIX 5L Version 5.3 Communications Programming Concepts) and [STREAMS Overview](AIX 5L Version 5.3 Communications Programming Concepts).

---

**t_snd Subroutine for Transport Layer Interface**

**Purpose**

Sends data or expedited data over a connection.

**Library**

Transport Layer Interface Library (**libtli.a**)  

**Syntax**

```c
#include <tiusert.h>
```
int t_snd(fd, buf, nbytes, flags)
int fd;
char *buf;
unsigned nbytes;
int flags;

Description
The t_snd subroutine is used to send either normal or expedited data.

By default, the t_snd subroutine operates in synchronous mode and may wait if flow-control restrictions prevent the data from being accepted by the local transport provider at the time the call is made. However, if the O_NDELAY or O_NONBLOCK flag is set (using the t_open subroutine or the fcntl command), the t_snd subroutine runs in asynchronous mode and stops immediately if there are flow-control restrictions.

Even when there are no flow-control restrictions, the t_snd subroutine will wait if STREAMS internal resources are not available, regardless of the state of the O_NDELAY or O_NONBLOCK flag.

On successful completion, the t_snd subroutine returns the number of bytes accepted by the transport provider. Normally this equals the number of bytes specified in the nbytes parameter. However, if the O_NDELAY or O_NONBLOCK flag is set, it is possible that only part of the data will be accepted by the transport provider. In this case, the t_snd subroutine sets the T_MORE flag for the data that was sent and returns a value less than the value of the nbytes parameter. If the value of the nbytes parameter is 0, no data is passed to the provider and the t_snd subroutine returns a value of 0.

Parameters

fd Identifies the local transport end point through which data is sent.
buf Points to the user data.
nbytes Specifies the number of bytes of user data to be sent.
flags Specifies any optional flags.

If the T_EXPEDITED flag is set in the flags parameter, the data is sent as expedited data and is subject to the interpretations of the transport provider.

If the T_MORE flag is set in the flags parameter, or as described above, an indication is sent to the transport provider that the transport service data unit (TSDU) or expedited transport service data unit (ETSDU) is being sent through multiple t_snd subroutine calls. Each t_snd subroutine with the T_MORE flag set indicates that another t_snd subroutine will follow with more data for the current TSDU. The end of the TSDU or ETSDU is identified by a t_snd subroutine call with the T_MORE flag not set. Use of the T_MORE flag enables a user to break up large logical data units without losing the boundaries of those units at the other end of the connection. The flag implies nothing about how the data is packaged for transfer below the transport interface. If the transport provider does not support the concept of a TSDU as indicated in the info parameter on return from the t_open or t_getinfo subroutine, the T_MORE flag is not meaningful and should be ignored.

The size of each TSDU or ETSDU must not exceed the limits of the transport provider as returned by the t_open or t_getinfo subroutine. If the size is exceeded, a TSYSERR error with system error EPROTO occurs. However, the t_snd subroutine may not fail because EPROTO errors may not be reported immediately. In this case, a subsequent call that accesses the transport endpoint fails with the associated TSYSERR error.

If the call to the t_snd subroutine is issued from the T_IDLE state, the provider may silently discard the data. If the call to the t_snd subroutine is issued from any state other than T_DATAXFER, T_INREL, or T_IDLE, the provider generates a TSYSERR error with system error EPROTO (which can be reported in the manner described above).
Return Values
On successful completion, the t_snd subroutine returns the number of bytes accepted by the transport provider. Otherwise, it returns a value of -1 and sets the t_errno variable to indicate the error.

Error Codes
If unsuccessful, the t_errno variable is set to one of the following:

<table>
<thead>
<tr>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>TBADDATA</td>
<td>The amount of user data specified was not within the bounds allowed by the transport provider.</td>
</tr>
<tr>
<td>TBADF</td>
<td>The specified file descriptor does not refer to a transport end point.</td>
</tr>
<tr>
<td>TBADFLAG</td>
<td>The value specified in the flags parameter is invalid.</td>
</tr>
<tr>
<td>TFLOW</td>
<td>The O_NDELAY or O_NONBLOCK flag was set, but the flow-control mechanism prevented the</td>
</tr>
<tr>
<td></td>
<td>transport provider from accepting data at this time.</td>
</tr>
<tr>
<td>TLOOK</td>
<td>An asynchronous event has occurred on the transport end point reference by the fd parameter</td>
</tr>
<tr>
<td></td>
<td>and requires immediate attention.</td>
</tr>
<tr>
<td>TNOTSUPPORT</td>
<td>This subroutine is not supported by the underlying transport provider.</td>
</tr>
<tr>
<td>TOUTSTATE</td>
<td>The subroutine was issued in the wrong sequence.</td>
</tr>
<tr>
<td>TSYSERR</td>
<td>A system error has been detected during execution of this subroutine.</td>
</tr>
</tbody>
</table>

Related Information
The t_getinfo subroutine, t_getstate subroutine, t_open subroutine, t_rcv subroutine.

List of Streams Programming References and STREAMS Overview in AIX 5L Version 5.3 Communications Programming Concepts.

**t_snddis Subroutine for Transport Layer Interface**

**Purpose**
Sends a user-initiated disconnect request.

**Library**
Transport Layer Interface Library (libtli.a)

**Syntax**
```
#include <tiuser.h>

int t_snddis(fd, call)

int fd;
struct t_call *call;
```

**Description**
The t_snddis subroutine is used to initiate an abortive release on an already established connection or to reject a connect request.

**Parameters**

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>fd</td>
<td>Identifies the local transport end point of the connection.</td>
</tr>
</tbody>
</table>
Specifies information associated with the abortive release.

The **call** parameter points to a **t_call** structure containing the following fields:

```c
struct netbuf addr;
struct netbuf opt;
struct netbuf udata;
int sequence;
```

The values in the **call** parameter have different semantics, depending on the context of the call to the **t_snddis** subroutine. When rejecting a connect request, the **call** parameter must not be null and must contain a valid value in the **sequence** field to uniquely identify the rejected connect indication to the transport provider. The **addr** and **opt** fields of the **call** parameter are ignored. In all other cases, the **call** parameter need only be used when data is being sent with the disconnect request. The **addr**, **opt**, and **sequence** fields of the **t_call** structure are ignored. If the user does not wish to send data to the remote user, the value of the **call** parameter can be null.

The **udata** field specifies the user data to be sent to the remote user. The amount of user data must not exceed the limits supported by the transport provider as returned by the **t_open** or **t_getinfo** subroutine. If the **len** field of the **udata** field is 0, no data will be sent to the remote user.

**Return Values**

On successful completion, the **t_snddis** subroutine returns a value of 0. Otherwise, it returns a value of -1 and sets the **t_errno** variable to indicate the error.

**Error Codes**

If unsuccessful, the **t_errno** variable is set to one of the following:

<table>
<thead>
<tr>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>TBADF</td>
<td>The specified file descriptor does not refer to a transport end point.</td>
</tr>
<tr>
<td>TOUTSTATE</td>
<td>The function was issued in the wrong sequence. The transport provider’s outgoing queue may be flushed, so data may be lost.</td>
</tr>
<tr>
<td>TBADDATA</td>
<td>The amount of user data specified was not within the bounds allowed by the transport provider. The transport provider’s outgoing queue will be flushed, so data may be lost.</td>
</tr>
<tr>
<td>TBADSEQ</td>
<td>An incorrect sequence number was specified, or a null call structure was specified when rejecting a connect request. The transport provider’s outgoing queue will be flushed, so data may be lost.</td>
</tr>
<tr>
<td>TLOOK</td>
<td>An asynchronous event has occurred on this transport endpoint and requires immediate attention.</td>
</tr>
<tr>
<td>TNOTSUPPORT</td>
<td>This subroutine is not supported by the underlying transport provider.</td>
</tr>
<tr>
<td>TSYSERR</td>
<td>A system error has occurred during execution of this subroutine.</td>
</tr>
</tbody>
</table>

**Related Information**

The **t_connect** subroutine, **t_getinfo** subroutine, **t_listen** subroutine, **t_look** subroutine, **t_open** subroutine.

[List of Streams Programming References](#) and [STREAMS Overview](#) in *AIX 5L Version 5.3 Communications Programming Concepts*.

---

**t_sndrel Subroutine for Transport Layer Interface**

**Purpose**

Initiates an orderly release of a transport connection.
Library
Transport Layer Interface Library (libtli.a)

Syntax
#include <tiuser.h>

int t_sndrel(fd)

Description
The t_sndrel subroutine is used to initiate an orderly release of a transport connection and indicates to the transport provider that the transport user has no more data to send.

After issuing a t_sndrel subroutine call, the user cannot send any more data over the connection. However, a user can continue to receive data if an orderly release indication has been received.

The t_sndrel subroutine is an optional service of the transport provider and is only supported if the transport provider returned service type T_COTS_ORD in the t_open or t_getinfo subroutine.

Parameter

fd Identifies the local transport endpoint where the connection exists.

Return Values
On successful completion, the t_sndrel subroutine returns a value of 0. Otherwise, it returns a value of -1 and sets the t_errno variable to indicate the error.

Error Codes
If unsuccessful, the t_errno variable is set to one of the following:

<table>
<thead>
<tr>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>TBADF</td>
<td>The specified file descriptor does not refer to a transport endpoint.</td>
</tr>
<tr>
<td>TFLOW</td>
<td>The O_NDELAY or O_NONBLOCK flag was set, but the flow-control mechanism prevented the transport provider from accepting the function at this time.</td>
</tr>
<tr>
<td>TLOOK</td>
<td>An asynchronous event has occurred on the transport end point reference by the fd parameter and requires immediate attention.</td>
</tr>
<tr>
<td>TNOTSUPPORT</td>
<td>This subroutine is not supported by the underlying transport provider.</td>
</tr>
<tr>
<td>TOUTSTATE</td>
<td>The subroutine was issued in the wrong sequence.</td>
</tr>
<tr>
<td>TSYSERR</td>
<td>A system error has occurred during execution of this subroutine.</td>
</tr>
</tbody>
</table>

Related Information
The t_getinfo subroutine, t_open subroutine, t_rcvrel subroutine.

List of Streams Programming References and STREAMS Overview in AIX 5L Version 5.3 Communications Programming Concepts.

_t_sndudata Subroutine for Transport Layer Interface

Purpose
Sends a data unit to another transport user.
Library
Transport Layer Interface Library (libtli.a)

Syntax
#include <tiuser.h>

int t_sndudata(fd, unitdata)
int fd;
struct t_unitdata *unitdata;

Description
The t_sndudata subroutine is used in connectionless mode to send a data unit to another transport user.

By default, the t_sndudata subroutine operates in synchronous mode and may wait if flow-control restrictions prevent the data from being accepted by the local transport provider at the time the call is made. However, if the O_NDELAY or O_NONBLOCK flag is set (using the t_open subroutine or the fcntl command), the t_sndudata subroutine runs in asynchronous mode and fails under such conditions.

Parameters
fd Identifies the local transport endpoint through which data is sent.
unitdata Points to a t_unitdata structure containing the following elements:
struct netbuf addr;
struct netbuf opt;
struct netbuf udata;

The elements are defined as follows:
addr Specifies the protocol address of the destination user.
opt Identifies protocol-specific options that the user wants associated with this request.
udata Specifies the user data to be sent. The user can choose not to specify what protocol options are associated with the transfer by setting the len field of the opt field to 0. In this case, the provider can use default options.

If the len field of the udata field is 0, no data unit is passed to the transport provider; the t_sndudata subroutine does not send zero-length data units.

If the t_sndudata subroutine is issued from an invalid state, or if the amount of data specified in the udata field exceeds the TSDU size as returned by the t_open or t_getinfo subroutine, the provider generates an EPROTO protocol error.

Return Values
On successful completion, the t_sndudata subroutine returns a value of 0. Otherwise, it returns a value of -1 and sets the t_errno variable to indicate the error.

Error Codes
If unsuccessful, the t_errno variable is set to one of the following:

<table>
<thead>
<tr>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>TBADDATA</td>
<td>The amount of user data specified was not within the bounds allowed by the transport provider.</td>
</tr>
<tr>
<td>TBADF</td>
<td>The specified file descriptor does not refer to a transport endpoint.</td>
</tr>
<tr>
<td>TFLOW</td>
<td>The O_NDELAY or O_NONBLOCK flag was set, but the flow-control mechanism prevented the transport provider from accepting data at this time.</td>
</tr>
<tr>
<td>Value</td>
<td>Description</td>
</tr>
<tr>
<td>---------</td>
<td>-----------------------------------------------------------------------------</td>
</tr>
<tr>
<td>TLOOK</td>
<td>An asynchronous event has occurred on this transport endpoint and requires immediate attention.</td>
</tr>
<tr>
<td>TNOTSUPPORT</td>
<td>This subroutine is not supported by the underlying transport provider.</td>
</tr>
<tr>
<td>TOUTSTATE</td>
<td>This subroutine was issued in the wrong sequence.</td>
</tr>
<tr>
<td>TSYSERR</td>
<td>A system error has occurred during execution of this subroutine.</td>
</tr>
</tbody>
</table>

**Related Information**
The `t_alloc` subroutine, `t_open` subroutine, `t_rcvudata` subroutine, `t_rcvuderr` subroutine.

List of Streams Programming References and STREAMS Overview in *AIX 5L Version 5.3 Communications Programming Concepts.*

---

### t_sync Subroutine for Transport Layer Interface

**Purpose**
Synchronizes transport library.

**Library**
Transport Layer Interface Library (*libtli.a*)

**Syntax**
```
#include <tiuser.h>

int t_sync(int fd);
```

**Description**
The `t_sync` subroutine synchronizes the data structures managed by the transport library with information from the underlying transport provider. In doing so, this subroutine can convert a raw file descriptor (obtained using the `open` or `dup` subroutine, or as a result of a `fork` operation and an `exec` operation) to an initialized transport endpoint, assuming that the file descriptor referenced a transport provider. This subroutine also allows two cooperating processes to synchronize their interaction with a transport provider.

For example, a process creates a new process with the `fork` subroutine and issues an `exec` subroutine call. The new process must issue a `t_sync` subroutine call to build the private library data structure associated with a transport endpoint and to synchronize the data structure with the relevant provider information.

**Note:** The transport provider treats all users of a transport endpoint as a single user. If multiple processes are using the same endpoint, they should coordinate their activities so as not to violate the state of the provider. The `t_sync` subroutine returns the current state of the provider to the user, thereby enabling the user to verify the state before taking further action. This coordination is only valid among cooperating processes; a process or an incoming event may change the provider’s state *after* a `t_sync` subroutine call is issued.

If the provider is undergoing a state transition when the `t_sync` subroutine is called, the subroutine will be unsuccessful.
Parameters

*fd* Specifies the transport end point.

Return Values

On successful completion, the `t_sync` subroutine returns the state of the transport provider. Otherwise, it returns a value of -1 and sets the `t_errno` variable to indicate the error. The state returned can be one of the following:

<table>
<thead>
<tr>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>T_UNBIND</td>
<td>Unbound</td>
</tr>
<tr>
<td>T_IDLE</td>
<td>Idle</td>
</tr>
<tr>
<td>T_OUTCON</td>
<td>Outgoing connection pending</td>
</tr>
<tr>
<td>T_INCON</td>
<td>Incoming connection pending</td>
</tr>
<tr>
<td>T_DATAXFER</td>
<td>Data transfer</td>
</tr>
<tr>
<td>T_OUTREL</td>
<td>Outgoing orderly release (waiting for an orderly release indication)</td>
</tr>
<tr>
<td>T_INREL</td>
<td>Incoming orderly release (waiting for an orderly release request)</td>
</tr>
</tbody>
</table>

Error Codes

If unsuccessful, the `t_errno` variable is set to one of the following:

<table>
<thead>
<tr>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>TBADF</td>
<td>The specified file descriptor is a valid open file descriptor, but does not refer to a transport endpoint.</td>
</tr>
<tr>
<td>TSTATECHNG</td>
<td>The transport provider is undergoing a state change.</td>
</tr>
<tr>
<td>TSYSERR</td>
<td>A system error has occurred during execution of this function.</td>
</tr>
</tbody>
</table>

Related Information

The `dup` subroutine, `exec` subroutine, `fork` subroutine, `open` subroutine.

[List of Streams Programming References](#) and [STREAMS Overview](#) in *AIX 5L Version 5.3 Communications Programming Concepts*.

**t_unbind Subroutine for Transport Layer Interface**

Purpose

Disables a transport endpoint.

Library

Transport Layer Interface Library (*libtli.a*)

Syntax

```c
#include <tiuser.h>

int t_unbind(fd)
    int fd;
```
Description
The **t_unbind** subroutine disables a transport endpoint, which was previously bound by the **t_bind** subroutine. On completion of this call, no further data or events destined for this transport endpoint are accepted by the transport provider.

Parameter

```
fd    Specifies the transport endpoint.
```

Return Values
On successful completion, the **t_unbind** subroutine returns a value of 0. Otherwise, it returns a value of -1 and sets the **t_errno** variable to indicate the error.

Error Codes
If unsuccessful, the **t_errno** variable is set to one of the following:

<table>
<thead>
<tr>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>TBADF</td>
<td>The specified file descriptor does not refer to a transport endpoint.</td>
</tr>
<tr>
<td>TOUTSTATE</td>
<td>The function was issued in the wrong sequence.</td>
</tr>
<tr>
<td>TLOOK</td>
<td>An asynchronous event has occurred on this transport endpoint.</td>
</tr>
<tr>
<td>TSYSERR</td>
<td>A system error has occurred during execution of this function.</td>
</tr>
</tbody>
</table>

Related Information
The **t_bind** subroutine.

List of Streams Programming References and STREAMS Overview in AIX 5L Version 5.3 Communications Programming Concepts.

testb Utility

Purpose
Checks for an available buffer.

Syntax

```
int testb(size, pri, register size, uint pri)
```

Description
The **testb** utility checks for the availability of a message buffer of the size specified in the **size** parameter without actually retrieving the buffer. A successful return value from the **testb** utility does not guarantee that a subsequent call to the **allocb** utility will succeed; for example, when an interrupt routine takes the buffers.

This utility is part of STREAMS Kernel Extensions.

Parameters

```
size    Specifies the buffer size.
```
Specifies the relative importance of the allocated blocks to the module. The possible values are:

- **BPRI_LO**
- **BPRI_MED**
- **BPRI_HI**

The *pri* parameter is currently unused and is maintained only for compatibility with applications developed prior to UNIX System V Release 4.0.

**Return Values**

If the buffer is available, the *testb* utility returns a value of 1. Otherwise, it returns a value of 0.

**Related Information**

The *allocb* utility.

List of Streams Programming References and Understanding STREAMS Flow Control in *AIX 5L Version 5.3 Communications Programming Concepts*.

---

### timeout Utility

**Purpose**

Schedules a function to be called after a specified interval.

**Syntax**

```c
int timeout(func, arg, ticks)
int (*func)();
caddr_t arg;
long ticks;
```

**Description**

The *timeout* utility schedules the function pointed to by the *func* parameter to be called with the *arg* parameter after the number of timer ticks specified by the *ticks* parameter. Multiple pending calls to the *timeout* utility with the same *func* and *arg* parameters are allowed. The function called by the *timeout* utility must adhere to the same restrictions as a driver interrupt handler. It must not sleep.

On multiprocessor systems, the function called by the *timeout* utility should be interrupt-safe. Otherwise, the STR_QSAFETY flag must be set when installing the module or driver with the *str_install* utility.

This utility is part of STREAMS Kernel Extension.

**Note:** This utility must not be confused with the kernel service of the same name in the *libsys.a* library. STREAMS modules and drivers inherently use this version, not the *libsys.a* library version. No special action is required to use this version in the STREAMS environment.

**Parameters**

- **func** Indicates the function to be called. The function is declared as follows:
  ```c
  void (*func)(arg)
  void *arg;
  ```
- **arg** Indicates the parameter to supply to the function specified by the *func* parameter.
ticks Specifies the number of timer ticks that must occur before the function specified by the func parameter is called. Many timer ticks can occur every second.

Return Values
The timeout utility returns an integer that identifies the request. This value may be used to withdraw the time-out request by using the umtimeout utility. If the timeout table is full, the timeout utility returns a value of 0 and the request is not registered.

Execution Environment
The timeout utility may be called from either the process or interrupt environment.

Related Information
The umtimeout utility.

List of Streams Programming References in AIX 5L Version 5.3 Communications Programming Concepts.

Understanding STREAMS Drivers and Modules in AIX 5L Version 5.3 Communications Programming Concepts.

Understanding STREAMS Synchronization in AIX 5L Version 5.3 Communications Programming Concepts.

timod Module

Purpose
Converts a set of streamio operations into STREAMS messages.

Description
The timod module is a STREAMS module for use with the Transport Interface (TI) functions of the Network Services Library. The timod module converts a set of streamio operations into STREAMS messages that may be consumed by a transport protocol provider that supports the Transport Interface. This allows a user to initiate certain TI functions as atomic operations.

The timod module must only be pushed (see Pushable Modules in AIX 5L Version 5.3 Communications Programming Concepts) onto a stream terminated by a transport protocol provider that supports the TI.

All STREAMS messages, with the exception of the message types generated from the streamio operations described below as values for the cmd field, will be transparently passed to the neighboring STREAMS module or driver. The messages generated from the following streamio operations are recognized and processed by the timod module.

This module is part of STREAMS Kernel Extensions.
Fields
The fields are described as follows:

<table>
<thead>
<tr>
<th>Field</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>cmd</td>
<td>Specifies the command to be carried out. The possible values for this field are:</td>
</tr>
<tr>
<td></td>
<td><strong>TI_BIND</strong> Binds an address to the underlying transport protocol provider. The message issued to the TI_BIND operation is equivalent to the TI message type T_BIND_REQ, and the message returned by the successful completion of the operation is equivalent to the TI message type T_BIND_ACK.</td>
</tr>
<tr>
<td></td>
<td><strong>TI_UNBIND</strong> Unbinds an address from the underlying transport protocol provider. The message issued to the TI_UNBIND operation is equivalent to the TI message type T_UNBIND_REQ, and the message returned by the successful completion of the operation is equivalent to the TI message type T_OK_ACK.</td>
</tr>
<tr>
<td></td>
<td><strong>TI_GETINFO</strong> Gets the TI protocol-specific information from the transport protocol provider. The message issued to the TI_GETINFO operation is equivalent to the TI message type T_INFO_REQ, and the message returned by the successful completion of the operation is equivalent to the TI message type T_INFO_ACK.</td>
</tr>
<tr>
<td></td>
<td><strong>TI_OPTMGMT</strong> Gets, sets, or negotiates protocol-specific options with the transport protocol provider. The message issued to the TI_OPTMGMT ioctl operation is equivalent to the TI message type T_OPTMGMT_REQ, and the message returned by the successful completion of the ioctl operation is equivalent to the TI message type T_OPTMGMT_ACK.</td>
</tr>
<tr>
<td>len</td>
<td>(On issuance) Specifies the size of the appropriate TI message to be sent to the transport provider. (On return) Specifies the size of the appropriate TI message from the transport provider in response to the issued TI message.</td>
</tr>
<tr>
<td>dp</td>
<td>Specifies a pointer to a buffer large enough to hold the contents of the appropriate TI messages. The TI message types are defined in the sys/tihdr.h file.</td>
</tr>
</tbody>
</table>

Examples
The following is an example of how to use the timod module:

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

struct strioctl strioctl;
struct t_info info;

strioctl.ic_cmd = TI_GETINFO;
strioctl.ic_timeout = INFTIM;
strioctl.ic_len = sizeof(info);
strioctl.ic_dp = (char *)&info;
ioctl(fildes, I_STR, &strioctl);
```

Related Information
The [tirdwr] module.

The [streamio] operations.

[Benefits and Features of STREAMS] [Building STREAMS] [Pushable Modules] [Understanding STREAMS in AIX 5L Version 5.3] [Drivers and Modules] [Understanding STREAMS Messages] [Using STREAMS]

Communications Programming Concepts.
**tirdwr Module**

**Purpose**

Supports the Transport Interface functions of the Network Services library.

**Description**

The `tirdwr` module is a STREAMS module that provides an alternate interface to a transport provider that supports the Transport Interface (TI) functions of the Network Services library. This alternate interface allows a user to communicate with the transport protocol provider by using the `read` and `write` subroutines. The `putmsg` and `getmsg` system calls can also be used. However, the `putmsg` and `getmsg` system calls can only transfer data messages between user and stream.

The `tirdwr` module must only be pushed (see the `I_PUSH` operation) onto a stream terminated by a transport protocol provider that supports the TI. After the `tirdwr` module has been pushed onto a stream, none of the TI functions can be used. Subsequent calls to TI functions will cause an error on the stream. Once the error is detected, subsequent system calls on the stream will return an error with the `errno` global variable set to `EPROTO`.

The following list describes actions taken by the `tirdwr` module when it is pushed or popped or when data passes through it:

<table>
<thead>
<tr>
<th>Action</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>push</td>
<td>Checks any existing data to ensure that only regular data messages are present. It ignores any messages on the stream that relate to process management. If any other messages are present, the <code>I_PUSH</code> operation returns an error and sets the <code>errno</code> global variable to <code>EPROTO</code>.</td>
</tr>
<tr>
<td>write</td>
<td>Takes the following actions on data that originated from a <code>write</code> subroutine:</td>
</tr>
<tr>
<td></td>
<td><strong>Messages with no control portions</strong></td>
</tr>
<tr>
<td></td>
<td>Passes the message on downstream.</td>
</tr>
<tr>
<td></td>
<td><strong>Zero length data messages</strong></td>
</tr>
<tr>
<td></td>
<td>Frees the message and does not pass downstream.</td>
</tr>
<tr>
<td></td>
<td><strong>Messages with control portions</strong></td>
</tr>
<tr>
<td></td>
<td>Generates an error, fails any further system calls, and sets the <code>errno</code> global variable to <code>EPROTO</code>.</td>
</tr>
<tr>
<td>read</td>
<td>Takes the following actions on data that originated from the transport protocol provider:</td>
</tr>
<tr>
<td></td>
<td><strong>Messages with no control portions</strong></td>
</tr>
<tr>
<td></td>
<td>Passes the message on upstream.</td>
</tr>
<tr>
<td></td>
<td><strong>Zero length data messages</strong></td>
</tr>
<tr>
<td></td>
<td>Frees the message and does not pass upstream.</td>
</tr>
<tr>
<td></td>
<td>Messages with control portions will produce the following actions:</td>
</tr>
<tr>
<td></td>
<td>• Messages that represent expedited data generate an error. All further calls associated with the stream fail with the <code>errno</code> global variable set to <code>EPROTO</code>.</td>
</tr>
<tr>
<td></td>
<td>• Any data messages with control portions have the control portions removed from the message prior to passing the message to the upstream neighbor.</td>
</tr>
<tr>
<td></td>
<td>• Messages that represent an orderly release indication from the transport provider generate a zero length data message, indicating the end of file, which is sent to the reader of the stream. The orderly release message itself is freed by the module.</td>
</tr>
<tr>
<td></td>
<td>• Messages that represent an abortive disconnect indication from the transport provider cause all further <code>write</code> and <code>putmsg</code> calls to fail with the <code>errno</code> global variable set to <code>ENXIO</code>. All further <code>read</code> and <code>getmsg</code> calls return zero length data (indicating end of file) once all previous data has been read.</td>
</tr>
<tr>
<td></td>
<td>• With the exception of the above rules, all other messages with control portions generate an error, and all further system calls associated with the stream fail with the <code>errno</code> global variable set to <code>EPROTO</code>.</td>
</tr>
</tbody>
</table>
**pop**
Sends an orderly release request to the remote side of the transport connection if an orderly release indication has been previously received.

**Related Information**
The `timod` module.

The `streamio` operations.

The `read` subroutine, `write` subroutine.

The `getmsg` system call, `putmsg` system call.

---

**unbufcall Utility**

**Purpose**
Cancels a `bufcall` request.

**Syntax**
```c
void unbufcall(id);
```

**Description**
The `unbufcall` utility cancels a `bufcall` request.

This utility is part of STREAMS Kernel Extensions.

**Parameters**
`id` Identifies an event in the `bufcall` request.

**Related Information**
The `bufcall` utility.

List of Streams Programming References and Understanding STREAMS Messages in AIX 5L Version 5.3 Communications Programming Concepts.

---

**unlinkb Utility**

**Purpose**
Removes a message block from the head of a message.
Syntax

\[
\begin{align*}
\text{mblk_t} & \ast \\
\text{unlinkb}(\text{bp}) \\
\text{register mblk_t} & \ast \text{bp};
\end{align*}
\]

Description

The `unlinkb` utility removes the first message block pointed to by the `bp` parameter and returns a pointer to the head of the resulting message. The `unlinkb` utility returns a null pointer if there are no more message blocks in the message.

This utility is part of STREAMS Kernel Extensions.

Parameters

`bp` Specifies which message block to unlink.

Related Information

The `linkb` utility.

List of Streams Programming References and Understanding STREAMS Messages in *AIX 5L Version 5.3 Communications Programming Concepts.*

untimout Utility

Purpose

Cancels a pending timeout request.

Syntax

\[
\begin{align*}
\text{int} & \\
\text{untimout}(\text{id}) \\
\text{int} & \text{id};
\end{align*}
\]

Description

The `untimout` utility cancels the specific request made with the `timeout` utility.

This utility is part of STREAMS Kernel Extensions.

Note: This utility must not be confused with the kernel service of the same name in the `libsys.a` library. STREAMS modules and drivers inherently use this version, not the `libsys.a` library version. No special action is required to use this version in the STREAMS environment.

Parameters

`id` Specifies the identifier returned from the corresponding timeout request.

Execution Environment

The `untimout` utility can be called from either the process or interrupt environment.
unweldq Utility

Purpose
Removes a previously established weld connection between STREAMS queues.

Syntax
```c
#include <sys/stream.h>

int unweldq ( q1, q2, q3, q4, func, arg, protect_q)
queue_t *q1;
queue_t *q2;
queue_t *q3;
queue_t *q4;
weld_fcn_t func;
weld_arg_t arg;
queue_t *protect_q;
```

Description
The `unweldq` utility removes a weld connection previously established with the `weld` utility between two STREAMS queues (q1 and q2). The `unweldq` utility can be used to unweld two pairs of queues in one call (q1 and q2, q3 and q4).

The unwelding operation is performed by changing the first queue’s `q_next` pointer so that it does not point to any queue. The `unweldq` utility does not actually perform the operation. Instead, it creates an unwelding request which STREAMS performs asynchronously. STREAMS acquires the appropriate synchronization queues before performing the operation.

Callers that need to know when the unwelding operation has actually taken place should specify a callback function (`func` parameter) when calling the `unweldq` utility. If the caller also specifies a synchronization queue (`protect_q` parameter), STREAMS acquires the synchronization associated with that queue when calling `func`. If the callback function is not a protected STREAMS utility, such as the `qenable` utility, the caller should always specify a `protect_q` parameter. The caller can also use this parameter to synchronize callbacks with protected STREAMS utilities.

Note: The `stream.h` header file must be the last included header file of each source file using the stream library.

Parameters
- `q1`: Specifies the queue whose `q_next` pointer must be nulled.
- `q2`: Specifies the queue that will be unwelded to `q1`.
- `q3`: Specifies the second queue whose `q_next` pointer must be nulled. If the `unweldq` utility is used to unweld only one pair of queues, this parameter should be set to `NULL`.
- `q4`: Specifies the queue that will be unwelded to `q3`.
- `func`: Specifies an optional callback function that will execute when the unwelding operation has completed.
- `arg`: Specifies the parameter for `func`.
Specifies an optional synchronization queue that protects `func`.

**Return Values**
Upon successful completion, 0 (zero) is returned. Otherwise, an error code is returned.

**Error Codes**
The `unweldq` utility fails if the following is true:

<table>
<thead>
<tr>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EAGAIN</td>
<td>The weld record could not be allocated. The caller may try again.</td>
</tr>
<tr>
<td>EINVAL</td>
<td>One or more parameters are not valid.</td>
</tr>
<tr>
<td>ENXIO</td>
<td>The weld mechanism is not installed.</td>
</tr>
</tbody>
</table>

**Related Information**

- [List of Streams Programming References](in AIX 5L Version 5.3 Communications Programming Concepts).
- [STREAMS Overview](in AIX 5L Version 5.3 Communications Programming Concepts).
- [Welding Mechanism](in AIX 5L Version 5.3 Communications Programming Concepts).

The `weldq` utility.

---

**wantio Utility**

**Purpose**
Register direct I/O entry points with the stream head.

**Syntax**

```c
#include <sys/stream.h>
int wantio(queue_t *q, struct wantio *w)
```

**Parameters**

<table>
<thead>
<tr>
<th>q</th>
<th>Pointer to the queue structure.</th>
</tr>
</thead>
<tbody>
<tr>
<td>w</td>
<td>Pointer to the wantio structure.</td>
</tr>
</tbody>
</table>

**Description**

The `wantio` STREAMS routine can be used by a STREAMS module or driver to register input/output (read/write/select) entry points with the stream head. The stream head then calls these entry points directly, by-passing all normal STREAMS processing, when an I/O request is detected. This service may be useful to increase STREAMS performance in cases where normal module processing is not required or where STREAMS processing is to be performed outside of this operating system.

STREAMS modules and drivers should precede a `wantio` call by sending a high priority M_LETSPLAY message upstream. The M_LETSPLAY message format is a message block containing an integer followed by a pointer to the write queue of the module or driver originating the M_LETSPLAY message. The integer counts the number of modules that can permit direct I/O. Each module passes this message to its neighbor after incrementing the count if direct I/O is possible. When this message reaches the stream head, the stream head compares the count field with the number of modules and drivers in the stream. If the count is not equal to the number of modules, then a M_DONTPLAY message is sent downstream.
indicating direct I/O will not be permitted on the stream. If the count is equal, then queued messages are cleared by sending them downstream as M_BACKWASH messages. When all messages are cleared, then an M_BACKDONE message is sent downstream. This process starts at the stream head and is repeated in every module in the stream. Modules will wait to receive an M_BACKDONE message from upstream. Upon receipt of this message, the module will send all queued data downstream as M_BACKWASH messages. When all data is cleared, the module will send an M_BACKDONE message to its downstream neighbor indicating that all data has been cleared from the stream to this point. wantio registration is cleared from a stream by issuing a wantio call with a NULL pointer to the wantio structure.

Multiprocessor serialization is the responsibility of the driver or module requesting direct I/O. The stream head acquires no STREAMS locks before calling the wantio entry point.

Currently, the write entry point of the wantio structure is ignored.

**Return Values**

Returns 0 always.

**Related Information**

The [wantmsg utility](#).

The queue and wantio structures in /usr/include/sys/stream.h.

---

**wantmsg Utility**

**Purpose**

Allows a STREAMS message to bypass a STREAMS module if the module is not interested in the message.

**Syntax**

```c
int wantmsg(q, f)
queue_t *q;
int (*f)();
```

**Description**

The wantmsg utility allows a STREAMS message to bypass a STREAMS module if the module is not interested in the message, resulting in performance improvements.

The module registers filter functions with the read and write queues of the module with the wantmsg utility. A filter function takes as input a message pointer and returns 1 if the respective queue is interested in receiving the message. Otherwise it returns 0. The putnext and qreply subroutines call a queue’s filter function before putting a message on that queue. If the filter function returns 1, then putnext or qreply put the message on that queue. Otherwise, putnext or qreply bypass the module by putting the message on the next module’s queue.

The filter functions must be defined so that a message bypasses a module only when the module does not need to see the message.

The wantmsg utility cannot be used if the module has a service routine associated with the queue specified by the q parameter. If wantmsg is called for a module that has a service routine associated with q, wantmsg returns a value of 0 without registering the filter function with q.

This utility is part of STREAMS Kernel Extensions.
Parameters

$q$ Specifies the read or write queue to which the filter function is to be registered.
$f$ Specifies the module’s filter function that is called at the putnext or qreply time.

Return Values

Upon successful completion, the wantmsg utility returns a 1, indicating that the filter function specified by the $f$ parameter has been registered for the queue specified by the $q$ parameter. In this case, the filter function is called from putnext or qreply. The wantmsg utility returns a value of 0 if the module has a service routine associated with the queue $q$, indicating that the filter function is not registered with $q$.

Example

wantmsg($q$, tioc_is_r_interesting);
    wantmsg(WR($q$), tioc_is_w_interesting);

/*
 * read queue filter function.
 * queue is only interested in IOCNAK, IOCACK, and
 * CTL messages.
 */
static int
tioc_is_r_interesting(mblk_t *mp)
{
    if (mp->b_datap->db_type == M_DATA)
        /* fast path for data messages */
        return 0;
    else if (mp->b_datap->db_type == M_IOCTL
             || mp->b_datap->db_type == M_IOCNAK
             || mp->b_datap->db_type == M_IOCACK
             || mp->b_datap->db_type == M_CTL)
        return 1;
    else
        return 0;
}

/*
 * write queue filter function.
 * queue is only interested in IOCTL and IOCDATA messages.
 */
static int
tioc_is_w_interesting(mblk_t *mp)
{
    if (mp->b_datap->db_type == M_DATA)
        /* fast path for data messages */
        return 0;
    else if (mp->b_datap->db_type == M_IOCTL
             || mp->b_datap->db_type == M_IOCDATA)
        return 1;
    else
        return 0;
}

Related Information

The putnext utility, the qreply utility.

List of Streams Programming References  STREAMS Messages  in AIX 5L Version 5.3 Communications Programming Concepts.
**weldq Utility**

**Purpose**
Establishes an uni-directional connection between STREAMS queues.

**Syntax**
```c
#include <sys/stream.h>
int weldq (q1, q2, q3, q4, func, arg, protect_q);
```

`queue_t *q1;`
`queue_t *q2;`
`queue_t *q3;`
`queue_t *q4;`
`weld_fcn_t func;`
`weld_arg_t arg;`
`queue_t *protect_q;`

**Description**
The `weldq` utility establishes an uni-directional connection (weld connection) between two STREAMS queues (`q1` and `q2`). The `weldq` utility can be used to weld two pairs of queues in one call (`q1` and `q2`, `q3` and `q4`).

The welding operation is performed by changing the first queue’s `q_next` pointer to point to the second queue. The `weldq` utility does not actually perform the operation. Instead, it creates a welding request which STREAMS performs asynchronously. STREAMS acquires the appropriate synchronization queues before performing the operation.

Callers that need to know when the welding operation has actually taken place should specify a callback function (`func` parameter) when calling the `weldq` utility. If the caller also specifies a synchronization queue (`protect_q` parameter), STREAMS acquires the synchronization associated with that queue when calling `func`. If the callback function is not a protected STREAMS utility, such as the `qenable` utility, the caller should always specify a `protect_q` parameter. The caller can also use this parameter to synchronize callbacks with protected STREAMS utilities.

This utility is part of STREAMS Kernel Extensions.

**Note:** The `stream.h` header file must be the last included header file of each source file using the stream library.

**Parameters**

- **q1** Specifies the queue whose `q_next` pointer must be modified.
- **q2** Specifies the queue that will be welded to `q1`.
- **q3** Specifies the second queue whose `q_next` pointer must be modified. If the `weldq` utility is used to weld only one pair of queues, this parameter should be set to `NULL`.
- **q4** Specifies the queue that will be welded to `q3`.
- **func** Specifies an optional callback function that will execute when the welding operation has completed.
- **arg** Specifies the parameter for `func`.
- **protect_q** Specifies an optional synchronization queue that protects `func`.

**Return Values**
Upon successful completion, 0 (zero) is returned. Otherwise, an error code is returned.
Error Codes
The **weldq** utility fails if the following is true:

<table>
<thead>
<tr>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EAGAIN</td>
<td>The weld record could not be allocated. The caller may try again.</td>
</tr>
<tr>
<td>EINVAL</td>
<td>One or more parameters are not valid.</td>
</tr>
<tr>
<td>ENXIO</td>
<td>The weld mechanism is not installed.</td>
</tr>
</tbody>
</table>

Related Information

- [List of Streams Programming References](#) in *AIX 5L Version 5.3 Communications Programming Concepts*.
- [STREAMS Overview](#) in *AIX 5L Version 5.3 Communications Programming Concepts*.
- [Welding Mechanism](#) in *AIX 5L Version 5.3 Communications Programming Concepts*.

The **unweldq** utility.

WR Utility

Purpose
Retrieves a pointer to the write queue.

Syntax

```c
#define WR(q) (((q)+1))
```

Description
The **WR** utility accepts a read queue pointer, the `q` parameter, as an argument and returns a pointer to the write queue for the same module.

This utility is part of STREAMS Kernel Extensions.

Parameters

- `q` Specifies the read queue.

Related Information

The **OTHERQ** utility, **RD** utility.

- [List of Streams Programming References](#) and [Understanding STREAMS Messages](#) in *AIX 5L Version 5.3 Communications Programming Concepts*.

xtiso STREAMS Driver

Purpose
Provides access to sockets-based protocols to STREAMS applications.
Description

The **xtiso** driver (X/Open Transport Interface (XTI) over Sockets) is a STREAMS-based pseudo-driver that provides a Transport Layer Interface (TLI) to the socket-based protocols. The only supported use of the **xtiso** driver is by the TLI and XTI libraries.

The TLI and XTI specifications do not describe the name of the transport provider and how to address local and remote hosts, two important items required for use.

The **xtiso** driver supports most of the protocols available through the socket interface. Each protocol has a /dev entry, which must be used as the name parameter in the `t_open` subroutine. The currently supported names (as configured by the `strload` subroutine) are:

<table>
<thead>
<tr>
<th>Name</th>
<th>Socket Equivalent</th>
</tr>
</thead>
<tbody>
<tr>
<td>/dev/xti/unixdg</td>
<td>AF_UNIX, SOCK_DGRAM</td>
</tr>
<tr>
<td>/dev/xti/unixst</td>
<td>AF_UNIX, SOCK_STREAM</td>
</tr>
<tr>
<td>/dev/xti/udp</td>
<td>AF_INET, SOCK_DGRAM</td>
</tr>
<tr>
<td>/dev/xti/tcp</td>
<td>AF_INET, SOCK_STREAM</td>
</tr>
</tbody>
</table>

Each of these protocols has a `sockaddr` structure that is used to specify addresses. These structures are also used by the TLI and XTI functions that require host addresses. The `netbuf` structure associated with the address for a particular function should refer to one of the `sockaddr` structure types. For instance, the TCP socket protocol uses a `sockaddr_in` structure; so a corresponding `netbuf` structure would be:

```c
struct netbuf addr;
struct sockaddr_in sin;
/* initialize sockaddr here */
sin.sin_family = AF_INET;
sin.sin_port = 0;
sin.sin_addr.s_addr = inet_addr("127.0.0.1");
addr.maxlen = sizeof(sin);
addr.len = sizeof(sin);
addr.buf = (char *)&sin;
```

The XTI Stream always consists of a Stream head and the transport interface module, `timod`. Depending on the transport provider specified by the application, `timod` accesses either the STREAMS-based protocol stack natively or a socket-based protocol through the pseudo-driver, **xtiso**.

The XTI library, `libxti.a` assumes a STREAMS-based transport provider. The routines of this library perform various operations for sending transport Provider Interface, TPI, messages down the XTI streams to the transport provider and receives them back.

The transport interface module, `timod`, is a STREAMS module that completes the translation of the TPI messages in the downstream and upstream directions.

The **xtiso** driver is a pseudo-driver that acts as the transport provider for socket-based communications. It interprets back and forth between the the TPI messages it receives from upstream and the socket interface.

AIX also provides the transport interface read/write module, `tirdwr`, which applications can push on to the XTI/TLI Stream for accessing the socket layer with standard UNIX read and write calls.

This driver is part of STREAMS Kernel Extensions.

Files

`/dev/xti/*` Contains names of supported protocols.
Related Information
The \texttt{strload} command.

The \texttt{t\_bind} subroutine for Transport Layer Interface, \texttt{t\_connect} subroutine for Transport Layer Interface, \texttt{t\_open} subroutine for Transport Layer Interface.

The \texttt{t\_bind} subroutine for X/Open Transport Layer Interface, \texttt{t\_connect} subroutine for X/Open Transport Layer Interface, \texttt{t\_open} subroutine for X/Open Transport Layer Interface.

\textit{Internet Transport-Level Protocols} in \textit{Networks and communication management}.


\textit{Understanding STREAMS Drivers and Modules} in \textit{AIX 5L Version 5.3 Communications Programming Concepts}.

\section*{t\_accept Subroutine for X/Open Transport Interface}

\subsection*{Purpose}
Accept a connect request.

\subsection*{Library}
\texttt{X/Open Transport Interface Library (libxti.a)}

\subsection*{Syntax}
\begin{verbatim}
#include <xti.h>
int t_accept (fd, resfd, call)
int fd;
int resfd;
const struct t\_call *call;
\end{verbatim}

\subsection*{Description}
The \texttt{t\_accept} subroutine is issued by a transport user to accept a command request. A transport user may accept a connection on either the same local transport endpoint or on an endpoint different than the one on which the connect indication arrived.

Before the connection can be accepted on the same endpoint, the user must have responded to any previous connect indications received on that transport endpoint via the \texttt{t\_accept} subroutine or the \texttt{t\_snddis} subroutine. Otherwise, the \texttt{t\_accept} subroutine will fail and set \texttt{t\_errno} to \texttt{T\_INDOUT}.

If a different transport endpoint is specified, the user may or may not choose to bind the endpoint before the \texttt{t\_accept} subroutine is issued. If the endpoint is not bound prior to the \texttt{t\_accept} subroutine, the transport provider will automatically bind the endpoint to the same protocol address specified in the \texttt{fd} parameter. If the transport user chooses to bind the endpoint, it must be bound to a protocol address with a \texttt{qlen} field of zero (see the \texttt{t\_bind} subroutine) and must be in the \texttt{T\_IDLE} state before the \texttt{t\_accept} subroutine is issued.

The call to the \texttt{t\_accept} subroutine fails with \texttt{t\_errno} set to \texttt{T\_LOOK} if there are indications (for example, connect or disconnect) waiting to be received on the endpoint specified by the \texttt{fd} parameter.

The value specified in the \texttt{udata} field enables the called transport user to send user data to the caller. The amount of user data sent must not exceed the limits supported by the transport provider. This limit is
specified in the connect field of the t_info structure of the t_open or t_getinfo subroutines. If the len field of udata is zero, no data is sent to the caller. All the maxlen fields are meaningless.

When the user does not indicate any option, it is assumed that the connection is to be accepted unconditionally. The transport provider may choose options other than the defaults to ensure that the connection is accepted successfully.

There may be transport provider-specific restrictions on address binding. See Appendix A, ISO Transport Protocol Information and Appendix B, Internet Protocol-specific Information.

Some transport providers do not differentiate between a connect indication and the connection itself. If the connection has already been established after a successful return of the t_listen subroutine, the t_accept subroutine will assign the existing connection to the transport endpoint specified by resfd (see Appendix B, Internet Protocol-specific Information).

Parameters

fd Identifies the local transport endpoint where the connect indication arrived.
resfd Specifies the local transport endpoint where the connection is to be established.
call Contains information required by the transport provider to complete the connection. The call parameter points to a t_call structure which contains the following members:

```c
struct netbuf addr;
struct netbuf opt;
struct netbuf udata;
int sequence;
```

The fields within the structure have the following meanings:

- `addr` Specifies the protocol address of the calling transport user. The address of the caller may be null (length zero). When this field is not null, the field may be optionally checked by the X/Open Transport Interface.
- `opt` Indicates any options associated with the connection.
- `udata` Points to any user data to be returned to the caller.
- `sequence` Specifies the value returned by the t_listen subroutine which uniquely associates the response with a previously received connect indication.

Valid States

- `fd`: T_INCON
- `resfd`: T_IDLE

Return Values

0 Successful completion.
-1 Unsuccessful completion, t_errno is set to indicate an error.

Error Codes

On failure, t_errno is set to one of the following:

<table>
<thead>
<tr>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>TACCES</td>
<td>The user does not have permission to accept a connection on the responding transport endpoint or to use the specified options.</td>
</tr>
<tr>
<td>TBADADDR</td>
<td>The specified protocol address was in an incorrect format or contained illegal information.</td>
</tr>
<tr>
<td>Value</td>
<td>Description</td>
</tr>
<tr>
<td>-------------</td>
<td>-----------------------------------------------------------------------------</td>
</tr>
<tr>
<td>TBADDATA</td>
<td>The amount of user data specified was not within the bounds allowed by the transport provider.</td>
</tr>
<tr>
<td>TBADF</td>
<td>The file descriptor ( fd ) or ( resfd ) does not refer to a transport endpoint.</td>
</tr>
<tr>
<td>TBADOPT</td>
<td>The specified options were in an incorrect format or contained illegal information.</td>
</tr>
<tr>
<td>TBADSEQ</td>
<td>An invalid sequence number was specified.</td>
</tr>
<tr>
<td>TINDOUT</td>
<td>The subroutine was called with the same endpoint, but there are outstanding connection indications on the endpoint. Those other connection indications must be handled either by rejecting them via the \texttt{t_snddis} subroutine or accepting them on a different endpoint via the \texttt{t_accept} subroutine.</td>
</tr>
<tr>
<td>TLOOK</td>
<td>An asynchronous event has occurred on the transport endpoint referenced by ( fd ) and requires immediate attention.</td>
</tr>
<tr>
<td>TNOTSUPPORT</td>
<td>This subroutine is not supported by the underlying transport provider.</td>
</tr>
<tr>
<td>TOUTSTATE</td>
<td>The subroutine was called in the wrong sequence on the transport endpoint referenced by ( fd ), or the transport endpoint referred to by ( resfd ) is not in the appropriate state.</td>
</tr>
<tr>
<td>TPROTO</td>
<td>This error indicates that a communication problem has been detected between X/Open Transport Interface and the transport provider for which there is no other suitable X/Open Transport Interface (\texttt{t_errno}).</td>
</tr>
<tr>
<td>TPROVMISMATCH</td>
<td>The file descriptors ( fd ) and ( resfd ) do not refer to the same transport provider.</td>
</tr>
<tr>
<td>TRESADDR</td>
<td>This transport provider requires both ( fd ) and ( resfd ) to be bound to the same address. This error results if they are not.</td>
</tr>
<tr>
<td>TRESQLEN</td>
<td>The endpoint referenced by ( resfd ) (where ( resfd ) is a different transport endpoint) was bound to a protocol address with a \textit{qlen} field value that is greater than zero.</td>
</tr>
<tr>
<td>TSYSERR</td>
<td>A system error has occurred during execution of this subroutine.</td>
</tr>
</tbody>
</table>

Related Information

The \texttt{t_connect} subroutine, \texttt{t_getstate} subroutine, \texttt{t_open} subroutine, \texttt{t_optmgmt} subroutine, \texttt{t_rcvconnect} subroutine.

\textbf{t_alloc Subroutine for X/Open Transport Interface}

\textbf{Purpose}

Allocate a library structure.

\textbf{Library}

X/Open Transport Interface Library (\textit{libxti.a})

\textbf{Syntax}

```c
#include <xti.h>

void *t_alloc (  
int \( fd \),  
int \texttt{struct_type},  
int \texttt{fields})
```

\textbf{Description}

The \texttt{t_alloc} subroutine dynamically allocates memory for the various transport function parameter structures. This subroutine allocates memory for the specified structure, and also allocates memory for buffers referenced by the structure.

Use of the \texttt{t_alloc} subroutine to allocate structures helps ensure the compatibility of user programs with future releases of the transport interface functions.
Parameters

*fd* Specifies the transport endpoint through which the newly allocated structure will be passed.

*struc_type* Specifies the structure to be allocated. The possible values are:

- **T_BIND**
  ```
  struct t_bind
  ```

- **T_CALL**
  ```
  struct t_call
  ```

- **T_OPTMGMT**
  ```
  struct t_optmgmt
  ```

- **T_DIS**
  ```
  struct t_discon
  ```

- **T_UNITDATA**
  ```
  struct t_unitdata
  ```

- **T_UDERROR**
  ```
  struct t_uderr
  ```

- **T_INFO**
  ```
  struct t_info
  ```

Each of these structures may subsequently be used as a parameter to one or more transport functions. Each of the above structures, except **T_INFO**, contains at least one field of the `struct netbuf` type. For each field of this type, the user may specify that the buffer for that field should be allocated as well. The length of the buffer allocated will be equal to or greater than the appropriate size as returned in the `info` parameter of the `t_open` or `t_getinfo` subroutines.

*fields* Specifies whether the buffer should be allocated for each field type. The `fields` parameter specifies which buffers to allocate, where the parameter is the bitwise-OR of any of the following:

- **T_ADDR**
  The `addr` field of the `t_bind`, `t_call`, `t_unitdata` or `t_uderr` structures.

- **T_OPT**
  The `opt` field of the `t_optmgmt`, `t_call`, `t_unitdata` or `t_uderr` structures.

- **T_UDATA**
  The `udata` field of the `t_call`, `t_discon` or `t_unitdata` structures.

- **T_ALL**
  All relevant fields of the given structure. Fields which are not supported by the transport provider specified by the `fd` parameter are not allocated.

For each relevant field specified in the `fields` parameter, the `t_alloc` subroutine allocates memory for the buffer associated with the field and initializes the `len` field to zero and initializes the `buf` pointer and `maxlen` field accordingly. Irrelevant or unknown values passed in fields are ignored. The length of the buffer allocated is based on the same size information returned to the user on a call to the `t_open` and `t_getinfo` subroutines. Thus, the `fd` parameter must refer to the transport endpoint through which the newly allocated structure is passed so that the appropriate size information is accessed. If the size value associated with any specified field is -1 or -2, (see the `t_open` or `t_getinfo` subroutines), the `t_alloc` subroutine is unable to determine the size of the buffer to allocate and fails, setting `t_errno` to `TSYSERR` and `errno` to `EINVAL`. For any field not specified in `fields`, `buf` will be set to the null pointer and `len` and `maxlen` will be set to zero.

Valid States

**ALL** - apart from T_UNINIT.

Return Values

On successful completion, the `t_alloc` subroutine returns a pointer to the newly allocated structure. On failure, a null pointer is returned.
Error Codes
On failure, _t_errno is set to one of the following:

<table>
<thead>
<tr>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>TBADF</td>
<td>The specified file descriptor does not refer to a transport endpoint.</td>
</tr>
<tr>
<td>TSYSERR</td>
<td>A system error has occurred during execution of this function.</td>
</tr>
<tr>
<td>TNOSTRUCTYPE</td>
<td>Unsupported structure type (struct_type) requested. This can include a request for a structure type which is inconsistent with the transport provider type specified, for example, connection-oriented or connectionless.</td>
</tr>
<tr>
<td>TPROTO</td>
<td>This error indicates that a communication problem has been detected between the X/Open Transport Interface and the transport provider for which there is no other suitable X/Open Transport Interface (_t_errno).</td>
</tr>
</tbody>
</table>

Related Information
The _t_free subroutine, _t_getinfo subroutine, _t_open subroutine.

t_bind Subroutine for X/Open Transport Interface

Purpose
Bind an address to a transport endpoint.

Library
X/Open Transport Interface Library (libxti.a)

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

int t_bind (fd, req, ret)
    int fd;
    const struct t_bind *req;
    struct t_bind *ret;
```

Description
The _t_bind subroutine associates a protocol address with the transport endpoint specified by the _fd parameter and activates that transport endpoint. In connection mode, the transport provider may begin enqueuing incoming connect indications or servicing a connection request on the transport endpoint. In connectionless mode, the transport user may send or receive data units through the transport endpoint.

The _req and _ret parameters point to a _t_bind structure containing the following members:
```c
struct netbuf addr;
unsigned qlen;
```

Within this structure, the fields have the following meaning:

<table>
<thead>
<tr>
<th>Field</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>_addr</td>
<td>Specifies a protocol address.</td>
</tr>
<tr>
<td>_qlen</td>
<td>Indicates the maximum number of outstanding connect indications.</td>
</tr>
</tbody>
</table>

If the requested address is not available, the _t_bind subroutine returns -1 with _t_errno set as appropriate. If no address is specified in the _req parameter, (that is, the _len field of the _addr field in the _req parameter is zero or the _req parameter is NULL), the transport provider assigns an appropriate address to be bound,
and returns that address in the `addr` field of the `ret` parameter. If the transport provider could not allocate an address, the `t_bind` subroutine fails with `t_errno` set to `TNOADDR`.

The `qlen` field has meaning only when initializing a connection-mode service. This field specifies the number of outstanding connect indications that the transport provider should support for the given transport endpoint. An outstanding connect indication is one that has been passed to the transport user by the transport provider but which has not been accepted or rejected. A `qlen` field value of greater than zero is only meaningful when issued by a passive transport user that expects other users to call it. The value of the `qlen` field is negotiated by the transport provider and may be changed if the transport provider cannot support the specified number of outstanding connect indications. However, this value of the `qlen` field is never negotiated from a requested value greater than zero to zero. This is a requirement on transport providers. See “Implementation Specifics” for more information. On return, the `qlen` field in the `ret` parameter contains the negotiated value.

The requirement that the value of the `qlen` field never be negotiated from a requested value greater than zero to zero implies that transport providers, rather than the X/Open Transport Interface implementation itself, accept this restriction.

A transport provider may not allow an explicit binding of more than one transport endpoint to the same protocol address, although it allows more than one connection to be accepted for the same protocol address. To ensure portability, it is, therefore, recommended not to bind transport endpoints that are used as responding endpoints, (those specified in the `resfd` parameter), in a call to the `t_accept` subroutine, if the responding address is to be the same as the called address.

### Parameters

- **`fd`** Specifies the transport endpoint. If the `fd` parameter refers to a connection-mode service, this function allows more than one transport endpoint to be bound to the same protocol address. However, the transport provider must also support this capability and it is not possible to bind more than one protocol address to the same transport endpoint. If a user binds more than one transport endpoint to the same protocol address, only one endpoint can be used to listen for connect indications associated with that protocol address. In other words, only one `t_bind` for a given protocol address may specify a `qlen` field value greater than zero. In this way, the transport provider can identify which transport endpoint should be notified of an incoming connect indication. If a user attempts to bind a protocol address to a second transport endpoint with a `qlen` field value greater than zero, `t_bind` will return `-1` and set `t_errno` to `TADDRBUSY`. When a user accepts a connection on the transport endpoint that is being used as the listening endpoint, the bound protocol address will be found to be busy for the duration of the connection, until a `t_unbind` or `t_close` call has been issued. No other transport endpoints may be bound for listening on that same protocol address while that initial listening endpoint is active (in the data transfer phase or in the `T_IDLE` state). This will prevent more than one transport endpoint bound to the same protocol address from accepting connect indications.

If the `fd` parameter refers to a connectionless-mode service, only one endpoint may be associated with a protocol address. If a user attempts to bind a second transport endpoint to an already bound protocol address, `t_bind` will return `-1` and set `t_errno` to `TADDRBUSY`.


req  Specifies the address to be bound to the given transport endpoint. The \textit{req} parameter is used to request that an address, represented by the \texttt{netbuf} structure, be bound to the given transport endpoint. The \texttt{netbuf} structure is described in the \texttt{xti.h} file. In the \textit{req} parameter, the \texttt{netbuf} structure \textit{addr} fields have the following meanings:

\begin{itemize}
  \item \texttt{buf}\quad Points to the address buffer.
  \item \texttt{len}\quad Specifies the number of bytes in the address.
  \item \texttt{maxlen}\quad Has no meaning for the \textit{req} parameter.
\end{itemize}

The \textit{req} parameter may be a null pointer if the user does not specify an address to be bound. Here, the value of the \texttt{qlen} field is assumed to be zero, and the transport provider assigns an address to the transport endpoint. Similarly, the \textit{ret} parameter may be a null pointer if the user does not care what address was bound by the provider and is not interested in the negotiated value of the \texttt{qlen} field. It is valid to set the \textit{req} and \textit{ret} parameters to the null pointer for the same call, in which case the provider chooses the address to bind to the transport endpoint and does not return that information to the user.

\textit{ret}\quad Specifies the maximum size of the address buffer. On return, the \textit{ret} parameter contains the address that the transport provider actually bound to the transport endpoint; this is the same as the address specified by the user in the \textit{req} parameter. In the \textit{ret} parameter, the \texttt{netbuf} structure fields have the following meanings:

\begin{itemize}
  \item \texttt{buf}\quad Points to the buffer where the address is to be placed. On return, this points to the bound address.
  \item \texttt{len}\quad Specifies the number of bytes in the bound address on return.
  \item \texttt{maxlen}\quad Specifies the the maximum size of the address buffer. If the value of the \texttt{maxlen} field is not large enough to hold the returned address, an error will result.
\end{itemize}

Valid States

\begin{itemize}
  \item \texttt{T\_UNBIND}.
\end{itemize}

Return Values

\begin{itemize}
  \item 0 \quad Successful completion.
  \item -1 \quad \texttt{t\_errno} is set to indicate an error.
\end{itemize}

Error Codes

On failure, \texttt{t\_errno} is set to one of the following:

\begin{itemize}
  \item Value  \quad Description
    \item \texttt{TACCESS} \quad The user does not have permission to use the specified address.
    \item \texttt{TADDRBUSY} \quad The requested address is in use.
    \item \texttt{TBADADDR} \quad The specified protocol address was in an incorrect format or contained illegal information.
    \item \texttt{TBADF} \quad The specified file descriptor does not refer to a transport endpoint.
    \item \texttt{TBUFOVLW} \quad The number of bytes allowed for an incoming argument (\texttt{maxlen}) is greater than 0 but not sufficient to store the value of that argument. The provider's state will change to \texttt{T\_IDLE} and the information to be returned in \textit{ret} will be discarded.
    \item \texttt{TNOADDR} \quad The transport provider could not allocate an address.
    \item \texttt{TOUTSTATE} \quad The function was issued in the wrong sequence.
    \item \texttt{TPROTO} \quad This error indicates that a communication problem has been detected between the X/Open Transport Interface and the transport provider for which there is no other suitable X/Open Transport Interface (\texttt{t\_errno}).
    \item \texttt{TSYSERR} \quad A system error has occurred during execution of this function.
\end{itemize}

Related Information

The \texttt{t\_alloc} subroutine, \texttt{t\_close} subroutine, \texttt{t\_open} subroutine, \texttt{t\_optmgmt} subroutine, \texttt{t\_unbind} subroutine.
**t_close Subroutine for X/Open Transport Interface**

**Purpose**
Close a transport endpoint.

**Library**
X/Open Transport Interface Library (libxti.a)

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

int t_close (fd)
int fd;
```

**Description**
The `t_close` subroutine informs the transport provider that the user is finished with the transport endpoint specified by the `fd` parameter and frees any local library resources associated with the endpoint. In addition, the `t_close` subroutine closes the file associated with the transport endpoint.

The `t_close` subroutine should be called from the T_UNBND state (see the `t_getstate` subroutine). However, this subroutine does not check state information, so it may be called from any state to close a transport endpoint. If this occurs, the local library resources associated with the endpoint will be freed automatically. In addition, the `close` subroutine is issued for that file descriptor. The `close` subroutine is abortive if there are no other descriptors in this process or if there are no other descriptors in another process which references the transport endpoint, and in this case, will break any transport connection that may be associated with that endpoint.

A `t_close` subroutine issued on a connection endpoint may cause data previously sent, or data not yet received, to be lost. It is the responsibility of the transport user to ensure that data is received by the remote peer.

**Parameter**

`fd` Specfies the transport endpoint to be closed.

**Valid States**
ALL - apart from T_UNINIT.

**Return Values**

0 Successful completion.
-1 `t_errno` is set to indicate an error.

**Errors**
On failure, `t_errno` is set to one of the following:

<table>
<thead>
<tr>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>TBADF</td>
<td>The specified file descriptor does not refer to a transport endpoint.</td>
</tr>
<tr>
<td>TPROTO</td>
<td>This error indicates that a communication problem has been detected between</td>
</tr>
<tr>
<td></td>
<td>the X/Open Transport Interface and the transport provider for which there is</td>
</tr>
<tr>
<td></td>
<td>no other suitable X/Open Transport Interface (t_errno).</td>
</tr>
</tbody>
</table>
Related Information
The `t_getstate` subroutine, `t_open` subroutine, `t_unbind` subroutine.

**t_connect Subroutine for X/Open Transport Interface**

**Purpose**
Establish a connection with another transport user.

**Library**
X/Open Transport Interface Library (`libxti.a`)

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

int t_connect (fd, sndcall, rcvcall)
    int fd;
    const struct t_call *sndcall;
    struct t_call *rcvcall;
```

**Description**
The `t_connect` subroutine enables a transport user to request a connection to the specified destination transport user. This subroutine can only be issued in the `T_IDLE` state.

The `sndcall` and `rcvcall` parameters both point to a `t_call` structure which contains the following members:
```c
struct netbuf addr;
struct netbuf opt;
struct netbuf udata;
int sequence;
```

In the `sndcall` parameter, the fields of the structure have the following meanings:

<table>
<thead>
<tr>
<th>Field</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>addr</code></td>
<td>Specifies the protocol address of the destination transport user.</td>
</tr>
<tr>
<td><code>opt</code></td>
<td>Presents any protocol-specific information that might be needed by the transport provider.</td>
</tr>
<tr>
<td><code>sequence</code></td>
<td>Has no meaning for this subroutine.</td>
</tr>
<tr>
<td><code>udata</code></td>
<td>Points to optional user data that may be passed to the destination transport user during connection establishment.</td>
</tr>
</tbody>
</table>

On return, the fields of the structure pointed to by the `rcvcall` parameter have the following meanings:

<table>
<thead>
<tr>
<th>Field</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>addr</code></td>
<td>Specifies the protocol address associated with the responding transport endpoint.</td>
</tr>
<tr>
<td><code>opt</code></td>
<td>Represents any protocol-specific information associated with the connection.</td>
</tr>
<tr>
<td><code>sequence</code></td>
<td>Has no meaning for this subroutine.</td>
</tr>
<tr>
<td><code>udata</code></td>
<td>Points to optional user data that may be returned by the destination transport user during connection establishment.</td>
</tr>
</tbody>
</table>

The `opt` field permits users to define the options that may be passed to the transport provider. These options are specific to the underlying protocol of the transport provider and are described for ISO and TCP protocols in Appendix A, ISO Transport Protocol Information, Appendix B, Internet Protocol-specific Information and Appendix F, Headers and Definitions. The user may choose not to negotiate protocol options by setting the `len` field of `opt` to zero. In this case, the provider may use default options.
If used, the value of the opt.buf field of the sndcall parameter netbuf structure must point to a buffer with the corresponding options; the maxlen and buf values of the addr and opt fields of the rcvcall parameter netbuf structure must be set before the call.

The udata field of the structure enables the caller to pass user data to the destination transport user and receive user data from the destination user during connection establishment. However, the amount of user data must not exceed the limits supported by the transport provider as returned in the connect field of the info parameter of the t_open or t_getinfo subroutines. If the value of udata.len field is zero in the sndcall parameter netbuf structure, no data will be sent to the destination transport user.

On return, the addr, opt, and udata fields of rcvcall are updated to reflect values associated with the connection. Thus, the maxlen value of each field must be set before issuing this subroutine to indicate the maximum size of the buffer for each. However, the value of the rcvcall parameter may be a null pointer, in which case no information is given to the user on return from the t_connect subroutine.

By default, the t_connect subroutine executes in synchronous mode, and waits for the destination user’s response before returning control to the local user. A successful return (for example, return value of zero) indicates that the requested connection has been established. However, if O_NONBLOCK is set via the t_open subroutine or the fcntl parameter, the t_connect subroutine executes in asynchronous mode. In this case, the call will not wait for the remote user’s response, but returns control immediately to the local user and returns -1 with t_errno set to TNODATA to indicate that the connection has not yet been established. In this way, the subroutine initiates the connection establishment procedure by sending a connect request to the destination transport user. The t_rcvconnect subroutine is used in conjunction with the t_connect subroutine to determine the status of the requested connection.

When a synchronous t_connect call is interrupted by the arrival of a signal, the state of the corresponding transport endpoint is T_OUTCON, allowing a further call to either the t_rcvconnect, t_rcvdis or t_snddis subroutines.

Parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>fd</td>
<td>Identifies the local transport endpoint where communication will be established.</td>
</tr>
<tr>
<td>sndcall</td>
<td>Specifies information needed by the transport provider to establish a connection.</td>
</tr>
<tr>
<td>rcvcall</td>
<td>Specifies information associated with the newly established connection.</td>
</tr>
</tbody>
</table>

Valid States

T_IDLE.

Return Values

<table>
<thead>
<tr>
<th>Return Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Successful completion.</td>
</tr>
<tr>
<td>-1</td>
<td>t_errno is set to indicate an error.</td>
</tr>
</tbody>
</table>

Error Codes

On failure, t_errno is set to one of the following:

<table>
<thead>
<tr>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>TACCES</td>
<td>The user does not have permission to use the specified address or options.</td>
</tr>
<tr>
<td>TADDRBUSY</td>
<td>This transport provider does not support multiple connections with the same local and remote addresses. This error indicates that a connection already exists.</td>
</tr>
<tr>
<td>TBADADDR</td>
<td>The specified protocol address was in an incorrect format or contained illegal information.</td>
</tr>
<tr>
<td>TBADDATA</td>
<td>The amount of user data specified was not within the bounds allowed by the transport provider.</td>
</tr>
<tr>
<td>TBADF</td>
<td>The specified file descriptor does not refer to a transport endpoint.</td>
</tr>
</tbody>
</table>
### Value Description

<table>
<thead>
<tr>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>TBADOPT</td>
<td>The specified protocol options were in an incorrect format or contained illegal information.</td>
</tr>
<tr>
<td>TBUFOVFLW</td>
<td>The number of bytes allocated for an incoming parameter (maxlen) is greater than 0 but not sufficient to store the value of that argument. If executed in synchronous mode, the provider’s state, as seen by the user, changes to T_DAXAXFER, and the information to be returned in the rcvcall parameter is discarded.</td>
</tr>
<tr>
<td>TLOOK</td>
<td>An asynchronous event has occurred on this transport endpoint and requires immediate attention.</td>
</tr>
<tr>
<td>TNODATA</td>
<td>O_NONBLOCK was set, so the subroutine successfully initiated the connection establishment procedure, but did not wait for a response from the remote user.</td>
</tr>
<tr>
<td>TNOTSUPPORT</td>
<td>This subroutine is not supported by the underlying transport provider.</td>
</tr>
<tr>
<td>TOUTSTATE</td>
<td>The subroutine was issued in the wrong sequence.</td>
</tr>
<tr>
<td>TPROTO</td>
<td>This error indicates that a communication problem has been detected between the X/Open Transport Interface and the transport provider for which there is no other suitable X/Open Transport Interface (t_errno).</td>
</tr>
<tr>
<td>TSYSErr</td>
<td>A system error has occurred during execution of this subroutine.</td>
</tr>
</tbody>
</table>

### Related Information

The `t_accept` subroutine, `t_alloc` subroutine, `t_getinfo` subroutine, `t_open` subroutine, `t_optmgmt` subroutine, `t_rcvconnect` subroutine.

### t_error Subroutine for X/Open Transport Interface

#### Purpose

Produce error message.

#### Library

X/Open Transport Interface Library (libxti.a)

#### Syntax

```c
#include <xti.h>

int t_error (const char *errmsg)
```

#### Description

The `t_error` subroutine produces a language-dependent message on the standard error output which describes the last error encountered during a call to a transport subroutine.

If the `errmsg` parameter is not a null pointer and the character pointed to be the `errmsg` parameter is not the null character, the error message is written as follows: the string pointed to by the `errmsg` parameter followed by a colon and a space and a standard error message string for the current error defined in `t_errno`. If `t_errno` has a value different from `TSYSERR`, the standard error message string is followed by a newline character. If, however, `t_errno` is equal to `TSYSERR`, the `t_errno` string is followed by the standard error message string for the current error defined in the `errno` global variable followed by a newline.

The language for error message strings written by the `t_error` subroutine is implementation-defined. If it is in English, the error message string describing the value in `t_errno` is identical to the comments following the `t_errno` codes defined in the `xti.h` header file. The contents of the error message strings describing the value in the `errno` global variable are the same as those returned by the `strerror` subroutine with an parameter of `errno`. 

---

Chapter 3. Streams 389
The error number, \texttt{t_errno}, is only set when an error occurs and it is not cleared on successful calls.

\textbf{Parameter}

\texttt{errmsg} \quad Specifies a user-supplied error message that gives the context to the error.

\textbf{Valid States}

\texttt{ALL} - apart from \texttt{T_UNINIT}.

\textbf{Return Values}

Upon completion, a value of 0 is returned.

\textbf{Errors Codes}

No errors are defined for the \texttt{t_error} subroutine.

\textbf{Examples}

If a \texttt{t_connect} subroutine fails on transport endpoint \texttt{fd2} because a bad address was given, the following call might follow the failure:

\begin{verbatim}
t_error("t_connect failed on fd2");
\end{verbatim}

The diagnostic message to be printed would look like:

\begin{verbatim}
t_connect failed on fd2: incorrect addr format
\end{verbatim}

where \texttt{incorrect addr format} identifies the specific error that occurred, and \texttt{t_connect failed on fd2} tells the user which function failed on which transport endpoint.

\textbf{Related Information}

The \texttt{strerror} subroutine, \texttt{t_connect} subroutine.

\begin{center}
\textbf{t_free Subroutine for X/Open Transport Interface}
\end{center}

\textbf{Purpose}

Free a library structure.

\textbf{Library}

X/Open Transport Interface Library (\texttt{libxti.a})

\textbf{Syntax}

\begin{verbatim}
#include \texttt{<xti.h>}

int \texttt{t_free} (  
    void *\texttt{ptr};  
    int \texttt{struct_type})
\end{verbatim}

\textbf{Description}

The \texttt{t_free} subroutine frees memory previously allocated by the \texttt{t_alloc} subroutine. This subroutine frees memory for the specified structure and buffers referenced by the structure.

The \texttt{t_free} subroutine checks the \texttt{addr}, \texttt{opt}, and \texttt{udata} fields of the given structure, as appropriate, and frees the buffers pointed to by the \texttt{buf} field of the \texttt{netbuf} structure. If \texttt{buf} is a null pointer, the \texttt{t_free} subroutine continues to free memory for the specified structure and the buffers referenced by the structure.
subroutine does not attempt to free memory. After all buffers are free, the `t_free` subroutine frees the memory associated with the structure pointed to by the `ptr` parameter.

Undefined results occur if the `ptr` parameter or any of the `buf` pointers points to a block of memory that was not previously allocated by the `t_alloc` subroutine.

**Parameters**

- `ptr`  
  Points to one of the seven structure types described for the `t_alloc` subroutine.  

- `struct_type`  
  Identifies the type of the structure specified by the `ptr` parameter. The type can be one of the following:
  
  **T_BIND**
  
  ```c
  struct t_bind
  ```  

  **T_CALL**
  
  ```c
  struct t_call
  ```  

  **T_OPTMGMT**
  
  ```c
  struct t_optmgmt
  ```  

  **T_DIS**
  
  ```c
  struct t_discon
  ```  

  **T_UNITDATA**
  
  ```c
  struct t_unitdata
  ```  

  **T_UDERROR**
  
  ```c
  struct t_uderr
  ```  

  **T_INFO**
  
  ```c
  struct t_info
  ```

  Each of these structures may subsequently be used as a parameter to one or more transport functions.

**Valid States**

ALL - apart from T_UNINIT.

**Return Values**

- `0`  
  Successful completion.

- `-1`  
  `t_errno` is set to indicate an error.

**Error Codes**

On failure, `t_errno` is set to one of the following:

<table>
<thead>
<tr>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>TSYSERR</td>
<td>A system error has occurred during execution of this function.</td>
</tr>
<tr>
<td>TNOSTRUCTYPE</td>
<td>Unsupported <code>struct_type</code> parameter value requested.</td>
</tr>
<tr>
<td>TPROTO</td>
<td>This error indicates that a communication problem has been detected between the X/Open Transport Interface and the transport provider for which there is no other suitable X/Open Transport Interface (<code>t_errno</code>).</td>
</tr>
</tbody>
</table>

**Related Information**

The `t_alloc` subroutine.
t_getinfo Subroutine for X/Open Transport Interface

Purpose
Get protocol-specific service information.

Library
X/Open Transport Interface Library (libxti.a)

Syntax
#include <xti.h>

int t_getinfo (fd, info)
int fd;
struct t_info *info;

Description
The t_getinfo subroutine returns the current characteristics of the underlying transport protocol and/or transport connection associated with the file descriptor specified by the fd parameter. The pointer specified by the info parameter returns the same information returned by the t_open subroutine, although not necessarily precisely the same values. This subroutine enables a transport user to access this information during any phase of communication.

Parameters

fd Specifies the file descriptor.
info Points to a t_info structure which contains the following members:
  long addr; /* max size of the transport protocol */
  long options; /* max number of bytes of protocol-specific */
  long tsdu; /* max size of a transport service data */
  long etsdu; /* max size of an expedited transport */
  long connect; /* max amount of data allowed on connection */
  long discon; /* max amount of data allowed on t_snddis */
  long servtype; /* service type supported by the transport */
  long flags; /* other info about the transport provider */

The values of the fields have the following meanings:

<table>
<thead>
<tr>
<th>Field</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>addr</td>
<td>A value greater than zero indicates the maximum size of a transport protocol address and a value of -2 specifies that the transport provider does not provide user access to transport protocol addresses.</td>
</tr>
<tr>
<td>options</td>
<td>A value greater than zero indicates the maximum number of bytes of protocol-specific options supported by the provider, and a value of -2 specifies that the transport provider does not support options set by users.</td>
</tr>
<tr>
<td>tsdu</td>
<td>A value greater than zero specifies the maximum size of a transport service data unit (TSDU); a value of zero specifies that the transport provider does not support the concept of TSDU, although it does support the sending of a datastream with no logical boundaries preserved across a connection; a value of -1 specifies that there is no limit on the size of a TSDU; and a value of -2 specifies that the transfer of normal data is not supported by the transport provider.</td>
</tr>
</tbody>
</table>
Field | Description
--- | ---
etsdu | A value greater than zero specifies the maximum size of an expedited transport service data unit (ETSDU); a value of zero specifies that the transport provider does not support the concept of ETSDU, although it does support the sending of an expedited data stream with no logical boundaries preserved across a connection; a value of -1 specifies that there is no limit on the size of an ETSDU; and a value of -2 specifies that the transfer of expedited data is not supported by the transport provider. Note that the semantics of expedited data may be quite different for different transport providers (see Appendix A, ISO Transport Protocol Information and Appendix B, Internet Protocol-specific Information).
connect | A value greater than zero specifies the maximum amount of data that may be associated with connection establishment functions and a value of -2 specifies that the transport provider does not allow data to be sent with connection establishment functions.
discon | A value greater than zero specifies the maximum amount of data that may be associated with the t_snddis and t_rcvdis subroutines and a value of -2 specifies that the transport provider does not allow data to be sent with the abortive release functions.
servtype | This field specifies the service type supported by the transport provider on return. The possible values are:

T_COTS | The transport provider supports a connection-mode service but does not support the optional orderly release facility.

T_COTS_ORD | The transport provider supports a connection-mode service with the optional orderly release facility.

T_CLTS | The transport provider supports a connectionless-mode service. For this service type, the t_open subroutine will return -2 for etsdu, connect and discon.
flags | This is a bit field used to specify other information about the transport provider. If the T_SENDZERO bit is set in flags, this indicates that the underlying transport provider supports the sending of zero-length TSDUs. See Appendix A, ISO Transport Protocol Information for a discussion of the separate issue of zero-length fragments within a TSDU.

If a transport user is concerned with protocol independence, the above sizes may be accessed to determine how large the buffers must be to hold each piece of information. Alternatively, the t_alloc subroutine may be used to allocate these buffers. An error results if a transport user exceeds the allowed data size on any subroutine. The value of each field may change as a result of protocol option negotiation during connection establishment (the t_optmgmt call has no affect on the values returned by the t_getinfo subroutine). These values will only change from the values presented to the t_open subroutine after the endpoint enters the T_DATAXFER state.

Valid States
ALL - apart from T_UNINIT.

Return Values

0 | Successful completion.
-1 | t_errno is set to indicate an error.

Error Codes
On failure, t_errno is set to one of the following:

<table>
<thead>
<tr>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>TBADF</td>
<td>The specified file descriptor does not refer to a transport endpoint.</td>
</tr>
<tr>
<td>TSYSERR</td>
<td>A system error has occurred during execution of this subroutine.</td>
</tr>
</tbody>
</table>
Value  Description
TPROTO  This error indicates that a communication problem has been detected between the X/Open Transport Interface and the transport provider for which there is no other suitable X/Open Transport Interface (t_errno).

Related Information
The t_alloc subroutine, t_open subroutine.

_t_getprotaddr Subroutine for X/Open Transport Interface

Purpose
Get the protocol addresses.

Library
X/Open Transport Interface Library (libxti.a)

Syntax
#include <xti.h>

int _t_getprotaddr (fd, boundaddr, peeraddr)
int fd;
struct t_bind *boundaddr;
struct t_bind *peeraddr;

Description
The _t_getprotaddr subroutine returns local and remote protocol addresses currently associated with the transport endpoint specified by the fd parameter.

Parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>fd</td>
<td>Specifies the transport endpoint.</td>
</tr>
<tr>
<td>boundaddr</td>
<td>Specifies the local address to which the transport endpoint is to be bound. The boundaddr parameter has the following fields:</td>
</tr>
<tr>
<td>maxlen</td>
<td>Specifies the maximum size of the address buffer.</td>
</tr>
<tr>
<td>buf</td>
<td>Points to the buffer where the address is to be placed. On return, the buf field of boundaddr points to the address, if any, currently bound to fd.</td>
</tr>
<tr>
<td>len</td>
<td>Specifies the length of the address. If the transport endpoint is in the T_UNBND state, zero is returned in the len field of boundaddr.</td>
</tr>
<tr>
<td>peeraddr</td>
<td>Specifies the remote protocol address associated with the transport endpoint.</td>
</tr>
<tr>
<td>maxlen</td>
<td>Specifies the maximum size of the address buffer.</td>
</tr>
<tr>
<td>buf</td>
<td>Points to the address, if any, currently connected to fd.</td>
</tr>
<tr>
<td>len</td>
<td>Specifies the length of the address. If the transport endpoint is not in the T_DATAXFER state, zero is returned in the len field of peeraddr.</td>
</tr>
</tbody>
</table>

Valid States
ALL - apart from T_UNINIT.
Return Values

0  Successful completion.
-1  t_errno is set to indicate an error.

Error Codes

On failure, t_errno is set to one of the following:

<table>
<thead>
<tr>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>TBADF</td>
<td>The specified file descriptor does not refer to a transport endpoint.</td>
</tr>
<tr>
<td>TBUFVIEW</td>
<td>The number of bytes allocated for an incoming parameter (maxlen) is greater than 0 but not sufficient to store the value of that parameter.</td>
</tr>
<tr>
<td>TSYSERR</td>
<td>A system error has occurred during execution of this subroutine.</td>
</tr>
<tr>
<td>TPROTO</td>
<td>This error indicates that a communication problem has been detected between the X/Open Transport Interface and the transport provider for which there is no other suitable X/Open Transport Interface (t_errno).</td>
</tr>
</tbody>
</table>

Related Information

The t_bind subroutine.

**t_getstate Subroutine for X/Open Transport Interface**

**Purpose**
Get the current state.

**Library**
X/Open Transport Interface Library (libxti.a)

**Syntax**

```c
#include <xti.h>
int t_getstate (fd)
int fd;
```

**Description**

The t_getstate subroutine returns the current state of the provider associated with the transport endpoint specified by the fd parameter.

**Parameter**

*fd*  Specifies the transport endpoint.

**Valid States**

ALL - apart from T_UNINIT.

**Return Values**

0  Successful completion.
-1 \texttt{t_errno} is set to indicate an error. The current state is one of the following:

\begin{verbatim}
T_UNBND
  Unbound
T_IDLE
  Idle
T_OUTCON
  Outgoing connection pending
T_INCON
  Incoming connection pending
T_DATAFER
  Data transfer
T_OUTREL
  Outgoing orderly release (waiting for an orderly release indication)
T_INREL
  Incoming orderly release (waiting to send an orderly release request)
\end{verbatim}

If the provider is undergoing a state transition when the \texttt{t_getstate} subroutine is called, the subroutine will fail.

**Error Codes**

On failure, \texttt{t_errno} is set to one of the following:

<table>
<thead>
<tr>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>TBADF</td>
<td>The specified file descriptor does not refer to a transport endpoint.</td>
</tr>
<tr>
<td>TSTATECHNG</td>
<td>The transport provider is undergoing a transient state change.</td>
</tr>
<tr>
<td>TSYSErr</td>
<td>A system error has occurred during execution of this subroutine.</td>
</tr>
<tr>
<td>TPROTO</td>
<td>This error indicates that a communication problem has been detected between the X/Open Transport Interface and the transport provider for which there is no other suitable X/Open Transport Interface (t_errno).</td>
</tr>
</tbody>
</table>

**Related Information**

The \texttt{t_open} subroutine.

---

**t_listen Subroutine for X/Open Transport Interface**

**Purpose**

Listen for a connect indication.

**Library**

X/Open Transport Interface Library (\texttt{libxti.a})

**Syntax**

```c
#include <xti.h>
int t_listen (fd, call)
int fd;
struct t_call *call;
```

**Description**

The \texttt{t_listen} subroutine listens for a connect request from a calling transport user.
By default, the `t_listen` subroutine executes in synchronous mode and waits for a connect indication to arrive before returning to the user. However, if `O_NONBLOCK` is set via the `t_open` subroutine or with the `fcntl` subroutine (`F_SETFL`), the `t_listen` subroutine executes asynchronously, reducing to a poll for existing connect indications. If none are available, the subroutine returns -1 and sets `t_errno` to `TNODATA`.

Some transport providers do not differentiate between a connect indication and the connection itself. If this is the case, a successful return of `t_listen` indicates an existing connection (see Appendix B, Internet Protocol-specific Information).

**Parameters**

`fd` Identifies the local transport endpoint where connect indications arrive.

`call` Contains information describing the connect indication. The parameter `call` points to a `t_call` structure which contains the following members:

```c
struct netbuf addr;
struct netbuf opt;
struct netbuf udata;
int sequence;
```

In this structure, the fields have the following meanings:

- `addr` Returns the protocol address of the calling transport user. This address is in a format usable in future calls to the `t_connect` subroutine. Note, however that `t_connect` may fail for other reasons, for example, `TADDRBUSY`.

- `opt` Returns options associated with the connect request.

- `udata` Returns any user data sent by the caller on the connect request.

- `sequence` A number that uniquely identifies the returned connect indication. The value of `sequence` enables the user to listen for multiple connect indications before responding to any of them.

Since this subroutine returns values for the `addr`, `opt` and `udata` fields of the `call` parameter, the `maxlen` field of each must be set before issuing the `t_listen` subroutine to indicate the maximum size of the buffer for each.

**Valid States**

`T_IDLE`, `T_INCON`.

**Return Values**

- `0` Successful completion.
- `-1` `t_errno` is set to indicate an error.

**Error Codes**

On failure, `t_errno` is set to one of the following:

<table>
<thead>
<tr>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>TBADF</td>
<td>The specified file descriptor does not refer to a transport endpoint.</td>
</tr>
<tr>
<td>TBADQLEN</td>
<td>The <code>qlen</code> parameter of the endpoint referenced by the <code>fd</code> parameter is zero.</td>
</tr>
<tr>
<td>TBODATA</td>
<td><code>O_NONBLOCK</code> was set, but no connect indications had been queued.</td>
</tr>
</tbody>
</table>
t_look Subroutine for X/Open Transport Interface

Purpose
Look at the current event on a transport endpoint.

Library
X/Open Transport Interface Library (libxti.a)

Syntax
```c
#include <xti.h>
int t_look (fd)
int fd;
```

Description
The t_look subroutine returns the current event on the transport endpoint specified by the fd parameter. This subroutine enables a transport provider to notify a transport user of an asynchronous event when the user is calling subroutines in synchronous mode. Certain events require immediate notification of the user and are indicated by a specific error, TLOOK, on the current or next subroutine to be executed. Details on events which cause subroutines to fail, T_LOOK, may be found in Section 4.6, Events and TLOOK Error Indication.

This subroutine also enables a transport user to poll a transport endpoint periodically for asynchronous events.

Additional functionality is provided through the Event Management (EM) interface.

Parameter

- **fd** Specifies the transport endpoint.
Valid States
ALL - apart from T_UNINIT.

Return Values
Upon success, the t_look subroutine returns a value that indicates which of the allowable events has occurred, or returns zero if no event exists. One of the following events is returned:

<table>
<thead>
<tr>
<th>Event</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>T_LISTEN</td>
<td>Connection indication received.</td>
</tr>
<tr>
<td>T_CONNECT</td>
<td>Connect confirmation received.</td>
</tr>
<tr>
<td>T_DATA</td>
<td>Normal data received.</td>
</tr>
<tr>
<td>T_EXDATA</td>
<td>Expedited data received.</td>
</tr>
<tr>
<td>T_DISCONNECT</td>
<td>Disconnect received.</td>
</tr>
<tr>
<td>T_UDERR</td>
<td>Datagram error indication.</td>
</tr>
<tr>
<td>T_ORDREL</td>
<td>Orderly release indication.</td>
</tr>
<tr>
<td>T_GODATA</td>
<td>Flow control restrictions on normal data flow that led to a TFLOW error have been lifted. Normal data may be sent again.</td>
</tr>
<tr>
<td>T_GOEXDATA</td>
<td>Flow control restrictions on expedited data flow that led to a TFLOW error have been lifted. Expedited data may be sent again.</td>
</tr>
</tbody>
</table>

On failure, -1 is returned and t_errno is set to indicate the error.

Error Codes
On failure, t_errno is set to one of the following:

<table>
<thead>
<tr>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>TBADF</td>
<td>The specified file descriptor does not refer to a transport endpoint.</td>
</tr>
<tr>
<td>TSYSERR</td>
<td>A system error has occurred during execution of this subroutine.</td>
</tr>
<tr>
<td>TPROTO</td>
<td>This error indicates that a communication problem has been detected between the X/Open Transport Interface and the transport provider for which there is no other suitable X/Open Transport Interface (t_errno).</td>
</tr>
</tbody>
</table>

Related Information
The t_open subroutine, t_snd subroutine, t_sndudata subroutine.

Chapter 3. Streams 399
Description
The **t_open** subroutine must be called as the first step in the initialization of a transport endpoint. This subroutine establishes a transport endpoint by supplying a transport provider identifier that indicates a particular transport provider (for example, transport protocol) and returning a file descriptor that identifies that endpoint.

This subroutine also returns various default characteristics of the underlying transport protocol by setting fields in the **t_info** structure.

Parameters
- **name**  Points to a transport provider identifier.
- **oflag** Identifies any open flags (as in the **open** exec). The **oflag** parameter is constructed from **O_RDWR** optionally bitwise inclusive-OR-ed with **O_NONBLOCK**. These flags are defined by the **fcntl.h** header file. The file descriptor returned by the **t_open** subroutine is used by all subsequent subroutines to identify the particular local transport endpoint.
The values of the fields have the following meanings:

**addr**  
A value greater than zero indicates the maximum size of a transport protocol address and a value of -2 specifies that the transport provider does not provide user access to transport protocol addresses.

**options**  
A value greater than zero indicates the maximum number of bytes of protocol-specific options supported by the provider, and a value of -2 specifies that the transport provider does not support user-settable options.

**tsdu**  
A value greater than zero specifies the maximum size of a transport service data unit (TSDU); a value of zero specifies that the transport provider does not support the concept of TSDU, although it does support the sending of a data stream with no logical boundaries preserved across a connection; a value of -1 specifies that there is no limit on the size of a TSDU; and a value of -2 specifies that the transfer of normal data is not supported by the transport provider.

**etsdu**  
A value greater than zero specifies the maximum size of an expedited transport service data unit (ETSDU); a value of zero specifies that the transport provider does not support the concept of ETSDU, although it does support the sending of an expedited data stream with no logical boundaries preserved across a connection; a value of -1 specifies that there is no limit on the size of an ETSDU; and a value of -2 specifies that the transfer of expedited data is not supported by the transport provider. Note that the semantics of expedited data may be quite different for different transport providers.

**connect**  
A value greater than zero specifies the maximum amount of data that may be associated with connection establishment subroutines and a value of -2 specifies that the transport provider does not allow data to be sent with connection establishment subroutines.

**discon**  
A value greater than zero specifies the maximum amount of data that may be associated with the **t_snddis** and **t_rcvdis** subroutines and a value of -2 specifies that the transport provider does not allow data to be sent with the abortive release subroutines.
servtype
This field specifies the service type supported by the transport provider. The valid values on return are:

T_COTS
The transport provider supports a connection-mode service but does not support the optional orderly release facility.

T_COTS_ORD
The transport provider supports a connection-mode service with the optional orderly release facility.

T_CLTS
The transport provider supports a connectionless-mode service. For this service type, t_open will return -2 for etsdu, connect and discon.

A single transport endpoint may support only one of the above services at one time.

flags
This is a bit field used to specify other information about the transport provider. If the T_SENDZERO bit is set in flags, this indicates the underlying transport provider supports the sending of zero-length TSDUs.

If a transport user is concerned with protocol independence, the above sizes may be accessed to determine how large the buffers must be to hold each piece of information. Alternatively, the t_alloc subroutine may be used to allocate these buffers. An error will result if a transport user exceeds the allowed data size on any subroutine.

If the info parameter is set to a null pointer by the transport user, no protocol information is returned by the t_open subroutine.

Valid States
T_UNINIT

Return Values
Valid file descriptor Successful completion.
-1 t_errno is set to indicate an error.

Error Codes
On failure, t_errno is set to one of the following:

<table>
<thead>
<tr>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>TBADFLAG</td>
<td>An invalid flag is specified.</td>
</tr>
<tr>
<td>TBADNAME</td>
<td>Invalid transport provider name.</td>
</tr>
<tr>
<td>TSYSERR</td>
<td>A system error has occurred during execution of this subroutine.</td>
</tr>
<tr>
<td>TPROTO</td>
<td>This error indicates that a communication problem has been detected between</td>
</tr>
<tr>
<td></td>
<td>the X/Open Transport Interface and the transport provider for which there</td>
</tr>
<tr>
<td></td>
<td>is no other suitable X/Open Transport Interface (t_errno).</td>
</tr>
</tbody>
</table>

Related Information
The t_open subroutine.
t_optmgmt Subroutine for X/Open Transport Interface

Purpose
Manage options for a transport endpoint.

Library
X/Open Transport Interface Library (libxti.a)

Syntax
```
#include <xti.h>

int t_optmgmt(
    int fd,
    const struct t_optmgmt *req,
    struct t_optmgmt *ret)
```

Description
The `t_optmgmt` subroutine enables a transport user to retrieve, verify, or negotiate protocol options with the transport provider.

The `req` and `ret` parameters both point to a `t_optmgmt` structure containing the following members:
```
struct netbuf opt;
long flags;
```
Within this structure, the fields have the following meaning:

<table>
<thead>
<tr>
<th>Field</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>opt</td>
<td>Identifies protocol options. The options are represented by a <code>netbuf</code> structure in a manner similar to the address in the <code>t_bind</code> subroutine:</td>
</tr>
<tr>
<td>len</td>
<td>Specifies the number of bytes in the options and on return, specifies the number of bytes of options returned.</td>
</tr>
<tr>
<td>buf</td>
<td>Points to the options buffer. For the <code>ret</code> parameter, <code>buf</code> points to the buffer where the options are to be placed. Each option in the options buffer is of the form <code>struct t_opthdr</code> possibly followed by an option value. The fields of this structure and the values are:</td>
</tr>
<tr>
<td>level</td>
<td>Identifies the X/Open Transport Interface level or a protocol of the transport provider.</td>
</tr>
<tr>
<td>name</td>
<td>Identifies the option within the level.</td>
</tr>
<tr>
<td>len</td>
<td>Contains its total length, for example, the length of the option header <code>t_opthdr</code> plus the length of the option value. If <code>t_optmgmt</code> is called with the action <code>T_NEGOTIATE</code> set.</td>
</tr>
<tr>
<td>status</td>
<td>Contains information about the success or failure of a negotiation.</td>
</tr>
</tbody>
</table>

Each option in the input or output option buffer must start at a long-word boundary. The macro `OPT_NEXTHDR (pbuf, buflen, poption)` can be used for that purpose. The macro parameters are as follows:

- `pbuf` Specifies a pointer to an option buffer `opt.buf`.
- `buflen` The length of the option buffer pointed to by `pbuf`.
- `poption` Points to the current option in the option buffer. `OPT_NEXTHDR` returns a pointer to the position of the next option or returns a null pointer if the option buffer is exhausted. The macro is helpful for writing and reading. See the `xti.h` header file for the exact definition of this structure.

If the transport user specifies several options on input, all options must address the same level. If any option in the options buffer does not indicate the same level as the first option, or the level specified is unsupported, then the `t_optmgmt` request fails with `TBADOPT`. If the error is detected, some options may have successfully negotiated. The transport user can check the current status by calling the `t_optmgmt` subroutine with the `T_CURRENT` flag set.

**Note:** “The Use of Options” contains a detailed description about the use of options and should be read before using this subroutine.

- `maxlen` Has no meaning for the `req` parameter, but must be set in the `ret` parameter to specify the maximum size of the options buffer. On return, `len` specifies the number of bytes of options returned. The value in `maxlen` has no meaning for the `req` argument.
**Flags**

Specifies the action to take with those options. The flags field of req must specify one of the following actions:

**T_CHECK**

This action enables the user to verify whether the options specified in the req parameter are supported by the transport provider. If an option is specified with no option value, (that is, it consists only of a t_opthdr structure), the option is returned with its status field set to one of the following:

- **T_SUCCESS** - if it is supported.
- **T_NOTSUPPORT** - if it is not or needs additional user privileges.
- **T_READONLY** - if it is read-only (in the current X/Open Transport Interface state).

No option value is returned. If an option is specified with an option value, the status field of the returned option has the same value, as if the user had tried to negotiate this value with **T_NEGOTIATE**. If the status is **T_SUCCESS**, **T_FAILURE**, **T_NOTSUPPORT**, or **T_READONLY**, the returned option value is the same as the one requested on input.

The overall result of the option checks is returned in the flags field of the netbuf structure pointed to by the ret parameter. This field contains the worst single result of the option checks, where the rating is the same as for **T_NEGOTIATE**.

Note, that no negotiation takes place. All currently effective option values remain unchanged.

**T_CURRENT**

This action enables the transport user to retrieve the currently effective option values. The user specifies the options of interest in the opt fields in the netbuf structure pointed to by the req parameter. The option values are irrelevant and will be ignored; it is sufficient to specify the t_opthdr part of an option only. The currently effective values are then returned in opt fields in the netbuf structure pointed to by the ret parameter.

The status field returned is on of the following:

- **T_NOTSUPPORT** if the protocol level does not support this option or the transport user illegally requested a privileged option.
- **T_READONLY** if the option is read-only.
- **T_SUCCESS** in all other cases.

The overall result of the option checks is returned in the flags field of the netbuf structure pointed to by the ret parameter. This field contains the worst single result of the option checks, where the rating is the same as for **T_NEGOTIATE**.

For each level, the **T_ALLOPT** option (see below) can be requested on input. All supported options of this level with their default values are then returned.
<table>
<thead>
<tr>
<th>Field</th>
<th>Description</th>
</tr>
</thead>
</table>
| T_DEFAULT | This action enables the transport user to retrieve the default option values. The user specifies the options of interest in the `opt` fields in the `netbuf` structure pointed to by the `req` parameter. The option values are irrelevant and will be ignored; it is sufficient to specify the `t_opthdr` part of an option only. The default values are then returned in the `opt` field of the `netbuf` structure pointed to by the `ret` parameter. The `status` field returned is one of the following:  
  - T_NOTSUPPORT if the protocol level does not support this option or the transport user illegally requested a privileged option.  
  - T_READONLY if the option is read-only.  
  - T_SUCCESS in all other cases.  
  The overall result of the option checks is returned in the `flags` field of the `ret` parameter `netbuf` structure. This field contains the worst single result of the option checks, where the rating is the same as for T_NEGOTIATE. The overall result of the option checks is returned in the `flags` field of the `ret` parameter `netbuf` structure. This field contains the worst single result of the option checks, where the rating is the same as for T_NEGOTIATE.  
  For each level, the T_ALLOPT option (see below) can be requested on input. All supported options of this level with their default values are then returned. In this case, the `maxlen` value of the `opt` field in the `ret` parameter `netbuf` structure must be given at least the value of the `options` field of the `info` parameter (see the `t_getinfo` or `t_open subroutines`) before the call. |
| T_NEGOTIATE | This action enables the transport user to negotiate option values. The user specifies the options of interest and their values in the buffer specified in the `req` parameter `netbuf` structure. The negotiated option values are returned in the buffer pointed to by the `opt` field of the `ret` parameter `netbuf` structure. The `status` field of each returned option is set to indicate the result of the negotiation. The value is one of the following:  
  - T_SUCCESS if the proposed value was negotiated.  
  - T_PARTSUCCESS if a degraded value was negotiated.  
  - T_FAILURE is the negotiation failed (according to the negotiation rules).  
  - T_NOTSUPPORT if the transport provider does not support this option or illegally requests negotiation of a privileged option  
  - T_READONLY if modification of a read-only option was requested.  
  If the status is T_SUCCESS, T_FAILURE, T_NOTSUPPORT or T_READONLY, the returned option value is the same as the one requested on input. The overall result of the negotiation is returned in the `flags` field of the `ret` parameter `netbuf` structure. This field contains the worst single result, whereby the rating is done according to the following order, where T_NOTSUPPORT is the worst result and T_SUCCESS is the best:  
  - T_NOTSUPPORT  
  - T_FAILURE  
  - T_PARTSUCCESS  
  - T_SUCCESS.  
  For each level, the T_ALLOPT option (see below) can be requested on input. This option has no value and consists of a `t_opthdr` only. This input requests negotiation of all supported options of this level to their default values. The result is returned option by option in the `opt` field of the structure pointed to in the `ret` parameter. Depending on the state of the transport endpoint, not all requests to negotiate the default value may be successful. |
The T_ALLOPT option can only be used with the t_optmgmt structure and the actions T_NEGOTIATE, T_DEFAULT and T_CURRENT. This option can be used with any supported level and addresses all supported options of this level. The option has no value and consists of a t_opthdr only. Since only options of one level may be addressed in a t_optmgmt call, this option should not be requested together with other options. The subroutine returns as soon as this option has been processed.

Options are independently processed in the order they appear in the input option buffer. If an option is multiply input, it depends on the implementation whether it is multiply output or whether it is returned only once.

Transport providers may not be able to provide an interface capable of supporting T_NEGOTIATE and/or T_CHECK functionalities. When this is the case, the error TNOTSUPPORT is returned.

The subroutine t_optmgmt may block under various circumstances and depending on the implementation. For example, the subroutine will block if the protocol addressed by the call resides on a separate controller. It may also block due to flow control constraints, if data previously sent across this transport endpoint has not yet been fully processed. If the subroutine is interrupted by a signal, the option negotiations that have been done so far may remain valid. The behavior of the subroutine is not changed if O_NONBLOCK is set.

### Parameters

- **fd**: Identifies a transport endpoint.
- **req**: Requests a specific action of the provider.
- **ret**: Returns options and flag values to the user.

### X/Open Transport Interface-Level Options

X/Open Transport Interface (XTI) level options are not specific for a particular transport provider. An XTI implementation supports none, all, or any subset of the options defined below. An implementation may restrict the use of any of these options by offering them only in the privileged or read-only mode, or if the bound transport endpoint identified by the fd parameter relates to specific transport providers.

The subsequent options are not association-related (see Chapter 5, The Use of Options). They may be negotiated in all XTI states except T_UNINIT.

The protocol level is XTI_GENERIC. For this level, the following options are defined (the type of each option value is of type unsigned long unless otherwise indicated):

#### XTI-Level Options

<table>
<thead>
<tr>
<th>Option Name</th>
<th>Legal Option Value</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>XTI_DEBUG (array of unsigned longs)</td>
<td>see text</td>
<td>enable debugging</td>
</tr>
<tr>
<td>XTI_LINGER (struct linger)</td>
<td>see text</td>
<td>linger on close if data is present</td>
</tr>
<tr>
<td>XTI_RCVBUF</td>
<td>size in octets</td>
<td>receive buffer size</td>
</tr>
<tr>
<td>XTI_RCVLOWAT</td>
<td>size in octets</td>
<td>receive low-water mark</td>
</tr>
<tr>
<td>XTI_SNDBUF</td>
<td>size in octets</td>
<td>send buffer size</td>
</tr>
<tr>
<td>XTI_SNDBUF0</td>
<td>size in octets</td>
<td>send low-water mark</td>
</tr>
</tbody>
</table>

A request for XTI_DEBUG is an absolute requirement. A request to activate XTI_LINGER is an absolute requirement; the timeout value to this option is not. XTI_RCVBUF, XTI_RCVLOWAT, XTI_SNDBUF and XTI_SNDBUF0 are not absolute requirements.
Option                        Description
XTI_DEBUG                    Enables debugging. The values of this option are implementation-defined. Debugging is
disabled if the option is specified with no value (for example, with an option header only).

The system supplies utilities to process the traces. An implementation may also provide other
means for debugging.

XTI_LINGER                   Lingers the execution of a t_close subroutine or the close exec if send data is still queued in
                              the send buffer. The option value specifies the linger period. If a close exec or t_close
                              subroutine is issued and the send buffer is not empty, the system attempts to send the
                              pending data within the linger period before closing the endpoint. Data still pending after the
                              linger period has elapsed is discarded.

Depending on the implementation, the t_close subroutine or close exec either, at a
maximum, block the linger period, or immediately return, whereupon, at most, the system
holds the connection in existence for the linger period.

The option value consists of a structure t_linger declared as:

struct t_linger {
    long l_onoff;
    long l_linger;
}                         

The fields of the structure and the legal values are:

l_onoff         Switches the option on or off. The value l_onoff is an absolute requirement. The
                possible values are:
                T_NO       switch option off
                T_YES      activate option

l_linger       Determines the linger period in seconds. The transport user can request the default
                value by setting the field to T_UNSPEC. The default timeout value depends on the
                underlying transport provider (it is often T_INFINITE). Legal values for this field are
                T_UNSPEC, T_INFINITE and all non-negative numbers.

                        The l_linger value is not an absolute requirement. The implementation may place
                        upper and lower limits to this value. Requests that fall short of the lower limit are
                        negotiated to the lower limit.

Note: Note that this option does not linger the execution of the t_snndis subroutine.

XTI_RCVBUF                Adjusts the internal buffer size allocated for the receive buffer. The buffer size may
                              be increased for high-volume connections, or decreased to limit the possible backlog of incoming
                              data.

                              This request is not an absolute requirement. The implementation may place upper and lower
                              limits on the option value. Requests that fall short of the lower limit are negotiated to the
                              lower limit.

                              Legal values are all positive numbers.

XTI_RCVLOWAT               Sets a low-water mark in the receive buffer. The option value gives the minimal number of
                              bytes that must have accumulated in the receive buffer before they become visible to the
                              transport user. If and when the amount of accumulated receive data exceeds the low-water
                              mark, a T_DATA event is created, an event mechanism (for example, the poll or select
                              subroutines) indicates the data, and the data can be read by the t_rcv or t_rcvudata
                              subroutines.

                              This request is not an absolute requirement. The implementation may place upper and lower
                              limits on the option value. Requests that fall short of the lower limit are negotiated to the
                              lower limit.

                              Legal values are all positive numbers.
<table>
<thead>
<tr>
<th>Option</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>XTI_SNDBUF</td>
<td>Adjusts the internal buffer size allocated for the send buffer.</td>
</tr>
<tr>
<td></td>
<td>This request is not an absolute requirement. The implementation may place</td>
</tr>
<tr>
<td></td>
<td>upper and lower limits on the option value. Requests that fall short of the</td>
</tr>
<tr>
<td></td>
<td>lower limit are negotiated to the lower limit.</td>
</tr>
<tr>
<td></td>
<td>Legal values are all positive numbers.</td>
</tr>
<tr>
<td>XTI_SNDLOWAT</td>
<td>Sets a low-water mark in the send buffer. The option value gives the</td>
</tr>
<tr>
<td></td>
<td>minimal number of bytes that must have accumulated in the send buffer</td>
</tr>
<tr>
<td></td>
<td>before they are sent.</td>
</tr>
<tr>
<td></td>
<td>This request is not an absolute requirement. The implementation may place</td>
</tr>
<tr>
<td></td>
<td>upper and lower limits on the option value. Requests that fall short of the</td>
</tr>
<tr>
<td></td>
<td>lower limit are negotiated to the lower limit.</td>
</tr>
<tr>
<td></td>
<td>Legal values are all positive numbers.</td>
</tr>
</tbody>
</table>

**Valid States**

ALL - except from T_UNINIT.

**Return Values**

0  Successful completion.
-1  t_errno is set to indicate an error.

**Error Codes**

On failure, t_errno is set to one of the following:

<table>
<thead>
<tr>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>TACCES</td>
<td>The user does not have permission to negotiate the specified options.</td>
</tr>
<tr>
<td>TBADF</td>
<td>The specified file descriptor does not refer to a transport endpoint.</td>
</tr>
<tr>
<td>TBADFLAG</td>
<td>An invalid flag was specified.</td>
</tr>
<tr>
<td>TBADOPT</td>
<td>The specified options were in an incorrect format or contained illegal</td>
</tr>
<tr>
<td></td>
<td>information.</td>
</tr>
<tr>
<td>TBUFOVFLW</td>
<td>The number of bytes allowed for an incoming argument (maxlen) is greater</td>
</tr>
<tr>
<td></td>
<td>than 0 but not sufficient to store the value of that argument. The</td>
</tr>
<tr>
<td></td>
<td>information to be returned in ret will be discarded.</td>
</tr>
<tr>
<td>TOUTSTATE</td>
<td>The subroutine was issued in the wrong sequence.</td>
</tr>
<tr>
<td>TPROTO</td>
<td>This error indicates that a communication problem has been detected between</td>
</tr>
<tr>
<td></td>
<td>the X/Open Transport Interface and the transport provider for which there</td>
</tr>
<tr>
<td></td>
<td>is no other suitable X/Open Transport Interface (t_errno).</td>
</tr>
<tr>
<td>TSYSERR</td>
<td>A system error has occurred during execution of this subroutine.</td>
</tr>
</tbody>
</table>

**Related Information**

The `t_accept` subroutine, `t_alloc` subroutine, `t_connect` subroutine, `t_getinfo` subroutine, `t_listen` subroutine, `t_open` subroutine, `t_rcvconnect` subroutine.

**t_rcv Subroutine for X/Open Transport Interface**

**Purpose**

Receive data or expedited data sent over a connection.

**Library**

X/Open Transport Interface Library (libxti.a)
Syntax

```c
#include <xti.h>

int t_rcv (  
    int fd,  
    void *buf,  
    unsigned int nbytes,  
    int *flags)
```

Description

The `t_rcv` subroutine receives either normal or expedited data. By default, the `t_rcv` subroutine operates in synchronous mode and waits for data to arrive if none is currently available. However, if `O_NONBLOCK` is set via the `t_open` subroutine or the `fcntl` parameter, the `t_rcv` subroutine executes in asynchronous mode and fails if no data is available. (See the `TNODATA` error in “Error Codes” below.)

Parameters

- `fd` Identifies the local transport endpoint through which data will arrive.
- `buf` Points to a receive buffer where user data will be placed.
- `nbytes` Specifies the size of the receive buffer.
- `flags` Specifies optional flags. This parameter may be set on return from the `t_rcv` subroutine. The possible values are:
  - `T_MORE` If set, on return from the call, indicates that there is more data, and the current transport service data unit (TSDU) or expedited transport service data unit (ETSDU) must be received in multiple `t_rcv` calls. In the asynchronous mode, the `T_MORE` flag may be set on return from the `t_rcv` call even when the number of bytes received is less than the size of the receive buffer specified. Each `t_rcv` call with the `T_MORE` flag set, indicates that another `t_rcv` call must follow to get more data for the current TSDU. The end of the TSDU is identified by the return of a `t_rcv` call with the `T_MORE` flag not set. If the transport provider does not support the concept of a TSDU as indicated in the `info` parameter on return from the `t_open` or `t_getinfo` subroutines, the `T_MORE` flag is not meaningful and should be ignored. If the `nbytes` parameter is greater than zero on the call to `t_rcv`, `t_rcv` returns 0 only if the end of a TSDU is being returned to the user.
  - `T_EXPEDITED` If set, the data returned is expedited data. If the number of bytes of expedited data exceeds the value of the `nbytes` parameter, `t_rcv` will set `T_EXPEDITED` and `T_MORE` on return from the initial call. Subsequent calls to retrieve the remaining ETSDU will have `T_EXPEDITED` set on return. The end of the ETSDU is identified by the return of a `t_rcv` call with the `T_MORE` flag not set.

In synchronous mode, the only way to notify the user of the arrival of normal or expedited data is to issue this subroutine or check for the `T_DATA` or `T_EXDATA` events using the `t_look` subroutine. Additionally, the process can arrange to be notified via the Event Management interface.

Valid States

- `T_DATAXFER`, `T_OUTREL`.

Return Values

On successful completion, the `t_rcv` subroutine returns the number of bytes received. Otherwise, it returns -1 on failure and `t_errno` is set to indicate the error.
Error Codes

On failure, **t_errno** is set to one of the following:

<table>
<thead>
<tr>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>TBADF</td>
<td>The specified file descriptor does not refer to a transport endpoint.</td>
</tr>
<tr>
<td>TLOOK</td>
<td>An asynchronous event has occurred on this transport endpoint and requires immediate attention.</td>
</tr>
<tr>
<td>TNODATA</td>
<td>O_NONBLOCK was set, but no data is currently available from the transport provider.</td>
</tr>
<tr>
<td>TNOTSUPPORT</td>
<td>This subroutine is not supported by the underlying transport provider.</td>
</tr>
<tr>
<td>TOUTSTATE</td>
<td>The subroutine was issued in the wrong sequence on the transport endpoint referenced by the fd parameter.</td>
</tr>
<tr>
<td>TPROTO</td>
<td>This error indicates that a communication problem has been detected between the X/Open Transport Interface and the transport provider for which there is no other suitable X/Open Transport Interface (<strong>t_errno</strong>).</td>
</tr>
<tr>
<td>TSYSERR</td>
<td>A system error has occurred during execution of this subroutine.</td>
</tr>
</tbody>
</table>

Related Information

The [fcntl](#) subroutine, [t_getinfo](#) subroutine, [t_look](#) subroutine, [t_open](#) subroutine, [t_snd](#) subroutine.

---

**t_rcvconnect Subroutine for X/Open Transport Interface**

**Purpose**

Receive the confirmation from a connect request.

**Library**

X/Open Transport Interface Library (**libxti.a**)

**Syntax**

```c
#include <xti.h>
int t_rcvconnect (fd, call)
  int fd;
  struct t_call *call;
```

**Description**

The **t_rcvconnect** subroutine enables a calling transport user to determine the status of a previously sent connect request and is used in conjunction with the **t_connect** subroutine to establish a connection in asynchronous mode. The connection is established on successful completion of this subroutine.

**Parameters**

- **fd** Identifies the local transport endpoint where communication will be established.
call: Contains information associated with the newly established connection. The call parameter points to a t_call structure which contains the following members:

```c
struct netbuf addr;
struct netbuf opt;
struct netbuf udata;
int sequence;
```

The fields of the t_call structure are:

- **addr**: Returns the protocol address associated with the responding transport endpoint.
- **opt**: Presents any options associated with the connection.
- **udata**: Points to optional user data that may be returned by the destination transport user during connection establishment.
- **sequence**: Has no meaning for this subroutine.

The maxlen field of each t_call member must be set before issuing this subroutine to indicate the maximum size of the buffer for each. However, the value of the call parameter may be a null pointer, in which case no information is given to the user on return from the t_rcvconnect subroutine. By default, the t_rcvconnect subroutine executes in synchronous mode and waits for the connection to be established before returning. On return, the addr, opt and udata fields reflect values associated with the connection.

If O_NONBLOCK is set (via the t_open subroutine or fcntl), the t_rcvconnect subroutine executes in asynchronous mode, and reduces to a poll for existing connect confirmations. If none are available, the t_rcvconnect subroutine fails and returns immediately without waiting for the connection to be established. (See T_NODATA in “Error Codes” below.) In this case, the t_rcvconnect subroutine must be called again to complete the connection establishment phase and retrieve the information returned in the call parameter.

### Valid States

**T_OUTCON**

### Return Values

<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>t_errno is set to indicate an error.</td>
</tr>
</tbody>
</table>

### Error Codes

On failure, t_errno is set to one of the following:

<table>
<thead>
<tr>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>TBADF</td>
<td>The specified file descriptor does not refer to a transport endpoint.</td>
</tr>
<tr>
<td>TBUFOVFLW</td>
<td>The number of bytes allocated for an incoming argument (maxlen) is greater</td>
</tr>
<tr>
<td></td>
<td>than 0 but not sufficient to store the value of that argument, and the</td>
</tr>
<tr>
<td></td>
<td>connect information to be returned in call will be discarded. The provider's</td>
</tr>
<tr>
<td></td>
<td>state, as seen by the user, will be changed to T_DATAFER.</td>
</tr>
<tr>
<td>TLOOK</td>
<td>An asynchronous event has occurred on the transport connection and requires</td>
</tr>
<tr>
<td></td>
<td>immediate attention.</td>
</tr>
<tr>
<td>T_NODATA</td>
<td>O_NONBLOCK was set, but a connect confirmation has not yet arrived.</td>
</tr>
<tr>
<td>TNOTSUPPORT</td>
<td>This subroutine is not supported by the underlying transport provider.</td>
</tr>
<tr>
<td>TOUTSTATE</td>
<td>The subroutine was issued in the wrong sequence on the transport endpoint</td>
</tr>
<tr>
<td></td>
<td>referenced by the fd parameter.</td>
</tr>
<tr>
<td>TPROTO</td>
<td>This error indicates that a communication problem has been detected between</td>
</tr>
<tr>
<td></td>
<td>the X/Open Transport Interface and the transport provider for which there is</td>
</tr>
<tr>
<td></td>
<td>no other suitable X/Open Transport Interface (t_errno).</td>
</tr>
<tr>
<td>T_SYSERR</td>
<td>A system error has occurred during execution of this subroutine.</td>
</tr>
</tbody>
</table>
t_rcvdis Subroutine for X/Open Transport Interface

Purpose
Retrieve information from disconnect.

Library
X/Open Transport Interface Library (libxti.a)

Syntax
```c
#include <xti.h>
int t_rcvdis (fd, discon)
int fd;
struct t_discon *discon;
```

Description
The t_rcvdis subroutine identifies the cause of a disconnect and retrieves any user data sent with the disconnect.

Parameters
- `fd` Identifies the local transport endpoint where the connection existed.
- `discon` Points to a `t_discon` structure containing the following members:
  ```
  struct netbuf udata;
  int reason;
  int sequence;
  ```

The `t_discon` structure fields are:
- `reason` Specifies the reason for the disconnect through a protocol-dependent reason code.
- `udata` Identifies any user data that was sent with the disconnect.
- `sequence` May identify an outstanding connect indication with which the disconnect is associated. The `sequence` field is only meaningful when the `t_rcvdis` subroutine is issued by a passive transport user who has executed one or more `t_listen` subroutines and is processing the resulting connect indications. If a disconnect indication occurs, the `sequence` field can be used to identify which of the outstanding connect indications is associated with the disconnect.

If a user does not care if there is incoming data and does not need to know the value of the `reason` or `sequence` fields, the `discon` field value may be a null pointer and any user data associated with the disconnect will be discarded. However, if a user has retrieved more than one outstanding connect indication (via the `t_listen` subroutine) and the `discon` field value is a null pointer, the user will be unable to identify with which connect indication the disconnect is associated.

Valid States
- `T_DATAXFER`, `T_OUTCON`, `T_OUTREL`, `T_INREL`, `T_INCON` (`ocnt > 0`).
Return Values

0    Successful completion.
-1    t_errno is set to indicate an error.

Error Codes

On failure, t_errno is set to one of the following:

<table>
<thead>
<tr>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>TBADF</td>
<td>The specified file descriptor does not refer to a transport endpoint.</td>
</tr>
<tr>
<td>TBUFOVFLW</td>
<td>The number of bytes allocated for incoming data (maxlen) is greater than 0 but not sufficient to store the data. If the fd parameter is a passive endpoint with ocnt &gt; 1, it remains in state T_INCON; otherwise, the endpoint state is set to T_IDLE.</td>
</tr>
<tr>
<td>TNODIS</td>
<td>No disconnect indication currently exists on the specified transport endpoint.</td>
</tr>
<tr>
<td>TNOTSUPPORT</td>
<td>This subroutine is not supported by the underlying transport provider.</td>
</tr>
<tr>
<td>TOUTSTATE</td>
<td>The subroutine was issued in the wrong sequence on the transport endpoint referenced by the fd parameter.</td>
</tr>
<tr>
<td>TPROTO</td>
<td>This error indicates that a communication problem has been detected between the X/Open Transport Interface and the transport provider for which there is no other suitable X/Open Transport Interface (t_errno).</td>
</tr>
<tr>
<td>TSYSERR</td>
<td>A system error has occurred during execution of this subroutine.</td>
</tr>
</tbody>
</table>

Related Information

The t_alloc subroutine, t_connect subroutine, t_listen subroutine, t_open subroutine, t_snddis subroutine.

---

**t_rcvrel Subroutine for X/Open Transport Interface**

**Purpose**

Acknowledging receipt of an orderly release indication.

**Library**

X/Open Transport Interface Library (libxti.a)

**Syntax**

```c
#include <xti.h>

int t_rcvrel (fd)
    int fd;
```

**Description**

The t_rcvrel subroutine is used to acknowledge receipt of an orderly release indication. After receipt of this indication, the user may not attempt to receive more data because such an attempt will block forever. However, the user may continue to send data over the connection if the t_sndrel subroutine has not been called by the user. This function is an optional service of the transport provider, and is only supported if the transport provider returned the T_COTS_ORD service type on t_open or t_getinfo calls.

**Parameter**

| fd    | Identifies the local transport endpoint where the connection exists. |
Valid States
T_DATAFER, T_OUTREL.

Return Values
0 Successful completion.
-1 t_errno is set to indicate an error.

Error Codes
On failure, t_errno is set to one of the following:

<table>
<thead>
<tr>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>TBADF</td>
<td>The specified file descriptor does not refer to a transport endpoint.</td>
</tr>
<tr>
<td>TLOOK</td>
<td>An asynchronous event has occurred on this transport endpoint and requires immediate attention.</td>
</tr>
<tr>
<td>TNOREL</td>
<td>No orderly release indication currently exists on the specified transport endpoint.</td>
</tr>
<tr>
<td>TNOTSUPPORT</td>
<td>This subroutine was issued in the wrong sequence on the transport endpoint referenced by the fd parameter.</td>
</tr>
<tr>
<td>TOUTSTATE</td>
<td>This error indicates that a communication problem has been detected between the X/Open Transport Interface and the transport provider for which there is no other suitable X/Open Transport Interface (t_errno).</td>
</tr>
<tr>
<td>TPROTO</td>
<td>This error indicates that a communication problem has been detected between the X/Open Transport Interface and the transport provider for which there is no other suitable X/Open Transport Interface (t_errno).</td>
</tr>
<tr>
<td>TSYSERR</td>
<td>A system error has occurred during execution of this subroutine.</td>
</tr>
</tbody>
</table>

Related Information
The t_getinfo subroutine, t_open subroutine, t_sndrel subroutine.

**t_rcvudata Subroutine for X/Open Transport Interface**

**Purpose**
Receive a data unit.

**Library**
X/Open Transport Interface Library (libxti.a)

**Syntax**
```
#include <xti.h>
int t_rcvudata (fd, unitdata, flags)
int fd;
struct t_unitdata *unitdata;
int *flags;
```

**Description**
The **t_rcvudata** subroutine is used in connectionless mode to receive a data unit from another transport user.

By default, the **t_rcvudata** subroutine operates in synchronous mode and waits for a data unit to arrive if none is currently available. However, if **O_NONBLOCK** is set (via the **t_open** subroutine or **fcntl**), the **t_rcvudata** subroutine executes in asynchronous mode and fails if no data units are available.
If the buffer defined in the \textit{udata} field of the \textit{unitdata} parameter is not large enough to hold the current data unit, the buffer is filled and \texttt{T\_MORE} is set in the \textit{flags} parameter on return to indicate that another \texttt{t\_rcvudata} subroutine should be called to retrieve the rest of the data unit. Subsequent calls to the \texttt{t\_rcvudata} subroutine return zero for the length and options until the full data unit is received.

\textbf{Parameters}

\texttt{fd} \hspace{1cm} Identifies the local transport endpoint through which data will be received.

\texttt{unitdata} \hspace{1cm} Holds information associated with the received data unit. The \textit{unitdata} parameter points to a \texttt{t\_unitdata} structure containing the following members:

\begin{verbatim}
struct netbuf addr;
struct netbuf opt;
struct netbuf udata;
\end{verbatim}

On return from this call:

\texttt{addr} \hspace{1cm} Specifies the protocol address of the sending user.

\texttt{opt} \hspace{1cm} Identifies options that were associated with this data unit.

\texttt{udata} \hspace{1cm} Specifies the user data that was received.

The \textit{maxlen} field of \texttt{addr}, \texttt{opt}, and \texttt{udata} must be set before calling this subroutine to indicate the maximum size of the buffer for each.

\texttt{flags} \hspace{1cm} Indicates that the complete data unit was not received.

\textbf{Valid States}

\texttt{T\_IDLE}

\textbf{Return Values}

Upon successful completion, a value of 0 is returned. Otherwise, a value of -1 is returned and \texttt{t\_errno} is set to indicate an error.

\textbf{Error Codes}

On failure, \texttt{t\_errno} is set to one of the following:

<table>
<thead>
<tr>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>TBADF</td>
<td>The specified file descriptor does not refer to a transport endpoint.</td>
</tr>
<tr>
<td>TBODATA</td>
<td>\texttt{O_NONBLOCK} was set, but no data units are currently available from the transport provider.</td>
</tr>
<tr>
<td>TBUFOVFLW</td>
<td>The number of bytes allocated for the incoming protocol address or options (\textit{maxlen}) is greater than 0 but not sufficient to store the information. The unit data information to be returned in the \textit{unitdata} parameter is discarded.</td>
</tr>
<tr>
<td>TLOOK</td>
<td>An asynchronous event has occurred on the transport endpoint and requires immediate attention.</td>
</tr>
<tr>
<td>TNOTSUPPORT</td>
<td>This subroutine is not supported by the underlying transport provider.</td>
</tr>
<tr>
<td>TOUTSTATE</td>
<td>The subroutine was issued in the wrong sequence on the transport endpoint referenced by the \texttt{fd} parameter.</td>
</tr>
<tr>
<td>TPROTO</td>
<td>This error indicates that a communication problem has been detected between the X/Open Transport Interface and the transport provider for which there is no other suitable X/Open Transport Interface (\texttt{t_errno}).</td>
</tr>
<tr>
<td>TSYSErr</td>
<td>A system error has occurred during execution of this subroutine.</td>
</tr>
</tbody>
</table>

\textbf{Related Information}

The \texttt{fcntl} subroutine, \texttt{t\_alloc} subroutine, \texttt{t\_open} subroutine, \texttt{t\_rcvuderr} subroutine, \texttt{t\_sndudata} subroutine.

416 \hspace{1cm} Technical Reference: Communications, Volume 2
t_rcvuderr Subroutine for X/Open Transport Interface

Purpose
Receive a unit data error indication.

Library
X/Open Transport Interface Library (libxti.a)

Syntax
#include <xti.h>
int t_rcvuderr (fd, uderr)
int fd;
struct t_uderr *uderr;

Description
The t_rcvuderr subroutine is used in connectionless mode to receive information concerning an error on a previously sent data unit, and should only be issued following a unit data error indication. It informs the transport user that a data unit with a specific destination address and protocol options produced an error.

Parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>fd</td>
<td>Identifies the local transport endpoint through which the error report will be received.</td>
</tr>
<tr>
<td>uderr</td>
<td>Points to a t_uderr structure containing the following members:</td>
</tr>
<tr>
<td>addr</td>
<td>Specifies the destination protocol address of the erroneous data unit.</td>
</tr>
<tr>
<td>opt</td>
<td>Identifies options that were associated with the data unit.</td>
</tr>
<tr>
<td>error</td>
<td>Specifies a protocol-dependent error code.</td>
</tr>
</tbody>
</table>

The maxlen field of addr and opt must be set before calling this subroutine to indicate the maximum size of the buffer for each.

On return from this call:

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>addr</td>
<td>Specifies the destination protocol address of the erroneous data unit.</td>
</tr>
<tr>
<td>opt</td>
<td>Identifies options that were associated with the data unit.</td>
</tr>
<tr>
<td>error</td>
<td>Specifies a protocol-dependent error code.</td>
</tr>
</tbody>
</table>

If the user does not care to identify the data unit that produced an error, uderr may be set to a null pointer, and the t_rcvuderr subroutine simply clears the error indication without reporting any information to the user.

Valid States
T_IDLE

Return Values

<table>
<thead>
<tr>
<th>Return Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Successful completion.</td>
</tr>
<tr>
<td>-1</td>
<td>t_errno is set to indicate an error.</td>
</tr>
</tbody>
</table>
Error Codes

On failure, \texttt{t_errno} is set to one of the following:

<table>
<thead>
<tr>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>TBADF</td>
<td>The specified file descriptor does not refer to a transport endpoint.</td>
</tr>
<tr>
<td>TBUFOVFLW</td>
<td>The number of bytes allocated for the incoming protocol address or options (\textit{maxlen}) is greater than 0 but not sufficient to store the information. The unit data information to be returned in the \texttt{uderr} parameter is discarded.</td>
</tr>
<tr>
<td>TNOTSUPPORT</td>
<td>This subroutine is not supported by the underlying transport provider.</td>
</tr>
<tr>
<td>TNOUDERR</td>
<td>No unit data error indication currently exists on the specified transport endpoint.</td>
</tr>
<tr>
<td>TPROTO</td>
<td>This error indicates that a communication problem has been detected between the X/Open Transport Interface and the transport provider for which there is no other suitable X/Open Transport Interface (t_errno).</td>
</tr>
<tr>
<td>TSYSErr</td>
<td>A system error has occurred during execution of this subroutine.</td>
</tr>
</tbody>
</table>

Related Information

The \texttt{t_rcvudata} subroutine, \texttt{t_sndudata} subroutine.

\textbf{t_snd Subroutine for X/Open Transport Interface}

\textbf{Purpose}

Send data or expedited data over a connection.

\textbf{Library}

X/Open Transport Interface Library (\texttt{libxti.a})

\textbf{Syntax}

\begin{verbatim}
#include <xti.h>

int t_snd (  
    int \textit{fd},  
    void *\textit{buf},  
    unsigned \textit{int} \textit{nbytes},  
    int *\textit{flags})
\end{verbatim}

\textbf{Description}

The \texttt{t_snd} subroutine is used to send either normal or expedited data. By default, the \texttt{t_snd} subroutine operates in synchronous mode and may wait if flow control restrictions prevents the data from being accepted by the local transport provider at the time the call is made. However, if \texttt{O\_NONBLOCK} is set (via the \texttt{t_open} subroutine or \texttt{fcntl}), the \texttt{t_snd} subroutine executes in asynchronous mode, and fails immediately if there are flow control restrictions. The process can arrange to be informed when the flow control restrictions are cleared via either the \texttt{t_look} subroutine or the Event Management interface.

On successful completion, the \texttt{t_snd} subroutine returns the number of bytes accepted by the transport provider. Normally this equals the number of bytes specified in the \textit{nbytes} parameter. However, if \texttt{O\_NONBLOCK} is set, it is possible that only part of the data is actually accepted by the transport provider. In this case, the \texttt{t_snd} subroutine returns a value that is less than the value of the \textit{nbytes} parameter. If the value of the \textit{nbytes} parameter is zero and sending of zero octets is not supported by the underlying transport service, the \texttt{t_snd} subroutine returns -1 with \texttt{t_errno} set to \texttt{TBADDATA}.

It is important to remember that the transport provider treats all users of a transport endpoint as a single user. Therefore if several processes issue concurrent \texttt{t_snd} calls then the different data may be intermixed.
Multiple sends which exceed the maximum TSDU or ETSDU size may not be discovered by the X/Open Transport Interface. In this case an implementation-dependent error will result (generated by the transport provider) perhaps on a subsequent XTI call. This error may take the form of a connection abort, a T_SYSERR, a TBADDATA or a TPROTO error.

If multiple sends which exceed the maximum TSDU or ETSDU size are detected by the X/Open Transport Interface, t_snd fails with TBADDATA.

Parameters

\texttt{fd} \quad \text{Identifies the local transport endpoint over which data should be sent.}
\texttt{buf} \quad \text{Points to the user data.}
\texttt{nbytes} \quad \text{Specifies the number of bytes of user data to be sent.}
\texttt{flags} \quad \text{Specifies any optional flags described below:}

\begin{tabular}{ll}
\textbf{T_EXPEDITED} & If set in the \texttt{flags} parameter, the data is sent as expedited data and is subject to the interpretations of the transport provider. \\
\textbf{T_MORE} & If set in the \texttt{flags} parameter, indicates to the transport provider that the transport service data unit (TSDU) or expedited transport service data unit - ETSDU) is being sent through multiple \texttt{t_snd} calls. Each \texttt{t_snd} call with the T\_MORE flag set indicates that another \texttt{t_snd} call will follow with more data for the current TSDU (or ETSDU).
\end{tabular}

The end of the TSDU (or ETSDU) is identified by a \texttt{t_snd} call with the T\_MORE flag not set. Use of T\_MORE enables a user to break up large logical data units without losing the boundaries of those units at the other end of the connection. The flag implies nothing about how the data is packaged for transfer below the transport interface. If the transport provider does not support the concept of a TSDU, as indicated in the info parameter on return from the \texttt{t_open} or \texttt{t_getinfo} subroutines, the T\_MORE flag is not meaningful and is ignored if set.

The sending of a zero-length fragment of a TSDU or ETSDU is only permitted where this is used to indicate the end of a TSDU or ETSDU, for example, when the T\_MORE flag is not set. Some transport providers also forbid zero-length TSDUs and ETSDUs. See Appendix A, ISO Transport Protocol Information for a fuller explanation.

Valid States

T\_DATAXFER, T\_INREL.

Return Values

On successful completion, the \texttt{t_snd} subroutine returns the number of bytes accepted by the transport provider. Otherwise, -1 is returned on failure and \texttt{t_errno} is set to indicate the error.

Note, that in asynchronous mode, if the number of bytes accepted by the transport provider is less than the number of bytes requested, this may indicate that the transport provider is blocked due to flow control.

Error Codes

On failure, \texttt{t_errno} is set to one of the following:

\begin{tabular}{ll}
\textbf{Value} & \textbf{Description} \\
TBADF & The specified file descriptor does not refer to a transport endpoint.
\end{tabular}
**Value**  
**Description**  

TBADDATA  
Illegal amount of data:  
- A single send was attempted specifying a TSDU (ETSDU) or fragment TSDU (ETSDU) greater than that specified by the current values of the TSDU or ETSDU fields in the *info* argument;  
- a send of a zero byte TSDU (ETSDU) or zero byte fragment of a TSDU (ETSDU) is not supported by the provider (see Appendix A, ISO Transport Protocol Information).  
- multiple sends were attempted resulting in a TSDU (ETSDU) larger than that specified by the current value of the TSDU or ETSDU fields in the *info* argument - the ability of an XTI implementation to detect such an error case is implementation-dependent. See “Implementation Specifics”.

TBADFLAG  
An invalid flag was specified.

TFLOW  
O_NONBLOCK was set, but the flow control mechanism prevented the transport provider from accepting any data at this time.

TNOTSUPPORT  
This subroutine is not supported by the underlying transport provider.

TLOOK  
An asynchronous event has occurred on this transport endpoint.

TOUTSTATE  
The subroutine was issued in the wrong sequence on the transport endpoint referenced by the *fd* parameter.

TSYSERR  
A system error has occurred during execution of this subroutine.

TPROTO  
This error indicates that a communication problem has been detected between the X/Open Transport Interface and the transport provider for which there is no other suitable X/Open Transport Interface (*t_errno*).

---

**Related Information**

The [t_getinfo subroutine](#), [t_open subroutine](#), [t_rcv subroutine](#).

---

**t_snddis Subroutine for X/Open Transport Interface**

**Purpose**

Send user-initiated disconnect request.

**Library**

X/Open Transport Interface Library ([libxti.a](#))

**Syntax**

```c
#include <xti.h>

int t_snddis (  
  int *fd,  
  const struct t_call *call)
```

**Description**

The `t_snddis` subroutine is used to initiate an abortive release on an already established connection, or to reject a connect request.

The `t_snddis` subroutine is an abortive disconnect. Therefore a `t_snddis` call issued on a connection endpoint may cause data previously sent via the `t_snd` subroutine, or data not yet received, to be lost (even if an error is returned).

**Parameters**

- `fd` Identifies the local transport endpoint of the connection.
Specifies information associated with the abortive release. The call parameter points to a t_call structure which contains the following members:

```c
struct netbuf addr;
struct netbuf opt;
struct netbuf udata;
int sequence;
```

The values in the call parameter have different semantics, depending on the context of the call to the t_snddis subroutine. When rejecting a connect request, the call parameter must be non-null and contain a valid value of sequence to uniquely identify the rejected connect indication to the transport provider. The sequence field is only meaningful if the transport connection is in the T_INCON state. The addr and opt fields of the call parameter are ignored. In all other cases, the call parameter need only be used when data is being sent with the disconnect request. The addr, opt and sequence fields of the t_call structure are ignored. If the user does not wish to send data to the remote user, the value of the call parameter may be a null pointer.

The udata field specifies the user data to be sent to the remote user. The amount of user data must not exceed the limits supported by the transport provider, as returned in the t_open or t_getinfo subroutines info parameter discon field. If the len field of udata is zero, no data will be sent to the remote user.

**Valid States**

T_DATAxFER, T_OUTxCON, T_OUTxREL, T_INxREL, T_INxCON (ocnt > 0).

**Return Values**

0 Successful completion.

-1 t_errno is set to indicate an error.

**Error Codes**

On failure, t_errno is set to one of the following:

<table>
<thead>
<tr>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>TBADDATA</td>
<td>The amount of user data specified was not within the bounds allowed by the transport provider.</td>
</tr>
<tr>
<td>TBADF</td>
<td>The specified file descriptor does not refer to a transport endpoint.</td>
</tr>
<tr>
<td>TBADSEQ</td>
<td>An invalid sequence number was specified, or a null call pointer was specified, when rejecting a connect request.</td>
</tr>
<tr>
<td>TLOOK</td>
<td>An asynchronous event, which requires attention has occurred.</td>
</tr>
<tr>
<td>TNOTSUPPORT</td>
<td>This subroutine is not supported by the underlying transport provider.</td>
</tr>
<tr>
<td>TOUTSTATE</td>
<td>The subroutine was issued in the wrong sequence on the transport endpoint referenced by the fd parameter.</td>
</tr>
<tr>
<td>TPROTO</td>
<td>This error indicates that a communication problem has been detected between the X/Open Transport Interface and the transport provider for which there is no other suitable X/Open Transport Interface (t_errno).</td>
</tr>
<tr>
<td>TSYSERR</td>
<td>A system error has occurred during execution of this subroutine.</td>
</tr>
</tbody>
</table>

**Related Information**

The t_connect subroutine, t_getinfo subroutine, t_listen subroutine, t_open subroutine.

**t_sndrel Subroutine for X/Open Transport Interface**

**Purpose**

Initiate an orderly release.
Library
X/Open Transport Interface Library (libxti.a)

Syntax
#include <xti.h>
int t_sndrel (fd)
int fd;

Description
The t_sndrel subroutine is used to initiate an orderly release of a transport connection and indicates to the transport provider that the transport user has no more data to send.

After calling the t_sndrel subroutine, the user may not send any more data over the connection. However, a user may continue to receive data if an orderly release indication has not been received. This subroutine is an optional service of the transport provider and is only supported if the transport provider returned service type T_COTS_ORD on the t_open or t_getinfo subroutines.

Parameter
fd Identifies the local transport endpoint where the connection exists.

Valid States
T_DATAXFER, T_INREL.

Return Values
0 Successful completion.
-1 t_errno is set to indicate an error.

Error Codes
On failure, t_errno is set to one of the following:

<table>
<thead>
<tr>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>TBADF</td>
<td>The specified file descriptor does not refer to a transport endpoint.</td>
</tr>
<tr>
<td>TFLOW</td>
<td>O_NONBLOCK was set, but the flow control mechanism prevented the transport provider from accepting the subroutine at this time.</td>
</tr>
<tr>
<td>TLOOK</td>
<td>An asynchronous event has occurred on this transport endpoint and requires immediate attention.</td>
</tr>
<tr>
<td>TNOTSUPPORT</td>
<td>This subroutine is not supported by the underlying transport provider.</td>
</tr>
<tr>
<td>TOUTSTATE</td>
<td>The subroutine was issued in the wrong sequence on the transport endpoint referenced by the fd parameter.</td>
</tr>
<tr>
<td>TPROTO</td>
<td>This error indicates that a communication problem has been detected between X/Open Transport Interface and the transport provider for which there is no other suitable X/Open Transport Interface (t_errno).</td>
</tr>
<tr>
<td>TSYSERR</td>
<td>A system error has occurred during execution of this subroutine.</td>
</tr>
</tbody>
</table>

Related Information
The t_getinfo subroutine, t_open subroutine, t_rcvrel subroutine.
t_sndudata Subroutine for X/Open Transport Interface

Purpose
Send a data unit.

Library
X/Open Transport Interface Library (libxti.a)

Syntax
#include <xti.h>

int t_sndudata (  
    int fd,  
    const struct t_unitdata *unitdata)

Description
The t_sndudata subroutine is used in connectionless mode to send a data unit from another transport user.

By default, the t_sndudata subroutine operates in synchronous mode and waits if flow control restrictions prevents the data from being accepted by the local transport provider at the time the call is made. However, if O_NONBLOCK is set (via the t_open subroutine or fcntl), the t_sndudata subroutine executes in asynchronous mode and fails under such conditions. The process can arrange to be notified of the clearance of a flow control restriction via either the t_look subroutine or the Event Management interface.

If the amount of data specified in the udata field exceeds the TSDU size as returned in the t_open or t_getinfo subroutines info parameter tsdu field, a TBADDATA error will be generated. If the t_sndudata subroutine is called before the destination user has activated its transport endpoint (see the t_bind subroutine), the data unit may be discarded.

If it is not possible for the transport provider to immediately detect the conditions that cause the errors TBADDADDR and TBADOPT. These errors will alternatively be returned by the t_rcvuderr subroutine. Therefore, an application must be prepared to receive these errors in both of these ways.

Parameters

fd  Identifies the local transport endpoint through which data will be sent.
unitdata  Points to a t_unitdata structure containing the following members:

struct netbuf addr;
struct netbuf opt;
struct netbuf udata;

In the unitdata structure:

addr  Specifies the protocol address of the destination user.
opt  Identifies options that the user wants associated with this request. The user may choose not to specify what protocol options are associated with the transfer by setting the len field of opt to zero. In this case, the provider may use default options.
udata  Specifies the user data to be sent. If the len field of udata is zero, and sending of zero octets is not supported by the underlying transport service, the t_sndudata subroutine returns -1 with t_errno set to TBADDATA.
Valid States

T_IDLE

Return Values

0 Successful completion.
-1 t_errno is set to indicate an error.

Error Codes

On failure, t_errno is set to one of the following:

<table>
<thead>
<tr>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>TBADADDR</td>
<td>The specified protocol address was in an incorrect format or contained illegal information.</td>
</tr>
<tr>
<td>TBADDATA</td>
<td>Illegal amount of data. A single send was attempted specifying a TSDU greater than that specified in the info parameter, or a send of a zero byte TSDU is not supported by the provider.</td>
</tr>
<tr>
<td>TBADF</td>
<td>The specified file descriptor does not refer to a transport endpoint.</td>
</tr>
<tr>
<td>TBADOPT</td>
<td>The specified options were in an incorrect format or contained illegal information.</td>
</tr>
<tr>
<td>TFLOW</td>
<td>O_NONBLOCK was set, but the flow control mechanism prevented the transport provider from accepting any data at this time.</td>
</tr>
<tr>
<td>TLOOK</td>
<td>An asynchronous event has occurred on the transport endpoint.</td>
</tr>
<tr>
<td>TNOTSUPPORT</td>
<td>This subroutine is not supported by the underlying transport provider.</td>
</tr>
<tr>
<td>TOUTSTATE</td>
<td>The subroutine was issued in the wrong sequence on the transport endpoint referenced by the fd parameter.</td>
</tr>
<tr>
<td>TPROTO</td>
<td>This error indicates that a communication problem has been detected between the X/Open Transport Interface and the transport provider for which there is no other suitable X/Open Transport Interface (t_errno).</td>
</tr>
<tr>
<td>TSYSERR</td>
<td>A system error has occurred during execution of this subroutine.</td>
</tr>
</tbody>
</table>

Related Information

The fcntl subroutine, t_alloc subroutine, t_open subroutine, t_rcvudata subroutine, t_rcvuderr subroutine.

---

t_strerror Subroutine for X/Open Transport Interface

**Purpose**

Produce an error message string.

**Library**

X/Open Transport Interface Library (libxti.a)

**Syntax**

```
#include <xti.h>

const char *t_strerror ( int errnum )
```

**Description**

The t_strerror subroutine maps the error number to a language-dependent error message string and returns a pointer to the string. The error number specified by the errnum parameter corresponds to an X/Open Transport Interface error. The string pointed to is not modified by the program, but may be overwritten by a subsequent call to the t_strerror subroutine. The string is not terminated by a newline.
character. The language for error message strings written by the `t_strerror` subroutine is implementation-defined. If it is English, the error message string describing the value in `t_errno` is identical to the comments following the `t_errno` codes defined in the `xti.h` header file. If an error code is unknown, and the language is English, `t_strerror` returns the string.

"<error>: error unknown"

where `<error>` is the error number supplied as input. In other languages, an equivalent text is provided.

**Parameter**

`errno` Specifies the error number.

**Valid States**

ALL - except T_UNINIT.

**Return Values**

The `t_strerror` subroutine returns a pointer to the generated message string.

**Related Information**

The `t_errno` subroutine.

---

### t_sync Subroutine for X/Open Transport Interface

**Purpose**

Synchronize transport library.

**Library**

X/Open Transport Interface Library (`libxti.a`)

**Syntax**

```c
#include <xti.h>
int t_sync (fd)
int fd;
```

**Description**

The `t_sync` subroutine synchronizes the data structures managed by the transport library with information from the underlying transport provider. In doing so, if the file descriptor referenced a transport endpoint, the subroutine can convert an uninitialized file descriptor (obtained using the `open` or `dup` subroutines or as a result of a `fork` operation and an `exec` operation) to an initialized transport endpoint, by updating and allocating the necessary library data structures. This subroutine also allows two cooperating processes to synchronize their interaction with a transport provider.

For example, if a process forks a new process and issues an `exec` operation, the new process must issue a `t_sync` to build the private library data structure associated with a transport endpoint and to synchronize the data structure with the relevant provider information.

It is important to remember that the transport provider treats all users of a transport endpoint as a single user. If multiple processes are using the same endpoint, they should coordinate their activities so as not to violate the state of the transport endpoint. The `t_sync` subroutine returns the current state of the transport endpoint to the user, thereby enabling the user to verify the state before taking further action. This
coordination is only valid among cooperating processes; it is possible that a process or an incoming event could change the endpoint’s state after a \texttt{t\_sync} call is issued.

If the transport endpoint is undergoing a state transition when the \texttt{t\_sync} subroutine is called, the subroutine will fail.

**Parameter**

\textit{fd} \quad \text{Specifies the transport endpoint.}

**Valid States**

\texttt{ALL} - except \texttt{T\_UNINIT}.

**Return Values**

On successful completion, the state of the transport endpoint is returned. Otherwise, a value of -1 is returned and \texttt{t\_errno} is set to indicate an error. The state returned is one of the following:

<table>
<thead>
<tr>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>\texttt{T_UNBND}</td>
<td>Unbound.</td>
</tr>
<tr>
<td>\texttt{T_IDLE}</td>
<td>Idle.</td>
</tr>
<tr>
<td>\texttt{T_OUTCON}</td>
<td>Outgoing connection pending.</td>
</tr>
<tr>
<td>\texttt{T_INCON}</td>
<td>Incoming connection pending.</td>
</tr>
<tr>
<td>\texttt{T_DATA}\texttt{TXFER}</td>
<td>Data transfer.</td>
</tr>
<tr>
<td>\texttt{T_OUTREL}</td>
<td>Outgoing orderly release (waiting for an orderly release indication).</td>
</tr>
<tr>
<td>\texttt{T_INREL}</td>
<td>Incoming orderly release (waiting for an orderly release request).</td>
</tr>
</tbody>
</table>

**Error Codes**

On failure, \texttt{t\_errno} is set to one of the following:

<table>
<thead>
<tr>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>\texttt{T_BADF}</td>
<td>The specified file descriptor does not refer to a transport endpoint. This error may be returned when the \textit{fd} parameter has been previously closed or an erroneous number may have been passed to the call.</td>
</tr>
<tr>
<td>\texttt{T_PROTO}</td>
<td>This error indicates that a communication problem has been detected between the X/Open Transport Interface and the transport provider for which there is no other suitable X/Open Transport Interface (\texttt{t_errno}).</td>
</tr>
<tr>
<td>\texttt{T_STATECHNG}</td>
<td>The transport endpoint is undergoing a state change.</td>
</tr>
<tr>
<td>\texttt{T_SYSERR}</td>
<td>A system error has occurred during execution of this function.</td>
</tr>
</tbody>
</table>

**Related Information**

The \texttt{dup} subroutine, \texttt{exec} subroutine, \texttt{fork} subroutine, \texttt{open} subroutine.

---

**t\_unbind Subroutine for X/Open Transport Interface**

**Purpose**

Disable a transport endpoint.

**Library**

X/Open Transport Interface Library (\texttt{libxti.a})
Syntax

```c
#include <xti.h>
int t_unbind (fd);
int fd;
```

Description

The `t_unbind` subroutine disables the transport endpoint which was previously bound by `t_bind`. On completion of this call, no further data or events destined for this transport endpoint are accepted by the transport provider. An endpoint which is disabled by using the `t_unbind` subroutine can be enabled by a subsequent call to the `t_unbind` subroutine.

Parameter

`fd` Specifies the transport endpoint.

Valid States

`T_IDLE`

Return Values

- 0 Successful completion.
- -1 `t_errno` is set to indicate an error.

Errors

On failure, `t_errno` is set to one of the following:

<table>
<thead>
<tr>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>TBADF</td>
<td>The specified file descriptor does not refer to a transport endpoint.</td>
</tr>
<tr>
<td>TOUTSTATE</td>
<td>The subroutine was issued in the wrong sequence.</td>
</tr>
<tr>
<td>TLOOK</td>
<td>An asynchronous event has occurred on this transport endpoint.</td>
</tr>
<tr>
<td>TSYSERR</td>
<td>A system error has occurred during execution of this subroutine.</td>
</tr>
<tr>
<td>TPROTO</td>
<td>This error indicates that a communication problem has been detected between the X/Open Transport Interface and the transport provider for which there is no other suitable X/Open Transport Interface (<code>t_errno</code>).</td>
</tr>
</tbody>
</table>

Related Information

The `t_bind` subroutine.

Options for the X/Open Transport Interface

Options are formatted according to the `t_opthdr` structure as described in "Use of Options for the X/Open Transport Interface". A transport provider compliant to this specification supports none, all, or any subset of the options defined in the following sections: "TCP/IP-Level Options" to "IP-level Options". An implementation may restrict the use of any of these options by offering them only in the privileged or read-only mode.
TCP-Level Options

The protocol level is INET_TCP. For this level, the following table shows the options that are defined.

<table>
<thead>
<tr>
<th>Option Name</th>
<th>Type of Option Value</th>
<th>Legal Option Value</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>TCP_KEEPALIVE</td>
<td>struct t_kpalive</td>
<td>see text following table</td>
<td>check if connections are live</td>
</tr>
<tr>
<td>TCP_MAXSEG</td>
<td>unsigned long</td>
<td>length in octets</td>
<td>get TCP maximum segment size</td>
</tr>
<tr>
<td>TCP_NODELAY</td>
<td>unsigned long</td>
<td>T_YES T_NO</td>
<td>don’t delay send to coalesce packets</td>
</tr>
</tbody>
</table>

**TCP_KEEPALIVE**

If set, a keep-alive timer is activated to monitor idle connections that may no longer exist. If a connection has been idle since the last keep-alive timeout, a keep-alive packet is sent to check if the connection is still alive or broken.

Keep-alive packets are not an explicit feature of TCP, and this practice is not universally accepted. According to RFC 1122:

"a keep-alive mechanism should only be invoked in server applications that might otherwise hang indefinitely and consume resources unnecessarily if a client crashes or aborts a connection during a network failure."

The option value consists of a structure `t_kpalive` declared as:

```c
struct t_kpalive {
    long kp_onoff;
    long kp_timeout;
}
```

The `t_kpalive` fields and the possible values are:

- **kp_onoff**
  - Switches option on or off. Legal values for the field are:
    - T_NO: Switch keep-alive timer off.
    - T_YES: Activate keep-alive timer.
    - T_YES | T_GARBAGE: Activate keep-alive timer and send garbage octet.

  Usually, an implementation should send a keep-alive packet with no data (T_GARBAGE not set). If T_GARBAGE is set, the keep-alive packet contains one garbage octet for compatibility with erroneous TCP implementations.

  An implementation is, however, not obliged to support T_GARBAGE (see RFC 1122). Since the `kp_onoff` value is an absolute requirement, the request 'T_YES | T_GARBAGE' may therefore be rejected.

- **kp_timeout**
  - Specifies the keep-alive timeout in minutes. This field determines the frequency of keep-alive packets being sent, in minutes. The transport user can request the default value by setting the field to T_UNSPEC. The default is implementation-dependent, but at least 120 minutes (see RFC 1122). Legal values for this field are T_UNSPEC and all positive numbers.

  The timeout value is not an absolute requirement. The implementation may pose upper and lower limits to this value. Requests that fall short of the lower limit may be negotiated to the lower limit.

  The use of this option might be restricted to privileged users.

**TCP_MAXSEG**

Used to retrieve the maximum TCP segment size. This option is read-only.
TCP_NODELAY  Under most circumstances, TCP sends data as soon as it is presented. When outstanding data has not yet been acknowledged, it gathers small amounts of output to be sent in a single packet once an acknowledgment is received. For a small number of clients, such as window systems (for example, Enhanced AIXwindows) that send a stream of mouse events which receive no replies, this packetization may cause significant delays. TCP_NODELAY is used to defeat this algorithm. Legal option values are:

T_YES  Do not delay.
T_NO    Delay.

These options are not association-related. The options may be negotiated in all X/Open Transport Interface states except T_UNBIND and T_UNINIT. The options are read-only in the T_UNBIND state. See "The Use of Options for the X/Open Transport Interface" for the differences between association-related options and those options that are not.

Absolute Requirements
A request for TCP_NODELAY and a request to activate TCP_KEEPALIVE is an absolute requirement. TCP_MAXSEG is a read-only option.

UDP-level Options
The protocol level is INET_UDP. The option defined for this level is shown in the following table.

<table>
<thead>
<tr>
<th>Option Name</th>
<th>Type of Option Value</th>
<th>Legal Option Value</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>UDP_CHECKSUM</td>
<td>unsigned long</td>
<td>T_YES/T_NO</td>
<td>checksum computation</td>
</tr>
</tbody>
</table>

UDP_CHECKSUM  Allows disabling and enabling of the UDP checksum computation. The legal values are:

T_YES  Checksum enabled.
T_NO   Checksum disabled.

This option is association-related. It may be negotiated in all XTI states except T_UNBIND and T_UNINIT. It is read-only in state T_UNBND.

If this option is returned with the t_rcvudata subroutine, its value indicates whether a checksum was present in the received datagram or not.

Numerous cases of undetected errors have been reported when applications chose to turn off checksums for efficiency. The advisability of ever turning off the checksum check is very controversial.

Absolute Requirements
A request for this option is an absolute requirement.

IP-level Options
The protocol level is INET_IP. The options defined for this level are listed in the following table.

<table>
<thead>
<tr>
<th>Option Name</th>
<th>Type of Option Value</th>
<th>Legal Option Value</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>IP_BROADCAST</td>
<td>unsigned int</td>
<td>T_YES/T_NO</td>
<td>permit sending of broadcast messages</td>
</tr>
<tr>
<td>IP_DONTROUTE</td>
<td>unsigned int</td>
<td>T_YES/T_NO</td>
<td>just use interface addresses</td>
</tr>
<tr>
<td>IP_OPTIONS</td>
<td>array of unsigned characters</td>
<td>see text</td>
<td>IP per-packet options</td>
</tr>
<tr>
<td>IP_REUSEADDR</td>
<td>unsigned int</td>
<td>T_YES/T_NO</td>
<td>allow local address reuse</td>
</tr>
</tbody>
</table>
### IP-Level Options

<table>
<thead>
<tr>
<th>Option Name</th>
<th>Type of Option Value</th>
<th>Legal Option Value</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>IP_TOS</td>
<td>unsigned char</td>
<td>see text</td>
<td>IP per-packet type of service</td>
</tr>
<tr>
<td>IP_TTL</td>
<td>unsigned char</td>
<td>time in seconds</td>
<td>IP per packet time-to-live</td>
</tr>
</tbody>
</table>

**IF_BROADCAST**
Requests permission to send broadcast datagrams. It was defined to make sure that broadcasts are not generated by mistake. The use of this option is often restricted to privileged users.

**IP_DONTROUTE**
Indicates that outgoing messages should bypass the standard routing facilities. It is mainly used for testing and development.

**IP_OPTIONS**
Sets or retrieves the OPTIONS field of each outgoing (incoming) IP datagram. Its value is a string of octets composed of a number of IP options, whose format matches those defined in the IP specification with one exception: the list of addresses for the source routing options must include the first-hop gateway at the beginning of the list of gateways. The first-hop gateway address will be extracted from the option list and the size adjusted accordingly before use.

The option is disabled if it is specified with "no value," for example, with an option header only.

The `t_connect` (in synchronous mode), `t_listen`, `t_rcvconnect` and `t_rcvudata` subroutines return the OPTIONS field, if any, of the received IP datagram associated with this call. The `t_rcvuderr` subroutine returns the OPTIONS field of the data unit previously sent that produced the error. The `t_optmgmt` subroutine with `T_CURRENT` set retrieves the currently effective IP_OPTIONS that is sent with outgoing datagrams.

Common applications never need this option. It is mainly used for network debugging and control purposes.

**IP_REUSEADDR**
Many TCP implementations do not allow the user to bind more than one transport endpoint to addresses with identical port numbers. If IP_REUSEADDR is set to `T_YES` this restriction is relaxed in the sense that it is now allowed to bind a transport endpoint to an address with a port number and an underspecified internet address ("wild card" address) and further endpoints to addresses with the same port number and (mutually exclusive) fully specified internet addresses.
**IP_TOS**

Sets or retrieves the *type-of-service* field of an outgoing (incoming) IP datagram. This field can be constructed by any OR'ed combination of one of the precedence flags and the type-of-service flags `T_LDELAY`, `T_HITHRPT`, and `T_HIREL`:

- **Precedence:**
  
  These flags specify datagram precedence, allowing senders to indicate the importance of each datagram. They are intended for Department of Defense applications. Legal flags are:
  
  `T_ROUTINE`
  `T_PRIORITY`
  `T_IMMEDIATE`
  `T_FLASH`
  `T OVERRIDEFLASH`
  `T_CRITIC_ECP`
  `T_INETCONTROL`
  `T_NETCONTROL`

  Applications using `IP_TOS` but not the precedence level should use the value `T_ROUTINE` for precedence.

- **Type of service:**
  
  These flags specify the type of service the IP datagram desires. Legal flags are:
  
  `T_NOTOS`
  requests no distinguished type of service
  
  `T_LDELAY`
  requests low delay
  
  `T_HITHRPT`
  requests high throughput
  
  `T_HIREL`
  requests high reliability

  The option value is set using the macro `SET_TOS(prec, tos)` where `prec` is set to one of the precedence flags and `tos` to one or an OR'ed combination of the type-of-service flags. `SET_TOS` returns the option value.

  The `t_connect`, `t_listen`, `t_rcvconnect` and `t_rcvdata` subroutines return the *type-of-service* field of the received IP datagram associated with this call. The `t_rcvuderr` subroutine returns the *type-of-service* field of the data unit previously sent that produced the error.

  The `t_optmgmt` subroutine with `T_CURRENT` set retrieves the currently effective `IP_TOS` value that is sent with outgoing datagrams.

  The requested *type-of-service* cannot be guaranteed. It is a hint to the routing algorithm that helps it choose among various paths to a destination. Note also, that most hosts and gateways in the Internet these days ignore the *type-of-service* field.

**IP_TIL**

This option is used to set the *time-to-live* field in an outgoing IP datagram. It specifies how long, in seconds, the datagram is allowed to remain in the Internet. The *time-to-live* field of an incoming datagram is not returned by any function (since it is not an association-related option).

**IP_OPTIONS** and **IP_TOS** are both association-related options. All other options are not association-related.

**IP_REUSEADDR** may be negotiated in all XTI states except `T_UNINIT`. All other options may be negotiated in all other XTI states except `T_UNBND` and `T_UNINIT`; they are read-only in the state `T_UNBND`.

**Absolute Requirements**

A request for any of these options in an absolute requirement.
Chapter 4. Packet Capture Library Subroutines

The packet capture library contains subroutines that allow users to communicate with the packet capture facility provided by the operating system to read unprocessed network traffic. Applications using these subroutines must be run as root. These subroutines are maintained in the **libpcap.a** library:

- `pcap_close`
- `pcap_compile`
- `pcap_datalink`
- `pcap_dispatch`
- `pcap_dump`
- `pcap_dump_close`
- `pcap_dump_open`
- `pcap_file`
- `pcap_fileno`
- `pcap_geterr`
- `pcap_is_swapped`
- `pcap_lookupdev`
- `pcap_lookupnet`
- `pcap_loop`
- `pcap_major_version`
- `pcap_minor_version`
- `pcap_next`
- `pcap_open_live`
- `pcap_open_offline`
- `pcap_perror`
- `pcap_setfilter`
- `pcap_snapshot`
- `pcap_stats`
- `pcap_strerror`
Appendix. 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

UNIX is a registered trademark of The Open Group in the United States and other countries.

Other company, product, or service names may be the trademarks or service marks of others.
Index

Special characters
_special_getlong subroutine 25
_special_getshort subroutine 26
_special_ll_log subroutine 3
_special_putchar subroutine 27
_special_putchar subroutine 28
/etc/hosts file
  closing 40, 41
  opening 195, 196
  retrieving host entries 66, 67, 68, 70, 72
  setting file markers 195, 196
/etc/networks file
  closing 42, 43, 44
  opening 199, 200
  retrieving network entries 76, 77, 78, 79, 80, 81
  setting file markers 199, 200
/etc/protocols file
  closing 44, 45
  opening 201, 202
  retrieving protocol entries 85, 86, 87, 88, 89, 90
  setting file markers 201, 202
/etc/resolv.conf file
  retrieving host entries 66, 67, 68, 70
  searching for domain names 162
  searching for Internet addresses 162
/etc/services file
  closing 46, 47
  opening 97, 98, 203, 204
  reading 97, 98
  retrieving service entries 92, 93, 94, 96
  setting file markers 203, 204

A
accept subroutine 29
adjmsg utility 245
administrative operations
  providing interface for 314
allocb utility 245, 247
arp subroutines
  arpresolve_common subroutine 30
  arpmactl 31
arpresolve_common subroutine 30
arpmactl subroutine 31
ASCII strings
  converting to Internet addresses 132
asynchronous mode
  sending data 357

B
backq utility 246
bcanput utility 247
bind subroutine 33
bufcall utility 247, 370
byte streams
  placing long byte quantities 27
  placing short byte quantities 28

C
canput utility 249
clients
  server authentication 177
clone device driver 249
code, terminating section 317
communications kernel service subroutines
  res_ninit 165
compressed domain names
  expanding 39
connect subroutine 34
connected sockets
  creating pairs 228
  receiving messages 156
  sending messages 182, 185
connection requests
  accepting 327
  listening 342
  receiving confirmation 350
connectionless mode
  receiving data 355
  receiving error data 356
  sending data 361
converter subroutines
  inet_net_ntop 127
  inet_net_pton 128
  inet_ntop 133
  inet_pton 134
copyb utility 250
copymsg utility 250
CreateloCompletionPort Subroutine 36
current domain names
  returning 65
  setting 194
current host identifiers
  retrieving 73

data
  receiving normal or expedited 349
  sending over connection 357
data blocks
  allocating 255
data link provider, providing interface 252
datamsg utility 251
default domains
  searching names 162
disconnects
  identifying cause and retrieving data 352
  user-initiated requests 359
dlpi STREAMS driver 252
dn_comp subroutine 38
dn_expand subroutine 39
domain names
  compressing 38
drivers
  installing 319
  setting processor levels 317
dupb utility 253
dupmsg utility 253

e
enableok utility 254
dendhostent subroutine 40
dendhostent_r subroutine 41
dendnetent subroutine 42
dendnetent_r subroutine 43
dendnetgrent subroutine 135
dendnetgrent_r subroutine 44
dendprotoent subroutine 44
dendprotoent_r subroutine 45
dendservent subroutine 46
dendservent_r subroutine 47
error logs
  generating messages 325
error messages
  producing 336
esballoc utility 255
ether_aton subroutine 47
ether_hostton subroutine 47
ether_line subroutine 47
ether_ntoa subroutine 47
ether_ntohost subroutine 47
event traces
  generating messages 325

f
file descriptors
  testing 293
flow control
  testing priority band 247
flushband utility 256
flushq utility 256
freeaddrinfo subroutine 61
freeb utility 257
freemsg utility 258
functions
  scheduling calls 366

G
get_auth_method subroutine
  authentication methods
    list of 64
getaddrinfo subroutine 61
getadmin utility 258
getdomainname subroutine 65
gethostbyaddr subroutine 66
gethostbyaddr_r subroutine 67
gethostbyname subroutine 68
gethostbyname_r subroutine 70
gethostent subroutine 72
gethostent_r subroutine 72
gethostid subroutine 73
gethostbyname subroutine 74
getmid utility 259
getmsg system call 259
getnameinfo subroutine 75
getnetbyaddr subroutine 76
getnetbyaddr_r subroutine 77
getnetbyname subroutine 78
getnetbyname_r subroutine 79
getnetent subroutine 80
getnetent_r subroutine 81
getnetgrent subroutine 135
getnetgrent_r subroutine 82
getpeername subroutine 83
getpmmsg system call 262
getprotobyname subroutine 85
getprotobyname_r subroutine 86
getprotobynumber subroutine 87
getprotobynumber_r subroutine 88
getprotoent subroutine 89
getprotoent_r subroutine 90
getq utility 263
GetQueuedCompletionStatus Subroutine 91
getservbyname subroutine 92
getservbyname_r subroutine 93
getservbyport subroutine 94
getservbyport_r subroutine 96
getservent subroutine 97
getservent_r subroutine 98
getsmuxEntrybyidentity subroutine 1
getsmuxEntrybyname subroutine 1
getsockname subroutine 99
getsockopt subroutine 100
group network
  entries in the
    handling 82, 135, 200

H
host machines
  setting names 198
  setting unique identifiers 197
htonl subroutine 108
htonll subroutine 108
htons subroutine 109

I
I_ATMARK operation 276
I_CANPUT operation 276
I_CKBAND operation 277
I_FDINSERT operation 277
I_FIND operation 278
IFlush operation 279
I_FLUSHBAND operation 279
I_GETBAND operation 280
I_GETCLTIME operation 280
I_GETSIG operation 281
I_GRDOPT operation 281
I_LINK operation 281
I_LIST operation 282
I_LOOK operation 283
I_NREAD operation 283
I_PEEK operation 284
I_PLINK operation 284
I_POP operation 285
I_PUNLINK operation 286
I_PUSH operation 286
I_RECVFD operation 287
I_SENDFD operation 288
I_SETCLTIME operation 288
I_SETSIG operation 289
I_SRDOPT operation 290
I_STR operation 291
I_UNLINK operation 292
I/O Completion Port (IOCP) Kernel Extension
  CreateCompletionPort 36
  GetQueuedCompletionStatus 91
  PostQueuedCompletionStatus 150
  ReadFile 155
  WriteFile 243
if_freenameindex subroutine 110
if_indextoname subroutine 111
if_nameindex subroutine 112
if_nametoindex subroutine 112
incoming connections
  limiting backlog 146
inet_addr subroutine 122
inet_lnaof subroutine 125
inet_makeaddr subroutine 126
inet_net_ntop subroutine 127
inet_net_pton subroutine 128
inet_netof subroutine 129
inet_network subroutine 130
inet_ntoa subroutine 132
inet_ntop subroutine 133
inet_pton subroutine 134
inet6_opt_append Subroutine 113
inet6_opt_find Subroutine 114
inet6_opt_finish Subroutine 115
inet6_opt_get_val Subroutine 115
inet6_opt_init Subroutine 116
inet6_opt_next Subroutine 117
inet6_opt_set_val Subroutine 118
inet_rth_add Subroutine 118
inet_rth_getaddr Subroutine 119
inet_rth_init Subroutine 119
inet_rth_reverse Subroutine 120
inet_rth_segments Subroutine 121
inet_rth_space Subroutine 122
initializing logging facility variables 2
initiating SMUX peers 15
innetgr subroutine 135
insq utility 264
Internet addresses
  constructing 126
  converting 122
  converting to ASCII strings 132
  returning network addresses 125
  searching 162
Internet numbers
  converting Internet addresses 122
  converting network addresses 130
ioctl commands 137
ioctl socket control operations 137
isastream function 293
is_addr Subroutine 143
ISO DE library
  extending base subroutines 9
  initializing logging facility variables 2
  logging subroutines 3
isodetailor subroutine 2
K
kvalid_user subroutine
  DCE principal mapping 145
L
library structures
  allocating 329
  freeing 337
linkb utility 293
listen subroutine 146
ll_dbinit subroutine 3
ll_hdinit subroutine 3
ll_log subroutine 3
local host names
  retrieving 74
long byte quantities
  retrieving 25
long integers, converting
  from host byte order 108
  from network byte order 147, 148
  to host byte order 147, 148
  to network byte order 108
M
Management Information Base (MIB)
  registering a section 16
mapping
  Ethernet number 47
memory management subroutines
  getaddrinfo 61
  getnameinfo 75
  if_freenameindex 110
mi_bufcall Utility 294
mi_close_comm Utility 295
mi_next_ptr Utility 296
mi_open_comm Utility 297
MIB variables
  encoding values from 5
  setting variable values 11
minor devices, opening on another driver 249
modules
  comparing names 278
  installing 319
listing all names on stream 282
pushing to top 286

Index 439
modules (continued)
  removing below stream head  285
  retrieving name below stream head  283
  retrieving ID(s)  259
  returning pointer to write queue  377
  returning pointer to  258
  returning pointer to read queue  312
  setting processor level  317
  testing flow control  247
msgdsiz utility  298
multiplexed streams
  connecting  281, 284
  disconnecting  286, 292

N
name servers
  creating packets  163
  creating query messages  163
  retrieving responses  171
  sending queries  171
name2inst subroutine  22
names
  binding to sockets  33
network addresses
  converting  130
  returning  125
  returning network numbers  129
network entries
  retrieving  80, 81
  retrieving by address  76, 77
  retrieving by name  78, 79
network host entries
  retrieving  72
  retrieving by address  66, 67
  retrieving by name  68, 70
network host files
  opening  195, 196
network services library
  supporting transport interface functions  369
next2inst subroutine  22
nextot2inst subroutine  22
noenable utility  254, 299
ntohl subroutine  147
ntoh1 subroutine  148
ntohs subroutine  149

O
o_ subroutines  5
  o_generic subroutine  5
  o_igeneric subroutine  5
  o_integer subroutine  5
  o_ipaddr subroutine  5
  o_number subroutine  5
  o_specific subroutine  5
  o_string subroutine  5
object identifier data structure  7
object tree (OT)
  freeing  14
  MIB list  14
ode2oid subroutine  7
OID
  adjusting the values of entries  9
  converting text strings to  23
  extending number of entries in  9
  manipulating entries  9
OID (object identifier data structure)
  manipulating the  7
  oid_cmp subroutine  7
  oid_cpy subroutine  7
  oid_extend subroutine  9
  oid_free subroutine  7
  oid_normalize subroutine  9
  oid2ode subroutine  7
  oid2ode_aux subroutine  7
  oid2prim subroutine  7
Options  427
OTHERQ utility  299

P
peer entries  1
peer socket names
  retrieving  83
pfmod Packet Filter Module
  upstream data messages, removing  300
PostQueuedCompletionStatus Subroutine  150
prim2oid  7
priority bands
  checking write status  276
  flushing messages  256
protocol data unit (PDU)  12
  sending  17
  sending an open  18
protocol entries
  retrieving  89, 90
  retrieving by name  85, 86
  retrieving by number  87, 88
psap.h file  8
pullupmsg utility  303
putbq utility  304
putctl utility  305
putct1 utility  304
putmmsg system call  306
putnext utility  308
putpmsg system call  308
putq utility  309

Q
qenable utility  311
qreply utility  311
qsize utility  312
queries
  awaiting response  169
  querying  100
queue bands
  flushing messages  279
set_auth_method subroutine
  authentication methods
    list of 193
setdomainname subroutine  194
sethostent subroutine  195
sethostent_r subroutine  196
sethostid subroutine  197
sethostname subroutine  198
setnetent subroutine  199
setnetent_r subroutine  200
setnetgrent subroutine  135
setnetgrent_r subroutine  200
setprotoent subroutine  201
setprotoent_r subroutine  202
setservent subroutine  203
setservent_r subroutine  204
setsockopt subroutine  205
short byte quantities
  retrieving  26
short integers, converting
  from host byte order 109
  to host byte order 149
  from network byte order 109
  to network byte order 149
shutdown subroutine  213
SIGPOLL signal
  informing stream head to issue  289
  returning events of calling process  281
SLP subroutines
  SLPAttrCallback  214
  SLPClose  215
  SLPEscape  216
  SLPFindAttr  217
  SLPFindScopes  218
  SLPFindSrvs  218
  SLPFindSrvTypes  219
  SLPFree  220
  SLPGetProperty  221
  SLPOpen  222
  SLPParseSrvURL  223
  SLPServiceCallback  223
  SLPGetURLCallback  224
  SLPUnescape  225
SLPAttrCallback subroutine  214
SLPClose subroutine  215
SLPEscape subroutine  216
SLPFindAttr subroutine  217
SLPFindScopes subroutine  218
SLPFindSrvs subroutine  218
SLPFindSrvTypes subroutine  219
SLPFree subroutine  220
SLPGetProperty subroutine  221
SLPOpen subroutine  222
SLPParseSrvURL subroutine  223
SLPServiceCallback subroutine  223
SLPGetURLCallback subroutine  224
SLPUnescape subroutine  225
SMUX
  communicating with the SNMP agent  17
  communicating with the snmpd daemon  15
  ending SNMP communications  12
SMUX (continued)

- Initiating transmission control protocol (TCP) 15
- Peer responsibility level 16
- Reading a MIB variable structure into 10
- Reading the smux_errno variable 13
- Registering an MIB tree for 16
- Retrieving peer entries 1
- Sending an open PDU 18
- Sending traps to SNMP 20
- Setting debug level for subroutines 15
- Unregistered trees 14
- Waiting for a message 21

### Smux Subroutines

- smux_close subroutine 12
- smux_error subroutine 13
- smux_free_tree subroutine 14
- smux_init subroutine 15
- smux_register subroutine 16
- smux_response subroutine 17
- smux_simple_open subroutine 18
- smux_trap subroutine 20
- smux_wait subroutine 21
- smux.h file 13
- SNMP multiplexing peers 1
- snmpd daemon
  - Incoming messages alert 18
- snmpd.peers file 1
- Socket connections
  - Accepting 29
  - Listening 146
- Socket names
  - Retrieving 99
- Socket options
  - Setting 205
- Socket receive operations
  - Disabling 213
- Socket send operations
  - Disabling 213
- Socket subroutine 226
- Socket subroutines
  - Freeaddrinfo subroutine 61
  - if_indexname subroutine 111
  - if_nameindex subroutine 112
  - if_nameindex subroutine 112
  - inet6_opt_append 113
  - inet6_opt_find 114
  - inet6_opt_finish 115
  - inet6_opt_get_val 115
  - inet6_opt_init 116
  - inet6_opt_next 117
  - inet6_opt_set_val 118
  - inet6_rth_add 118
  - inet6_rth_getaddr 119
  - inet6_rth_init 119
  - inet6_rth_reverse 120
  - inet6_rth_segments 121
  - inet6_rth_space 122
  - rcmd_af 153
  - reexec_af 174
  - rresvport_af 176
  - Socketpair subroutine 228

### Sockets

- Connecting 34
- Creating 226
- Initiating TCP for SMUX peers 15
- Managing 242
- Retrieving with privileged addresses 175

### Sockets Kernel Service Subroutines

- Accept 29
- Bind 33
- Connect 34
- Dn_comp 38
- Getdomainname 65
- Gethostid 73
- Gethostname 74
- Getpeername 83
- Getsockname 99
- Getsockopt 100
- Listen 146
- Recv 156
- Recvfrom 158
- Rcvmsg 160
- Send 182
- Sendmsg 184
- Sendto 186
- Setdomainname 194
- Sethostid 197
- Sethostname 198
- Setssockopt 205
- Shutdown 213
- Socket 226
- Socketpair 228
- Sockets Messages
  - Receiving from connected sockets 156
  - Receiving from sockets 158, 160
  - Sending through any socket 185

### Sockets Network Library Subroutines

- _getlong 25
- _getshort 26
- _putlong 27
- _putshort 28
- Dn_expand 39
- Endhostent 40
- Endhostent_r 41
- Endnetent 42
- Endnetent_r 43
- Endnetgrent_r 44
- Endprotoent 44
- Endprotoent_r 45
- Endservent 46
- Endservent_r 47
- Gethostbyaddr 66
- Gethostbyaddr_r 67
- Gethostbyname 68
- Gethostbyname_r 70
- Gethostent 72
- Gethostent_r 72
- Getnetbyaddr 76
- Getnetbyaddr_r 77
- Getnetbyname 78
- Getnetbyname_r 79
- Getnetent 80
sockets network library subroutines (continued)
  getnetent_r 81
  getprotobyname 85
  getprotobyname_r 86
  getprotobynumber 87
  getprotobynumber_r 88
  getprotoent 89
  getprotoent_r 90
  getservbyname 92
  getservbyname_r 93
  getservbyport 94
  getservbyport_r 96
  getservent 97
  getservent_r 98
  htonl 108
  htonl 108
  htons 109
  inet_addr 122
  inet_lnaof 125
  inet_makeaddr 126
  inet_network 130
  inet_ntoa 132
  ntohs 147
  ntohs 148
  ntohs 149
  rcmd 151
  res_init 162
  res_mkquery 163
  res_query 167
  res_search 169
  res_send 171
  rexec 172
  rresvport 175
  ruserok 177
  sethostent 195
  sethostent_r 196
  setnetent 199
  setnetent_r 200
  setprotoent 201
  setprotoent_r 202
  setservent 203
  setservent_r 204
sockets-based protocols, providing access 377
splice subroutine 242
splstr utility 317
splx utility 317
sprintoid subroutine 7
srv utility 317
  messages queued 317
str_install utility 319
str2oid subroutine 7
stream heads
  checking queue for message 277
  counting data bytes in first message 283
  issuing SIGPOLL signal 289
  removing modules 285
  retrieving messages 284
  retrieving module names 283
  returning set delay time 283
  setting delay 288
streamio operations
  I_ATMARK 276
  I_CANPUT 276
  I_CKBAND 277
  I_FDINSERT 277
  I_FIND 278
  I_FLUSH 279
  I_FLUSHBAND 279
  I_GETBAND 280
  I_GETCLTIME 280
  I_GETSIG 281
  I_GRDOPT 281
  I_LINK 281
  I_LIST 282
  I_LOOK 283
  I_NREAD 283
  I_PEEK 284
  I_PLINK 284
  I_POP 285
  I_PUNLINK 284
  I_PUSH 286
  I_RECVFD 287
  I_SENDFD 288
  I_SETCLTIME 288
  I_SETSIG 289
  I_SRDOPT 290
  I_STR 291
  I_UNLINK 292
STREAMS
  mi_bufcall Utility 294
  mi_close_comm Utility 295
  mi_next_ptr Utility 296
  mi_open_comm Utility 297
  performing control functions 323
  unweldq Utility 372
  weldq Utility 376
STREAMS buffers
  checking availability 365
STREAMS device drivers
  clone 249
  sad 314
STREAMS drivers
  dlpi 252
  xtiso 377
STREAMS message blocks
  copying 250
  duplicating descriptors 253
  freeing 257, 258
  removing from head of message 370
  removing from messages 313
STREAMS messages
  allocating 255
  allocating data blocks 245
  checking buffer availability 365
  checking markings 276
  concatenating 293
  concatenating and aligning data bytes 303
  constructing internal ioctl 291
  converting streamio operations 367
  counting data bytes 283
  creating control 304, 305
STREAMS messages (continued)
creating, adding information, and sending downstream 277
determining whether data message 251
duplicating 253
flushing in given priority band 256
generating error-logging and event-tracing 325
getting next from queue 263
generating next priority 262
getting off stream 259
passing to next queue 308
placing in queue 264
putting on queue 309
removing from queue 313
retrieving file descriptors 287
retrieving without removing 284
returning number of data bytes 298
returning number on queue 312
returning priority band of first on queue 280
returning to beginning of queue 304
sending 306
sending in reverse direction 311
sending priority 308
sending to stream head at other end of stream pipe 288
trimming bytes 245
STREAMS modules
timod 367
tirdwr 369
STREAMS queues
checking for messages 277
counting data bytes in first message 283
enabling 311
flushing 256
flushing input or output 279
getting next message 263
obtaining information 326
passing message to next 308
preventing scheduling 299
putting messages on 309
retrieving pointer to write queue 377
returning message to beginning 304
returning number of messages 312
returning pointer to mate 299
returning pointer to preceding 246
returning pointer to read queue 312
returning priority band of first message 280
returning to beginning of queue 304
returning number on queue 312
returning priority band of first on queue 280
returning to beginning of queue 304
returning number of messages 312
returning pointer to mate 299
returning pointer to preceding 246
returning pointer to read queue 312
returning priority band of first message 280
returning to beginning of queue 304
testing for space 249
STREAMS subroutines
isastream 293
t_accept 327
t_alloc 329
t_bind 331
t_close 333
t_connect 334
t_error 336
t_free 337
t_getinfo 338
t_getsockopt 341
t_listen 342
STREAMS subroutines (continued)
t_look 344
t_open 345
t_optmgmt 347
t_rcv 349
t_rcvconnect 350
t_rcvdis 352
t_rcvrel 354
t_rcvudata 355
t_rcvuderr 356
t_snd 357
t_snddis 359
t_sndrel 360
t_sndudata 361
t_sync 363
t_unbind 364
STREAMS system calls
getmsg 259
getpmsg 262
putmsg 306
putpmsg 308
STREAMS utilities
adjmsg 245
allocb 245
backq 246
canput 247
bufcall 247
canput 249
copyb 250
copymsg 250
datamsg 251
dupb 253
dupmsg 253
enableok 254
esballoc 255
flushband 256
flushq 256
freeb 257
freemsg 258
getadmin 258
getmid 259
getq 263
insq 264
linkb 293
msgdszize 298
noenable 299
OTHERQ 299
pullupmsg 303
putbq 304
putct1 304
putnext 308
putq 309
qenable 311
qreply 311
qsize 312
RD 312
rmvb 313
rmvq 313
splstr 317
spx 317
STREAMS utilities (continued)
   str_install 319
   strlog 325
   strqget 326
   testb 365
   timeout 366
   unbufcall 370
   unlinkb 370
   untimeout 371
   WR 377
string conversions 23
strlog utility 325
strqget utility 326
synchronous mode
   sending data 357
T
   t_accept subroutine 327
   t_accept Subroutine 379
   t_alloc subroutine 329
   t_alloc Subroutine 381
   t_bind subroutine 331
   t_bind Subroutine 383
   t_close subroutine 333
   t_close Subroutine 386
   t_connect subroutine 334
   t_connect Subroutine 387
   t_error subroutine 336
   t_error Subroutine 389
   t_free subroutine 337
   t_free Subroutine 390
   t_getinfo subroutine 338
   t_getinfo Subroutine 392
   t_getprotaddr Subroutine 394
   t_getstate subroutine 341
   t_getstate Subroutine 395
   t_listen subroutine 342
   t_listen Subroutine 396
   t_look subroutine 344
   t_look Subroutine 398
   t_open subroutine 345
   t_open Subroutine 399
   t_opthdr 427
   t_optmgmt subroutine 347
   t_optmgmt Subroutine 403
   t_rcv subroutine 349
   t_rcv Subroutine 409
   t_rcvconnect subroutine 350
   t_rcvconnect Subroutine 411
   t_rcvdis subroutine 352
   t_rcvdis Subroutine 413
   t_rcvrel subroutine 354
   t_rcvrel Subroutine 414
   t_rcvudata subroutine 355
   t_rcvudata Subroutine 415
   t_rcvuderr Subroutine 417
   t_rdvuderr subroutine 356
   t_snd subroutine 357
   t_snd Subroutine 418
   t_snddis subroutine 359
   t_snddis Subroutine 420
   t_sndrel subroutine 360
   t_sndrel Subroutine 421
   t_sndudata subroutine 361
   t_sndudata Subroutine 423
   t_strerror Subroutine 424
   t_sync subroutine 363
   t_sync Subroutine 425
   t_unbind subroutine 364
   t_unbind Subroutine 426
   testb utility 365
   text2inst subroutine 22
   text2obj subroutine 23
   tirdwr module 369
   transport connections, initiating release 360
   transport endpoints
      binding addresses 331
      closing 333
      disabling 364
      establishing 345
      establishing connection 334
      examining current events 344
      getting current states 341
      managing options 347
   transport interfaces
      converting streamio operations into messages 367
      supporting network services library functions 369
   transport library, synchronizing data 363
   transport protocols
      getting service information 338
   traps 20
U
   unbufcall utility 370
   unconnected sockets
      receiving messages 158
      sending messages 185, 186
   unique identifiers
      retrieving 73
   unlinkb utility 370
   untimeout utility 371
   unweldq Utility 372
V
   variable bindings 5
   variable initialization 2
W
   wantio utility 373
   wantmsg Utility 374
   weldq Utility 376
   WR utility 377
   write queue
      retrieve a pointer to 377
   WriteFile Subroutine 243
xtiso STREAMS driver 377
Readers’ Comments — We’d Like to Hear from You

AIX 5L Version 5.3
Technical Reference: Communications, Volume 2

Publication No. SC23-4916-03

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>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</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.