Bull

Technical Reference
Base Operating System and Extensions

Volume 1/2

AIX

ORDER REFERENCE
86 A2 81AP 05

Bull

Technical Reference
Base Operating System and Extensions

Volume 1/2

AIX

Software

February 1999

BULL ELECTRONICS ANGERS
CEDOC
34 Rue du Nid de Pie – BP 428
49004 ANGERS CEDEX 01
FRANCE

ORDER REFERENCE
86 A2 81AP 05

Printed in France

Suggestions and criticisms concerning the form, content, and presentation of this book are invited. A form is provided at the end of this book for this purpose.

To order additional copies of this book or other Bull Technical Publications, you are invited to use the Ordering Form also provided at the end of this book.

Trademarks and Acknowledgements

We acknowledge the right of proprietors of trademarks mentioned in this book.

AIX® is a registered trademark of International Business Machines Corporation, and is being used under licence.

UNIX is a registered trademark in the United States of America and other countries licensed exclusively through the Open Group.

Year 2000

The product documented is this manual is Year 2000 Ready.

Table of Contents

About This Book

This book provides information on *Technical Reference, Volumes 1 and 2: Base Operating System and Extensions*. Topics covered provide information on application programming interfaces to the Advanced Interactive Executive Operating System (referred to in this text as AIX).

These two books are part of the six–volume technical reference set, *AIX Technical Reference*, 86 A2 81AP to 86 A2 91AP, which provides information on system calls, kernel extension calls, and subroutines in the following volumes:

- *Base Operating System and Extensions, Volumes 1 and 2* provide information on system calls, subroutines, functions, macros, and statements associated with AIX base operating system runtime services.
- *Communications, Volumes 1 and 2* provide information on entry points, functions, system calls, subroutines, and operations related to communications services.
- *Kernel and Subsystems, Volumes 1 and 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.

Who Should Use This Book

This book is intended for experienced C programmers. To use the book effectively, you should be familiar with AIX or UNIX System V commands, system calls, subroutines, file formats, and special files.

Before You Begin

Before you begin the tasks discussed in this book, you should see *AIX 4.3 System Management Guide: Operating System and Devices* and *AIX 4.3 System Management Guide: Communications and Networks* for more information.

How to Use This Book

Overview of Contents

This book contains the following chapters and appendixes:

- *Base Operating System and Extension Technical Reference, Volumes 1 and 2* contain alphabetically arranged system calls (called subroutines), subroutines, functions, macros, and statements on Base Operating System Runtime (BOS) Services.
- Volume 2 also contains alphabetically arranged Fortran Basic Linear Algebra Subroutines (BLAS).

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.

ISO 9000

ISO 9000 registered quality systems were used in the development and manufacturing of this product.

AIX 32–Bit Support for the X/Open UNIX95 Specification

Beginning with AIX Version 4.2, the operating system is designed to support the X/Open UNIX95 Specification for portability of UNIX–based operating systems. Many new interfaces, and some current ones, have been added or enhanced to meet this specification. Beginning with Version 4.2, AIX is even more open and portable for applications.

At the same time, compatibility with previous AIX releases is preserved. This is accomplished by the creation of a new environment variable, which can be used to set the system environment on a per–system, per–user, or per–process basis.

To determine the proper way to develop a UNIX95–portable application, you may need to refer to the X/Open UNIX95 Specification, which can be obtained on a CD–ROM by ordering the printed copy of *AIX Commands Reference*, order number 86 A2 38JX to 86 A2 43JX, or by ordering *Go Solo: How to Implement and Go Solo with the Single Unix Specification*, a book which includes the X/Open UNIX95 Specification on a CD–ROM.

AIX 32–Bit and 64–Bit Support for the UNIX98 Specification

Beginning with AIX Version 4.3, the operating system is designed to support the X/Open UNIX98 Specification for portability of UNIX–based operating systems. Many new interfaces, and some current ones, have been added or enhanced to meet this specification. Making AIX Version 4.3 even more open and portable for applications.

At the same time, compatibility with previous AIX releases is preserved. This is accomplished by the creation of a new environment variable, which can be used to set the system environment on a per–system, per–user, or per–process basis.

To determine the proper way to develop a UNIX98–portable application, you may need to refer to the X/Open UNIX98 Specification, which can be obtained on a CD–ROM by ordering the printed copy of *AIX Commands Reference*, order number 86 A2 38JX to 86 A2 43JX, or by ordering *Go Solo: How to Implement and Go Solo with the Single Unix Specification*, order number SR28–5705, a book which includes the X/Open UNIX98 Specification on a CD–ROM.

Related Publications

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

- *AIX General Programming Concepts : Writing and Debugging Programs*, Order Number 86 A2 34JX.
- *AIX Communications Programming Concepts*, Order Number 86 A2 35JX.
- *AIX Kernel Extensions and Device Support Programming Concepts*, Order Number 86 A2 36JX.

- *AIX Files Reference*, Order Number 86 A2 79AP.
- *AIX Version 4.3 Problem Solving Guide and Reference*, Order Number 86 A2 32JX.
- *Hardware Technical Information-General Architectures*, Order Number 86 A1 09WD.

Ordering Publications

You can order publications from your sales representative or from your point of sale.

To order additional copies of this book, use the following order numbers:

- *AIX Technical Reference, Volume 1: Base Operating System and Extensions* Order Number 86 A2 81AP.
- *AIX Technical Reference, Volume 2: Base Operating System and Extensions*, Order Number 86 A2 82AP.

Use *AIX and Related Products Documentation Overview*, order number 86 A2 71WE, for information on related publications and how to obtain them.

Base Operating System (BOS) Runtime Services (A–P)

a64l or l64a Subroutine

Purpose

Converts between long integers and base–64 ASCII strings.

Library

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

Syntax

```
#include <stdlib.h>

long a64l (String)
char *String;

char *l64a (LongInteger )
long LongInteger;
```

Description

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

The following characters are used to represent digits:

.	Represents 0.
/	Represents 1.
0 –9	Represents the numbers 2–11.
A–Z	Represents the numbers 12–37.
a–z	Represents the numbers 38–63.

Parameters

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

Return Values

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

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

The value returned by the **l64a** subroutine is a pointer into a static buffer, the contents of which are overwritten by each call.

If the **String* parameter is a null string, the **a64l** subroutine returns a value of 0L.

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

Implementation Specifics

These a64l and l64a subroutines are part of Base Operating System (BOS) Runtime.

Related Information

Subroutines Overview in *AIX General Programming Concepts : Writing and Debugging Programs*.

List of Multithread Subroutines in *AIX General Programming Concepts : Writing and Debugging Programs*.

abort Subroutine

Purpose

Sends a **SIGIOT** signal to end the current process.

Library

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

Syntax

```
#include <stdlib.h>

int abort (void)
```

Description

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

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

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

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

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

Return Values

The **abort** subroutine does not return a value.

Implementation Specifics

This subroutine is part of Base Operating System (BOS) Runtime.

Related Information

The **exit**, **atexit,** or **_exit** subroutine, **fclose** subroutine, **kill**, or **killpg** subroutine, **raise** subroutine, **sigaction**, **sigvec**, **signal** subroutine, **wait** or **waidtpid** subroutine.

The **dbx** command.

Subroutines Overview in *AIX General Programming Concepts : Writing and Debugging Programs*.

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

Purpose

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

Library

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

Syntax

```
#include <stdlib.h>

int abs ( i  )
int i;

#include <stdlib.h>

long labs ( i )
long i;

#include <stdlib.h>

div_t div (Numerator, Denominator)
int Numerator: Denominator;

#include <stdlib.h>

void imul_dbl (i, j, Result)
long i, j;
long *Result;

#include <stdlib.h>

ldiv_t ldiv (Numerator, Denominator)
long Numerator: Denominator;

#include <stdlib.h>

void umul_dbl (i, j, Result)
unsigned long i, j;
unsigned long *Result;

#include <stdlib.h>

long long int llabs(i)
long long int i;

#include <stdlib.h>

lldiv_t lldiv (Numerator, Denominator)
long long int Numerator, Denominator;
```

Description

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

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

The **div** subroutine computes the quotient and remainder of the division of the number represented by the *Numerator* parameter by that specified by the *Denominator* parameter. If the division is inexact, the sign of the resulting quotient is that of the algebraic quotient, and the magnitude of the resulting quotient is the largest integer less than the magnitude of the algebraic quotient. If the result cannot be represented (for example, if the denominator is 0), the behavior is undefined.

The **labs** and **ldiv** subroutines are included for compatibility with the ANSI C library, and accept long integers as parameters, rather than as integers.

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

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

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

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

Parameters

i	Specifies, for the **abs** subroutine, some integer; for **labs** and **imul_dbl,** some long integer; for the **umul_dbl** subroutine, some unsigned long integer; for the **llabs** subroutine, some long long integer.
Numerator	Specifies, for the **div** subroutine, some integer; for the **ldiv** subroutine, some long integer; for **lldiv**, some long long integer.
j	Specifies, for the **imul_dbl** subroutine, some long integer; for the **umul_dbl** subroutine, some unsigned long integer.
Denominator	Specifies, for the **div** subroutine, some integer; for the **ldiv** subroutine, some long integer; for **lldiv**, some long long integer.
Result	Specifies, for the **imul_dbl** subroutine, some long integer; for the **umul_dbl** subroutine, some unsigned long integer.

Return Values

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

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

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

access, accessx, or faccessx Subroutine

Purpose

Determines the accessibility of a file.

Library

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

Syntax

```
#include <unistd.h>


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

int accessx (PathName, Mode, Who)
char *PathName;
int Mode, Who;

int faccessx (FileDescriptor, Mode, Who)
int FileDescriptor;
int Mode, Who;
```

Description

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

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

Parameters

PathName	Specifies the path name of the file. If the *PathName* parameter refers to a symbolic link, the **access** subroutine returns information about the file pointed to by the symbolic link.
FileDescriptor	Specifies the file descriptor of an open file.
Mode	Specifies the access modes to be checked. This parameter is a bit mask containing 0 or more of the following values, which are defined in the **sys/access.h** file:

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

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

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

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

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

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

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

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

Return Values

If the requested access is permitted, the **access**, **accessx**, and **faccessx** subroutines return a value of 0. If the requested access is not permitted or the function call fails, a value of –1 is returned and the **errno** global variable is set to indicate the error.

Error Codes

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

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

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

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

ENOENT A component of the *PathName* does not exist or the process has the **disallow truncation** attribute set.

ENOTDIR A component of the *PathName* is not a directory.

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

ENOENT The named file does not exist.

ENOENT The *PathName* parameter was null.

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

ENAMETOOLONG A component of the *PathName* parameter exceeded 255 characters or the entire *PathName* parameter exceeded 1023 characters.

The **faccessx** subroutine fails if the following is true:

EBADF The value of the *FileDescriptor* parameter is not valid.

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

EIO	An I/O error occurred during the operation.
EACCES	The file protection does not allow the requested access.
EROFS	Write access is requested for a file on a read–only file system.

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

ETIMEDOUT	The connection timed out.
ETXTBSY	Write access is requested for a shared text file that is being executed.
EINVAL	The value of the *Mode* argument is invalid.

Implementation Specifics

These subroutines are part of Base Operating System (BOS) Runtime.

Related Information

The **acl_get** subroutine, **chacl** subroutine, **statx** subroutine, **statacl** subroutine.

The **aclget** command, **aclput** command, **chmod** command, **chown** command.

Files, Directories, and File Systems for Programmers in *AIX General Programming Concepts : Writing and Debugging Programs*.

acct Subroutine

Purpose

Enables and disables process accounting.

Library

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

Syntax

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

Description

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

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

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

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

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

Parameters

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

Return Values

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

Error Codes

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

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

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

ETIMEDOUT	The connection timed out.

acl_chg or acl_fchg Subroutine

Purpose

Changes the access control information on a file.

Library

Security Library (**libc.a**)

Syntax

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

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

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

Description

The **acl_chg** and **acl_fchg** subroutines modify the access control information of a specified file.

Parameters

FileDescriptor
: Specifies the file descriptor of an open file.

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

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

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

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

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

  R_ACC Allows read permission.

  W_ACC Allows write permission.

  X_ACC Allows execute or search permission.

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

Who Specifies which entries in the ACL are affected. This parameter takes one of the following values:

ACC_OBJ_OWNER Changes the owner entry in the base ACL.

ACC_OBJ_GROUP Changes the group entry in the base ACL.

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

ACC_ALL Changes all entries in the ACL.

Return Values

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

Error Codes

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

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

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

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

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

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

ENOENT The *Path* parameter was null.

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

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

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

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

EBADF The *FileDescriptor* value is not valid.

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

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

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

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

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

EIO	An I/O error occurred during the operation.
EPERM	The effective user ID does not match the ID of the owner of the file and the invoker does not have root user authority.

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

ETIMEDOUT	The connection timed out.

Implementation Specifics

These subroutines are part of Base Operating System (BOS) Runtime.

Related Information

The **acl_get** subroutine, **acl_put** subroutine, **acl_set** subroutine, **chacl** subroutine, **chmod** subroutine, **stat** subroutine, **statacl** subroutine.

The **aclget** command, **aclput** command, **chmod** command.

List of Security and Auditing Subroutines and Subroutines Overview in *AIX General Programming Concepts : Writing and Debugging Programs*.

acl_get or acl_fget Subroutine

Purpose

Gets the access control information of a file.

Library

Security Library (**libc.a**)

Syntax

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

char *acl_get (Path)
char *Path;

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

Description

The **acl_get** and **acl_fget** subroutines retrieve the access control information for a file system object. This information is returned in a buffer pointed to by the return value. The structure of the data in this buffer is unspecified. The value returned by these subroutines should be used only as an argument to the **acl_put** or **acl_fput** subroutines to copy or restore the access control information.

Parameters

Path	Specifies the path name of the file.
FileDescriptor	Specifies the file descriptor of an open file.

Return Values

On successful completion, the **acl_get** and **acl_fget** subroutines return a pointer to the buffer containing the access control information. Otherwise, a null pointer is returned and the **errno** global variable is set to indicate the error.

Error Codes

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

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

The **acl_fget** subroutine fails if the following is true:

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

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

EIO	An I/O error occurred during the operation.

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

ETIMEDOUT	The connection timed out.

Security

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

Implementation Specifics

These subroutines are part of Base Operating System (BOS) Runtime.

Related Information

The **acl_chg** or **acl_fchg** subroutine, **acl_put** or **acl_fput** subroutine, **acl_set** or **acl_fset** subroutine, **chacl** subroutine, **chmod** subroutine, **stat** subroutine, **statacl** subroutine.

The **aclget** command, **aclput** command, **chmod** command.

List of Security and Auditing Subroutines and Subroutines Overview in *AIX General Programming Concepts : Writing and Debugging Programs*.

acl_put or acl_fput Subroutine

Purpose

Sets the access control information of a file.

Library

Security Library (**libc.a**)

Syntax

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

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

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

Description

The **acl_put** and **acl_fput** subroutines set the access control information of a file system object. This information is contained in a buffer returned by a call to the **acl_get** or **acl_fget** subroutine. The structure of the data in this buffer is unspecified. However, the entire Access Control List (ACL) for a file cannot exceed one memory page (4096 bytes) in size.

Parameters

Path	Specifies the path name of a file.
FileDescriptor	Specifies the file descriptor of an open file.
Access	Specifies a pointer to the buffer containing the access control information.
Free	Specifies whether the buffer space is to be deallocated. The following values are valid:

0	Space is not deallocated.
1	Space is deallocated.

Return Values

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

Error Codes

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

EACCES	Search permission is denied on a component of the *Path* prefix.
EFAULT	The *Path* parameter points to a location outside of the allocated address space of the process.
ELOOP	Too many symbolic links were encountered in translating the *Path* parameter.

ENAMETOOLONG	A component of the *Path* parameter exceeded 255 characters, or the entire *Path* parameter exceeded 1023 characters.
ENOENT	A component of the *Path* does not exist or has the **disallow truncation** attribute (see the **ulimit** subroutine).
ENOENT	The *Path* parameter was null.
ENOENT	A symbolic link was named, but the file to which it refers does not exist.
ENOTDIR	A component of the *Path* prefix is not a directory.
ESTALE	The process' root or current directory is located in a virtual file system that has been unmounted.

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

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

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

EINVAL	The *Access* parameter does not point to a valid access control buffer.
EINVAL	The *Free* parameter is not 0 or 1.
EIO	An I/O error occurred during the operation.
EROFS	The named file resides on a read–only file system.

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

ETIMEDOUT	The connection timed out.

Security

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

Auditing Events:

Event	**Information**
chacl	*Path*
fchacl	*FileDescriptor*

Implementation Specifics

These subroutines are part of Base Operating System (BOS) Runtime.

Related Information

The **acl_chg** subroutine, **acl_get** subroutine, **acl_set** subroutine, **chacl** subroutine, **chmod** subroutine, **stat** subroutine, **statacl** subroutine.

The **aclget** command, **aclput** command, **chmod** command.

List of Security and Auditing Subroutines and Subroutines Overview in *AIX General Programming Concepts : Writing and Debugging Programs*.

acl_set or acl_fset Subroutine

Purpose

Sets the access control information of a file.

Library

Security Library (**libc.a**)

Syntax

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

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

int acl_fset (FileDescriptor, OwnerMode, GroupMode, DefaultMode)
int *FileDescriptor;
int OwnerMode;
int GroupMode;
int DefaultMode;
```

Description

The **acl_set** and **acl_fset** subroutines set the base entries of the Access Control List (ACL) of the file. All other entries are discarded. Other access control attributes are left unchanged.

Parameters

DefaultMode	Specifies the access permissions for the default class.
FileDescriptor	Specifies the file descriptor of an open file.
GroupMode	Specifies the access permissions for the group of the file.
OwnerMode	Specifies the access permissions for the owner of the file.
Path	Specifies a pointer to the path name of a file.

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

R_ACC	Authorize read permission.
W_ACC	Authorize write permission.
X_ACC	Authorize execute or search permission.

Return Values

Upon successful completion, the **acl_set** and **acl_fset** subroutines return the value 0. Otherwise, the value –1 is returned and the **errno** global variable is set to indicate the error.

Error Codes

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

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

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

EBADF	The file descriptor *FileDescriptor* is not valid.

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

EIO	An I/O error occurred during the operation.
EPERM	The effective user ID does not match the ID of the owner of the file and the invoker does not have root user authority.
EROFS	The named file resides on a read–only file system.

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

ETIMEDOUT	The connection timed out.

Security

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

Auditing Events:

Event	Information
chacl	*Path*
fchacl	*FileDescriptor*

Implementation Specifics

These subroutines are part of Base Operating System (BOS) Runtime.

Related Information

The **acl_chg** subroutine, **acl_get** subroutine, **acl_put** subroutine, **chacl** subroutine, **chmod** subroutine, **stat** subroutine, **statacl** subroutine.

The **aclget** command, **aclput** command, **chmod** command.

List of Security and Auditing Subroutines and Subroutines Overview in *AIX General Programming Concepts : Writing and Debugging Programs*.

addssys Subroutine

Purpose

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

Library

System Resource Controller Library (**libsrc.a**)

Syntax

```
#include <sys/srcobj.h>
#include <spc.h>


int addssys (SRCSubsystem )
struct SRCsubsys *SRCSubsystem;
```

Description

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

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

Parameters

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

Return Values

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

Error Codes

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

SRC_BADFSIG	Invalid stop force signal.
SRC_BADNSIG	Invalid stop normal signal.
SRC_CMDARG2BIG	Command arguments too long.
SRC_GRPNAM2BIG	Group name too long.
SRC_NOCONTACT	Contact not signal, sockets, or message queue.
SRC_NONAME	No subsystem name specified.
SRC_NOPATH	No subsystem path specified.
SRC_PATH2BIG	Subsystem path too long.
SRC_STDERR2BIG	stderr path too long.
SRC_STDIN2BIG	stdin path too long.
SRC_STDOUT2BIG	stdout path too long.

SRC_SUBEXIST	New subsystem name already on file.
SRC_SUBSYS2BIG	Subsystem name too long.
SRC_SYNEXIST	New subsystem synonym name already on file.
SRC_SYN2BIG	Synonym name too long.

Security

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

SET_PROC_AUDIT

Files Accessed:

Mode	File
644	**/etc/objrepos/SRCsubsys**

Auditing Events:

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

Event	Information
SRC_addssys	Lists the SRCsubsys records added.

See "How to Set Up Auditing" in *AIX 4.3 System Management Guide: Operating System and Devices* for details about selecting and grouping audit events, and configuring audit event data collection.

Implementation Specifics

This subroutine is part of Base Operating System (BOS) Runtime.

Files

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

Related Information

The **chssys** subroutine, **defssys** subroutine, **delssys** subroutine.

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

Auditing Overview and System Resource Controller Overview in *AIX 4.3 System Management Guide: Operating System and Devices*.

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

adjtime Subroutine

Purpose

Corrects the time to allow synchronization of the system clock.

Library

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

Syntax

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

Description

The **adjtime** subroutine makes small adjustments to the system time, as returned by the **gettimeofday** subroutine, advancing or retarding it by the time specified by the *Delta* parameter of the **timeval** structure. If the *Delta* parameter is negative, the clock is slowed down by incrementing it more slowly than normal until the correction is complete. If the *Delta* parameter is positive, a larger increment than normal is used. The skew used to perform the correction is generally a fraction of one percent. Thus, the time is always a monotonically increasing function, unless the clock is read more than 100 times per second. A time correction from an earlier call to the **adjtime** subroutine may not be finished when the **adjtime** subroutine is called again. If the *Olddelta* parameter is nonzero, then the structure pointed to will contain, upon return, the number of microseconds still to be corrected from the earlier call.

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

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

Parameters

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

Return Values

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

Error Codes

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

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

aio_cancel or aio_cancel64 Subroutine

Purpose

Cancels one or more outstanding asynchronous I/O requests.

Library

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

Syntax

```
#include <aio.h>

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

aio_cancel64 (FileDescriptor, aiocbp)
int FileDescriptor;
struct aiocb64 *aiocbp;
```

Description

The **aio_cancel** subroutine attempts to cancel one or more outstanding asynchronous I/O requests issued on the file associated with the *FileDescriptor* parameter. If the pointer to the **aio control block (aiocb)** structure (the *aiocbp* parameter) is not null, then an attempt is made to cancel the I/O request associated with this **aiocb**. If the *aiocbp* parameter is null, then an attempt is made to cancel all outstanding asynchronous I/O requests associated with the *FileDescriptor* parameter.

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

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

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

Parameters

FileDescriptor	Identifies the object to which the outstanding asynchronous I/O requests were originally queued.
aiocbp	Points to the **aiocb** structure associated with the I/O operation. The **aiocb** structure is defined in the **/usr/include/sys/aio.h** file and contains the following members:

```
int                aio_whence
off_t              aio_offset
char              *aio_buf
size_t             aio_nbytes
int                aio_reqprio
struct event       aio_event

struct osigevent   aio_event
int                aio_flag
aiohandle_t        aio_handle
```

aiocbp64	Points to the **aiocb64** structure associated with the I/O operation. The **aiocb** structure is defined in the **/usr/include/sys/aio.h** file and the same field as the aiocb structure with the execption that the **aio_offset** field is a 64 bit (**off64_t**) quantity.

Execution Environment

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

Return Values

AIO_CANCELED	Indicates that all of the asynchronous I/O requests were canceled successfully. The **aio_error** subroutine call with the handle to the **aiocb** structure of the request will return **ECANCELED**.
AIO_NOTCANCELED	Indicates that the **aio_cancel** subroutine did not cancel one or more outstanding I/O requests. This may happen if an I/O request is already in progress. The corresponding error status of the I/O request is not modified.
AIO_ALLDONE	Indicates that none of the I/O requests is in the queue or in progress.
–1	Indicates that the subroutine was not successful. Sets the **errno** global variable to identify the error.

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

EBADF	Indicates that the *FileDescriptor* parameter is not valid.

Implementation Specifics

The **aio_cancel** or **aio_cancel64** subroutine is part of Base Operating System (BOS) Runtime.

Related Information

The **aio_error** or **aio_error64** subroutine, **aio_read** or **aio_read64** subroutine, **aio_return** or **aio_return64** subroutine, **aio_suspend** or **aio_suspend64** subroutine, **aio_write** or **aio_write64** subroutine, **lio_listio** or **lio_listo64** subroutine.

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

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

aio_error or aio_error64 Subroutine

Purpose

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

Library

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

Syntax

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

```
int
aio_error(handle)
aio_handle_t handle;
```

```
int aio_error64(handle)
aio_handle_t handle;
```

Description

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

The **aio_error64** subroutine is similar to the **aio_error** subroutine except that it retrieves the error status associated with an **aiocb64** control block.

Parameters

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

Execution Environment

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

Return Values

0	Indicates that the operation completed successfully.
ECANCELED	Indicates that the I/O request was canceled due to an **aio_cancel** subroutine call.
EINPROG	Indicates that the I/O request has not completed.
	An **errno** value described in the **aio_read**, **aio_write**, and **lio_listio** subroutines: Indicates that the operation was not queued successfully. For example, if the **aio_read** subroutine is called with an unusable file descriptor, it (**aio_read**) returns a value of –1 and sets the **errno** global variable to **EBADF**. A subsequent call of the **aio_error** subroutine with the handle of the unsuccessful **aio control block** (**aiocb**) structure returns **EBADF**.
	An **errno** value of the corresponding I/O operation: Indicates that the operation was initiated successfully, but the actual I/O operation was unsuccessful. For example, calling the **aio_write** subroutine on a file located in a full file system returns a value of 0, which indicates the request was queued successfully. However, when the I/O operation is complete (that is, when the **aio_error** subroutine no longer returns **EINPROG**), the **aio_error** subroutine returns **ENOSPC**. This indicates that the I/O was unsuccessful.

Implementation Specifics

The **aio_error** and **aio_error64** subroutines are part of Base Operating System (BOS) Runtime.

Related Information

The **aio_cancel** or **aio_cancel64** subroutine, **aio_read** or **aio_read64** subroutine, **aio_return** or **aio_return64** subroutine, **aio_suspend** or **aio_suspend64** subroutine, **aio_write** or **aio_write64** subroutine, **lio_listio** or **lio_listio64** subroutine.

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

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

aio_read or aio_read64 Subroutine

Purpose

Reads asynchronously from a file.

Library

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

Syntax

```
#include <aio.h>

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

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

Description

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

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

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

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

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

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

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

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

Note: The **SIGIO** signal is replaced by real–time signals when they are available. The **event** structure in the **aiocb** structure is currently not in use but is included for future compatibility.

Parameters

FileDescriptor	Identifies the object to be read as returned from a call to open.
aiocbp	Points to the asynchronous I/O control block structure associated with the I/O operation. The **aiocb** and the **aiocb64** structures are defined in the **aio.h** file and contains the following members:

```
int            aio_whence
off_t          aio_offset
char           *aio_buf
size_t         aio_nbytes
int            aio_flag
aio_handle_t   aio_handle
```

Execution Environment

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

Return Values

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

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

EAGAIN	Indicates that the system resources required to queue the request are not available. Specifically, the transmit queue may be full, or the maximum number of opens may be reached.
EBADF	Indicates that the *FileDescriptor* parameter is not valid.
EFAULT	Indicates that the address specified by the *aiocbp* parameter is not valid.
EINVAL	Indicates that the `aio_whence` field does not have a valid value, or that the resulting pointer is not valid.

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

Implementation Specifics

The **aio_read** and **aio_read64** subroutines are part of Base Operating System (BOS) Runtime.

Related Information

The **aio_cancel** or **aio_cancel64** subroutine, **aio_error** or **aio_error64** subroutine, **aio_return** or **aio_return64** subroutine, **aio_suspend** or **aio_suspend64** subroutine, **aio_write** subroutine, **lio_listio** or **lio_listo64** subroutine.

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

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

aio_return or aio_return64 Subroutine

Purpose

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

Library

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

Syntax

```
#include <aio.h>


int aio_return(handle)
aio_handle_t handle;


int aio_return64(handle)
aio_handle_t handle;
```

Description

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

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

Parameters

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

Execution Environment

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

Return Values

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

Examples

An **aio_read** request to read 1000 bytes from a disk device eventually, when the **aio_error** subroutine returns a 0, causes the **aio_return** subroutine to return 1000. An **aio_read** request to read 1000 bytes from a 500 byte file eventually causes the **aio_return** subroutine to return 500. An **aio_write** request to write to a read–only file system results in the

aio_error subroutine eventually returning **EROFS** and the **aio_return** subroutine returning a value of –1.

Implementation Specifics

The **aio_return** and **aio_return64** subroutines are part of Base Operating System (BOS) Runtime.

Related Information

The **aio_cancel** or **aio_cancel64** subroutine, **aio_error** or **aio_error64** subroutine, **aio_read** or **aio_read64** subroutine, **aio_suspend** or **aio_suspend64** subroutine, **aio_write** or **aio_write64** subroutine, **lio_listio** or **lio_listio64** subroutine.

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

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

aio_suspend or aio_suspend64 Subroutine

Purpose

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

Library

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

Syntax

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

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

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

Description

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

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

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

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

Parameters

count	Specifies the number of entries in the *aiocbpa* array.
aiocbpa	Points to the **aiocb** or **aiocb64** structures associated with the asynchronous I/O operations. The **aiocb** structure is defined in the **aio.h** file and contains the following members:

```
int                aio_whence
off_t              aio_offset
char               *aio_buf
size_t             aio_nbytes
int                aio_reqprio
struct event       aio_event

struct osigevent   aio_event
int                aio_flag
aio_handle_t       aio_handle
```

Execution Environment

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

Return Values

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

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

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

Implementation Specifics

The **aio_suspend** or **aio_suspend64** subroutines are part of Base Operating System (BOS) Runtime.

Related Information

The **aio_cancel** or **aio_cancel64** subroutine, **aio_error** or **aio_error64** subroutine, **aio_read** or **aio_read64** subroutine, **aio_return** or **aio_return64** subroutine, **aio_write** or **aio_write64** subroutine, **lio_listio** or **lio_listo64** subroutine.

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

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

aio_write or aio_write64 Subroutine

Purpose

Writes to a file asynchronously.

Library

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

Syntax

```
#include <aio.h>

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

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

Description

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

`aio_buf`	Indicates the buffer to use.
`aio_nbytes`	Indicates the number of bytes to write.

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

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

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

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

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

Note: The **SIGIO** signal will be replaced by real–time signals when they are available. The **event** structure in the **aiocb** structure is currently not in use but is included for future compatibility.

Parameters

FileDescriptor Identifies the object to be written as returned from a call to open.

aiocbp Points to the asynchronous I/O control block structure associated with the I/O operation.

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

```
int                aio_whence
off_t              aio_offset
char               *aio_buf
size_t             aio_nbytes
int                aio_reqprio
struct event       aio_event

struct osigevent   aio_event
int                aio_flag
aio_handle_t       aio_handle
```

Execution Environment

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

Return Values

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

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

EAGAIN Indicates that the system resources required to queue the request are not available. Specifically, the transmit queue may be full, or the maximum number of opens may have been reached.

EBADF Indicates that the *FileDescriptor* parameter is not valid.

EFAULT Indicates that the address specified by the *aiocbp* parameter is not valid.

EINVAL Indicates that the `aio_whence` field does not have a valid value or that the resulting pointer is not valid.

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

Implementation Specifics

The **aio_write** or **aio_write64** subroutines are part of Base Operating System (BOS) Runtime.

Related Information

The **aio_cancel** or **aio_cancel64** subroutine, **aio_error** or **aio_error64** subroutine, **aio_read** or **aio_read64** subroutine, **aio_return** or **aio_return64** subroutine, **aio_suspend** or **aio_suspend64** subroutine, **lio_listio** or **lio_listio64** subroutine.

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

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

asin, asinl, acos, acosl, atan, atanl, atan2, or atan2l Subroutine

Purpose

Computes inverse trigonometric functions.

Libraries

IEEE Math Library (**libm.a**)
or System V Math Library (**libmsaa.a**)

Syntax

```
double asin (x)
double x;

long double asinl (x)
long double x;

double acos (x)
double x;

long double acosl (x)
long double x;

double atan (x)
double x;

long double atanl (x)
long double x;

double atan2 (y, x)
double y, x;

long double atan2l (x,y)
long double y, x;
```

Description

The **asin** and **asinl** subroutines return the principal value of the arc sine of *x,* in the range [–pi/2, pi/2].

The **acos** and **acosl** subroutines return the principal value of the arc cosine of *x,* in the range [0, pi].

The **atan** and **atanl** subroutines return the principal value of the arc tangent of *x,* in the range [–pi/2, pi/2].

The **atan2** and **atan2l** subroutines return the principal value of the arc tangent of *y/x,* using the signs of both parameters to determine the quadrant of the return value. The return values are in the range [–pi, pi].

Parameters

x	Specifies a double–precision floating–point value. For the **asinl**, **acosl**, **atanl**, and **atan2l** subroutines, specifies a long double–precision floating–point value.
y	Specifies a double–precision floating–point value. For the **asinl**, **acosl**, **atanl**, and **atan2l** subroutines, specifies long double–precision floating–point value.

Error Codes

When using the **libm.a** (**–lm**) library:

asin, asinl, acos, acosl	Return a NaNQ and set the **errno** global variable to **EDOM** if the absolute value of the parameter is greater than 1.

When using **libmsaa.a** (**–lmsaa**):

asin, acos, atan2,	If the absolute value of the parameter of **asin** or **acos** is greater than 1, or if both parameters of **atan2** are 0, then 0 is returned and **errno** is set to **EDOM**. In addition, a message indicating **DOMAIN** error is printed on the standard output.
asinl, acosl, atan2l	Return a NaNQ and set the **errno** global variable to **EDOM** if the absolute value of the parameter is greater than 1.

These error–handling procedures may be changed with the **matherr** subroutine when using the **libmsaa.a** (**–lmsaa**) library.

Implementation Specifics

These subroutines are part of Base Operating System (BOS) Runtime.

Related Information

The **asinh**, **acosh**, or **atanh** subroutine, **matherr** subroutine.

Subroutines Overview in *AIX General Programming Concepts : Writing and Debugging Programs*.

128–Bit long double Floating–Point Format in *AIX General Programming Concepts : Writing and Debugging Programs*.

asinh, acosh, or atanh Subroutine

Purpose

Computes inverse hyperbolic functions.

Libraries

IEEE Math Library (**libm.a**) or System V Math Library (**libmsaa.a**)

Syntax

```
#include <math.h>

double asinh (x)
double x;

double acosh (x)
double x;

double atanh (x)
double x;
```

Description

The **asinh**, **acosh**, and **atanh** subroutines compute the inverse hyperbolic functions.

The **asinh** subroutine returns the hyperbolic arc sine specified by the *x* parameter, in the range of the **–HUGE_VAL** value to the **+HUGE_VAL** value. The **acosh** subroutine returns the hyperbolic arc cosine specified by the *x* parameter, in the range 1 to the **+HUGE_VAL** value. The **atanh** subroutine returns the hyperbolic arc tangent specified by the *x* parameter, in the range of the **–HUGE_VAL** value to the **+HUGE_VAL** value.

Note: Compile any routine that uses subroutines from the **libm.a** library with the **–lm** flag. For example: to compile the **asinh.c** file, enter:

```
cc asinh.c –lm
```

Parameters

x	Specifies a double–precision floating–point value.

Error Codes

The **acosh** subroutine returns **NaNQ** (not–a–number) and sets **errno** to **EDOM** if the *x* parameter is less than the value of 1.

The **atanh** subroutine returns **NaNQ** and sets **errno** to **EDOM** if the absolute value of x is greater than 1.

Implementation Specifics

These subroutines are part of Base Operating System (BOS) Runtime.

Related Information

The **copysign**, **nextafter**, **scalb**, **logb**, or **ilogb** subroutine, **exp**, **expm1**, **log**, **log10**, or **pow** subroutine, **sinh**, **cosh**, or **tanh** subroutine.

Subroutines Overview in *AIX General Programming Concepts : Writing and Debugging Programs*.

assert Macro

Purpose

Verifies a program assertion.

Library

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

Syntax

```
#include <assert.h>

void assert (Expression)
int  Expression;
```

Description

The **assert** macro puts error messages into a program. If the specified expression is false, the **assert** macro writes the following message to standard error and stops the program:

```
Assertion failed: Expression, file FileName, line LineNumber
```

In the error message, the *FileName* value is the name of the source file and the *LineNumber* value is the source line number of the **assert** statement.

Parameters

Expression	Specifies an expression that can be evaluated as true or false. This expression is evaluated in the same manner as the C language IF statement.

Implementation Specifics

This macro is part of Base Operating System (BOS) Runtime.

The **assert** macro uses the **_assert** subroutine.

Related Information

The **abort** subroutine.

The **cpp** command.

Subroutines Overview in *AIX General Programming Concepts : Writing and Debugging Programs*.

atof, strtod, strtold, atoff, or strtof Subroutine

Purpose

Converts an ASCII string to a floating–point or double floating–point number.

Libraries

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

Syntax

```
#include <stdlib.h>

double atof (NumberPointer)
const char *NumberPointer;

double strtod (NumberPointer, EndPointer)
const char *NumberPointer
char**EndPointer;

long double strtold (NumberPointer, EndPointer)
char *NumberPointer, **EndPointer;

float atoff (NumberPointer)
char *NumberPointer;

float strtof (NumberPointer, EndPointer)
char *NumberPointer, **EndPointer;
```

Description

The **atof** subroutine and **strtod** subroutine convert a character string, pointed to by the *NumberPointer* parameter, to a double–precision floating–point number. Similarly, the **strtold** subroutine converts a character string to a long double–precision floating–point number. The **atoff** subroutine and **strtof** subroutine convert a character string, pointed to by the *NumberPointer* parameter, to a single–precision floating–point number. The first unrecognized character ends the conversion.

Except for behavior on error, the **atof** subroutine is equivalent to the **strtod** subroutine call, with the *EndPointer* parameter set to (**char****) NULL.

Except for behavior on error, the **atoff** subroutine is equivalent to the **strtof** subroutine call, with the *EndPointer* parameter set to (**char****) NULL.

These subroutines recognize a character string when the characters are in one of two formats: numbers or numeric symbols.

- For a string to be recognized as a number, it should contain the following pieces in the following order:
  a. An optional string of white–space characters
  b. An optional sign
  c. A nonempty string of digits optionally containing a radix character
  d. An optional exponent in E–format or e–format followed by an optionally signed integer.
- For a string to be recognized as a numeric symbol, it should contain the following pieces in the following order:
  a. An optional string of white–space characters
  b. An optional sign
  c. One of the strings: **INF**, **infinity**, **NaNQ**, **NaNS**, or **NaN** (case insensitive)

Parameters

NumberPointer	Specifies a character string to convert.
EndPointer	Specifies a pointer to the character that ended the scan or a null value.

Return Values

Upon successful completion, the **atof**, **atoff**, **strtod**, **strtold**, and **strtof** subroutines return the converted value. If no conversion could be performed, a value of 0 is returned and the **errno** global variable is set to indicate the error.

Error Codes

Note: Because a value of 0 can indicate either an error or a valid result, an application that checks for errors with the **strtod**, **strtof**, and **strtold** subroutines should set the **errno** global variable equal to 0 prior to the subroutine call. The application can check the **errno** global variable after the subroutine call.

If the string pointed to by *NumberPointer* is empty or begins with an unrecognized character, a value of 0 is returned for the **strtod**, **strtof**, and **strtold** subroutines.

If the conversion cannot be performed, a value of 0 is returned, and the **errno** global variable is set to indicate the error.

If the conversion causes an overflow (that is, the value is outside the range of representable values), **+/– HUGE_VAL** is returned with the sign indicating the direction of the overflow, and the **errno** global variable is set to **ERANGE**.

If the conversion would cause an underflow, a properly signed value of 0 is returned and the **errno** global variable is set to **ERANGE**.

For the **strtod**, **strtof**, and **strtold** subroutines, if the value of the *EndPointer* parameter is not (**char****) NULL, a pointer to the character that stopped the subroutine is stored in **EndPointer*. If a floating–point value cannot be formed, **EndPointer* is set to *NumberPointer*.

The **atoff** and **strtof** subroutines have only one rounding error. (If the **atof** or **strtod** subroutines are used to create a double–precision floating–point number and then that double–precision number is converted to a floating–point number, two rounding errors could occur.)

Implementation Specifics

These subroutines are part of Base Operating System (BOS) Runtime.

The **atoff** and **strtof** subroutines are not part of the ANSI C Library. These subroutines are at least as accurate as required by the *IEEE Standard for Binary Floating–Point Arithmetic*. The **atof** and **strtod** subroutines accept at least 17 significant decimal digits. The **atoff** and **strtof** subroutines accept at least 9 leading 0's. Leading 0's are not counted as significant digits.

Related Information

The **scanf** subroutine, **strtol**, **strtoul**, **atol**, or **atoi** subroutine, **wstrtol**, **watol**, or **watoi** subroutine.

Subroutines Overview in *AIX General Programming Concepts : Writing and Debugging Programs*.

128–Bit long double Floating–Point Format in *AIX General Programming Concepts : Writing and Debugging Programs*.

audit Subroutine

Purpose

Enables and disables system auditing.

Library

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

Syntax

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

int audit (Command, Argument)
int Command;
int Argument;
```

Description

The **audit** subroutine enables or disables system auditing.

When auditing is enabled, audit records are created for security–relevant events. These records can be collected through the **auditbin** subroutine, or through the /**dev**/**audit** special file interface.

Parameters

Command — Defined in the **sys**/**audit.h** file, can be one of the following values:

AUDIT_QUERY Returns a mask indicating the state of the auditing subsystem. The mask is a logical ORing of the **AUDIT_ON**, **AUDIT_OFF**, and **AUDIT_PANIC** flags. The *Argument* parameter is ignored.

AUDIT_ON Enables auditing. If auditing is already enabled, only the failure–mode behavior changes. The *Argument* parameter specifies recovery behavior in the event of failure and may be either 0 or the value **AUDIT_PANIC**.

Note: If **AUDIT_PANIC** is specified, bin–mode auditing must be enabled before the **audit** subroutine call.

AUDIT_OFF Disables the auditing system if auditing is enabled. If the auditing system is disabled, the **audit** subroutine does nothing. The *Argument* parameter is ignored.

AUDIT_RESET Disables the auditing system (as does **AUDIT_OFF**) and resets the auditing system. If auditing is already disabled, only the system configuration is reset. Resetting the audit configuration involves clearing the audit events and audited objects table, and terminating bin and stream auditing. The *Argument* parameter is ignored.

AUDIT_EVENT_THRESHOLD
Audit event records will be buffered until a total of *Argument* records have been saved, at which time the audit event records will be flushed to disk. An *Argument* value of zero disables this functionality. This parameter only applies to AIX Version 4.1.4 and later.

AUDIT_BYTE_THRESHOLD
Audit event data will be buffered until a total of *Argument* bytes of data have been saved, at which time the audit event data will be flushed to disk. An *Argument* value of zero disables this functionality. This parameter only applies to AIX Version 4.1.4 and later.

Argument Specifies the behavior when a bin write fails (for **AUDIT_ON**) or specifies the size of the audit event buffer (for **AUDIT_EVENT_THRESHOLD** and **AUDIT_BYTE_THRESHOLD**). For all other commands, the value of **Argument** is ignored. The valid values are:

AUDIT_PANIC The operating system shuts down if an audit record cannot be written to a bin.

Note: If **AUDIT_PANIC** is specified, bin–mode auditing must be enabled before the **audit** subroutine call.

BufferSize The number of bytes or audit event records which will be buffered. This parameter is valid only with the command **AUDIT_BYTE_THRESHOLD** and **AUDIT_EVENT_THRESHOLD**. A value of zero will disable either byte (for **AUDIT_BYTE_THRESHOLD**) or event (for **AUDIT_EVENT_THRESHOLD**) buffering.

Return Values

For a *Command* value of **AUDIT_QUERY**, the **audit** subroutine returns, upon successful completion, a mask indicating the state of the auditing subsystem. The mask is a logical ORing of the **AUDIT_ON**, **AUDIT_OFF**, **AUDIT_PANIC**, and **AUDIT_NO_PANIC** flags. For any other *Command* value, the **audit** subroutine returns 0 on successful completion.

If the **audit** subroutine fails, a value of –1 is returned and the **errno** global variable is set to indicate the error.

Error Codes

The **audit** subroutine fails if either of the following is true:

EINVAL The *Command* parameter is not one of **AUDIT_ON**, **AUDIT_OFF**, **AUDIT_RESET**, or **AUDIT_QUERY**.

EINVAL The *Command* parameter is **AUDIT_ON** and the *Argument* parameter specifies values other than **AUDIT_PANIC**.

EPERM The calling process does not have root user authority.

Implementation Specifics

This subroutine is part of Base Operating System (BOS) Runtime.

Files

dev/audit Specifies the audit pseudo–device from which the audit records are read.

Related Information

The **auditbin** subroutine, **auditevents** subroutine, **auditlog** subroutine, **auditobj** subroutine, **auditproc** subroutine.

The **audit** command.

List of Security and Auditing Subroutines and Subroutines Overview in *AIX General Programming Concepts : Writing and Debugging Programs*.

auditbin Subroutine

Purpose

Defines files to contain audit records.

Library

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

Syntax

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

int auditbin (Command, Current, Next, Threshold)
int Command;
int Current;
int Next;
int Threshold;
```

Description

The **auditbin** subroutine establishes an audit bin file into which the kernel writes audit records. Optionally, this subroutine can be used to establish an overflow bin into which records are written when the current bin reaches the size specified by the *Threshold* parameter.

Parameters

Command — If nonzero, this parameter is a logical ORing of the following values, which are defined in the **sys/audit.h** file:

- **AUDIT_EXCL** — Requests exclusive rights to the audit bin files. If the file specified by the *Current* parameter is not the kernel's current bin file, the **auditbin** subroutine fails immediately with the **errno** variable set to **EBUSY**.
- **AUDIT_WAIT** — The **auditbin** subroutine should not return until:
  - **bin full** — The kernel writes the number of bytes specified by the *Threshold* parameter to the file descriptor specified by the *Current* parameter. Upon successful completion, the **auditbin** subroutine returns a 0. The kernel writes subsequent audit records to the file descriptor specified by the *Next* parameter.
  - **bin failure** — An attempt to write an audit record to the file specified by the *Current* parameter fails. If this occurs, the **auditbin** subroutine fails with the **errno** variable set to the return code from the **auditwrite** subroutine.
  - **bin contention** — Another process has already issued a successful call to the **auditbin** subroutine. If this occurs, the **auditbin** subroutine fails with the **errno** variable set to **EBUSY**.
  - **system shutdown** — The auditing system was shut down. If this occurs, the **auditbin** subroutine fails with the **errno** variable set to **EINTR**.

Current — A file descriptor for a file to which the kernel should immediately write audit records.

Next	Specifies the file descriptor that will be used as the current audit bin if the value of the *Threshold* parameter is exceeded or if a write to the current bin fails. If this value is –1, no switch occurs.
Threshold	Specifies the maximum size of the current bin. If 0, the auditing subsystem will not switch bins. If it is nonzero, the kernel begins writing records to the file specified by the *Next* parameter, if writing a record to the file specified by the *Cur* parameter would cause the size of this file to exceed the number of bytes specified by the *Threshold* parameter. If no next bin is defined and **AUDIT_PANIC** was specified when the auditing subsystem was enabled, the system is shut down. If the size of the *Threshold* parameter is too small to contain a bin header and a bin tail, the **auditbin** subroutine fails and the **errno** variable is set to **EINVAL**.

Return Values

If the **auditbin** subroutine is successful, a value of 0 returns.

If the **auditbin** subroutine fails, a value of –1 returns and the **errno** global variable is set to indicate the error. If this occurs, the result of the call does not indicate whether any records were written to the bin.

Error Codes

The **auditbin** subroutine fails if any of the following is true:

EBADF	The *Current* parameter is not a file descriptor for a regular file open for writing, or the *Next* parameter is neither –1 nor a file descriptor for a regular file open for writing.
EBUSY	The *Command* parameter specifies **AUDIT_EXCL** and the kernel is not writing audit records to the file specified by the *Current* parameter.
EBUSY	The *Command* parameter specifies **AUDIT_WAIT** and another process has already registered a bin.
EINTR	The auditing subsystem is shut down.
EINVAL	The *Command* parameter specifies a nonzero value other than **AUDIT_EXCL** or **AUDIT_WAIT**.
EINVAL	The *Threshold* parameter value is less than the size of a bin header and trailer.
EPERM	The caller does not have root user authority.

Implementation Specifics

This subroutine is part of Base Operating System (BOS) Runtime.

Related Information

The **audit** subroutine, **auditevents** subroutine, **auditlog** subroutine, **auditobj** subroutine, **auditproc** subroutine.

The **audit** command.

The **audit** file format.

List of Security and Auditing Subroutines and Subroutines Overview in *AIX General Programming Concepts : Writing and Debugging Programs*.

auditevents Subroutine

Purpose

Gets or sets the status of system event auditing.

Library

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

Syntax

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

int auditevents (Command, Classes, NClasses)
int Command;
struct audit_class *Classes;
int NClasses;
```

Description

The **auditevents** subroutine queries or sets the audit class definitions that control event auditing. Each audit class is a set of one or more audit events.

System auditing need not be enabled before calling the **auditevents** subroutine. The **audit** subroutine can be directed with the **AUDIT_RESET** command to clear all event lists.

Parameters

Command
: Specifies whether the event lists are to be queried or set. The values, defined in the **sys/audit.h** file, for the *Command* parameter are:

  AUDIT_SET
  : Sets the lists of audited events after first clearing all previous definitions.

  AUDIT_GET
  : Queries the lists of audited events.

  AUDIT_LOCK
  : Queries the lists of audited events. This value also blocks any other process attempting to set or lock the list of audit events. The lock is released when the process holding the lock dies or calls the **auditevents** subroutine with the *Command* parameter set to **AUDIT_SET**.

Classes
: Specifies the array of **a_event** structures for the **AUDIT_SET** operation, or after an **AUDIT_GET** or **AUDIT_LOCK** operation. The **audit_class** structure is defined in the **sys/audit.h** file and contains the following members:

  `ae_name`
  : A pointer to the name of the audit class.

  `ae_list`
  : A pointer to a list of null–terminated audit event names for this audit class. The list is ended by a null name (a leading null byte or two consecutive null bytes).

  Note: Event and class names are limited to 15 significant characters.

  `ae_len`
  : The length of the event list in the **ae_list** member. This length includes the terminating null bytes. On an **AUDIT_SET** operation, the caller must set this member to indicate the actual length of the list (in bytes) pointed to by `ae_list`. On an **AUDIT_GET** or **AUDIT_LOCK** operation, the **auditevents** subroutine sets this member to indicate the actual size of the list.

NClasses
: Serves a dual purpose. For **AUDIT_SET**, the *NClasses* parameter specifies the number of elements in the events array. For **AUDIT_GET** and **AUDIT_LOCK**, the *NClasses* parameter specifies the size of the buffer pointed to by the *Classes* parameter.

Attention: Only 32 audit classes are supported. One class is implicitly defined by the system to include all audit events (ALL). The administrator of your system should not attempt to define more than 31 audit classes.

Security

The calling process must have root user authority in order to use the **auditevents** subroutine.

Return Codes

If the **auditevents** subroutine completes successfully, the number of audit classes is returned if the *Command* parameter is **AUDIT_GET** or **AUDIT_LOCK**. A value of 0 is returned if the *Command* parameter is **AUDIT_SET**. If this call fails, a value of –1 is returned and the **errno** global variable is set to indicate the error.

Error Codes

The **auditevents** subroutine fails if one or more of the following are true:

EPERM	The calling process does not have root user authority.
EINVAL	The value of *Command* is not **AUDIT_SET**, **AUDIT_GET**, or **AUDIT_LOCK**.
EINVAL	The *Command* parameter is **AUDIT_SET**, and the value of the *NClasses* parameter is greater than or equal to 32.
EINVAL	A class name or event name is longer than 15 significant characters.
ENOSPC	The value of *Command* is **AUDIT_GET** or **AUDIT_LOCK** and the size of the buffer specified by the *NClasses* parameter is not large enough to hold the list of event structures and names. If this occurs, the first word of the buffer is set to the required buffer size.
EFAULT	The *Classes* parameter points outside of the process' address space.
EFAULT	The `ae_list` member of one or more **audit_class** structures passed for an **AUDIT_SET** operation points outside of the process' address space.
EFAULT	The *Command* value is **AUDIT_GET** or **AUDIT_LOCK** and the size of the *Classes* buffer is not large enough to hold an integer.
EBUSY	Another process has already called the **auditevents** subroutine with **AUDIT_LOCK**.
ENOMEM	Memory allocation failed.

Implementation Specifications

This subroutine is part of Base Operating System (BOS) Runtime.

Related Information

The **audit** subroutine, **auditbin** subroutine, **auditlog** subroutine, **auditobj** subroutine, **auditproc** subroutine, **auditread** subroutine, **auditwrite** subroutine.

The **audit** command.

List of Security and Auditing Subroutines and Subroutines Overview in *AIX General Programming Concepts : Writing and Debugging Programs*.

auditlog Subroutine

Purpose

Appends an audit record to the audit trail file.

Library

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

Syntax

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

int auditlog (Event, Result, Buffer, BufferSize)
char *Event;
int Result;
char *Buffer;
int BufferSize;
```

Description

The **auditlog** subroutine generates an audit record. The kernel audit–logging component appends a record for the specified *Event* if system auditing is enabled, process auditing is not suspended, and the *Event* parameter is in one or more of the audit classes for the current process.

The audit logger generates the audit record by adding the *Event* and *Result* parameters to the audit header and including the resulting information in the *Buffer* parameter as the audit tail.

Parameters

Event — The name of the audit event to be generated. This parameter should be the name of an audit event. Audit event names are truncated to 15 characters plus null.

Result — Describes the result of this event. Valid values are defined in the **sys/audit.h** file and include the following:

- **AUDIT_OK** — The event was successful.
- **AUDIT_FAIL** — The event failed.
- **AUDIT_FAIL_ACCESS** — The event failed because of any access control denial.
- **AUDIT_FAIL_DAC** — The event failed because of a discretionary access control denial.
- **AUDIT_FAIL_PRIV** — The event failed because of a privilege control denial.
- **AUDIT_FAIL_AUTH** — The event failed because of an authentication denial.

Other nonzero values of the *Result* parameter are converted into the **AUDIT_FAIL** value.

Buffer — Points to a buffer containing the tail of the audit record. The format of the information in this buffer depends on the event name.

BufferSize — Specifies the size of the *Buffer* parameter, including the terminating null.

Return Values

Upon successful completion, the **auditlog** subroutine returns a value of 0. If **auditlog** fails, a value of –1 is returned and the **errno** global variable is set to indicate the error.

The **auditlog** subroutine does not return any indication of failure to write the record where this is due to inappropriate tailoring of auditing subsystem configuration files or user–written code. Accidental omissions and typographical errors in the configuration are potential causes of such a failure.

Error Codes

The **auditlog** subroutine fails if any of the following are true:

EFAULT	The *Event* or *Buffer* parameter points outside of the process' address space.
EINVAL	The auditing system is either interrupted or not initialized.
EINVAL	The length of the audit record is greater than 32 kilobytes.
EPERM	The process does not have root user authority.
ENOMEM	Memory allocation failed.

Implementation Specifics

This subroutine is part of Base Operating System (BOS) Runtime.

Related Information

The **audit** subroutine, **auditbin** subroutine, **auditevents** subroutine, **auditobj** subroutine, **auditproc** subroutine, **auditwrite** subroutine.

List of Security and Auditing Subroutines and Subroutines Overview in *AIX General Programming Concepts : Writing and Debugging Programs*.

auditobj Subroutine

Purpose

Gets or sets the auditing mode of a system data object.

Library

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

Syntax

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

int auditobj (Command, Obj_Events, ObjSize)
int Command;
struct o_event *Obj_Events;
int ObjSize;
```

Description

The **auditobj** subroutine queries or sets the audit events to be generated by accessing selected objects. For each object in the file system name space, it is possible to specify the event generated for each access mode. Using the **auditobj** subroutine, an administrator can define new audit events in the system that correspond to accesses to specified objects. These events are treated the same as system–defined events.

System auditing need not be enabled to set or query the object audit events. The **audit** subroutine can be directed with the **AUDIT_RESET** command to clear the definitions of object audit events.

Parameters

Command Specifies whether the object audit event lists are to be read or written. The valid values, defined in the **sys**/**audit.h** file, for the *Command* parameter are:

AUDIT_SET Sets the list of object audit events, after first clearing all previous definitions.

AUDIT_GET Queries the list of object audit events.

AUDIT_LOCK Queries the list of object audit events and also blocks any other process attempting to set or lock the list of audit events. The lock is released when the process holding the lock dies or calls the **auditobj** subroutine with the *Command* parameter set to **AUDIT_SET**.

Obj_Events Specifies the array of **o_event** structures for the **AUDIT_SET** operation or for after the **AUDIT_GET** or **AUDIT_LOCK** operation. The **o_event** structure is defined in the **sys/audit.h** file and contains the following members:

`o_type` Specifies the type of the object, in terms of naming space. Currently, only one object–naming space is supported:

AUDIT_FILE Denotes the file system naming space.

`o_name` Specifies the name of the object.

`o_event` Specifies any array of event names to be generated when the object is accessed. Note that event names in AIX are currently limited to 16 bytes, including the trailing null. The index of an event name in this array corresponds to an access mode. Valid indexes are defined in the **audit.h** file and include the following:

- **AUDIT_READ**
- **AUDIT_WRITE**
- **AUDIT_EXEC**

ObjSize For an **AUDIT_SET** operation, the *ObjSize* parameter specifies the number of object audit event definitions in the array pointed to by the *Obj_Events* parameter. For an **AUDIT_GET** or **AUDIT_LOCK** operation, the *ObjSize* parameter specifies the size of the buffer pointed to by the *Obj_Events* parameter.

Return Values

If the **auditobj** subroutine completes successfully, the number of object audit event definitions is returned if the *Command* parameter is **AUDIT_GET** or **AUDIT_LOCK**. A value of 0 is returned if the *Command* parameter is **AUDIT_SET**. If this call fails, a value of –1 is returned and the **errno** global variable is set to indicate the error.

Error Codes

The **auditobj** subroutine fails if any of the following are true:

EFAULT The *Obj_Events* parameter points outside the address space of the process.

EFAULT The *Command* parameter is **AUDIT_SET**, and one or more of the `o_name` members points outside the address space of the process.

EFAULT The *Command* parameter is **AUDIT_GET** or **AUDIT_LOCK**, and the buffer size of the *Obj_Events* parameter is not large enough to hold the integer.

EINVAL The value of the *Command* parameter is not **AUDIT_SET**, **AUDIT_GET** or **AUDIT_LOCK**.

EINVAL The *Command* parameter is **AUDIT_SET**, and the value of one or more of the `o_type` members is not **AUDIT_FILE**.

EINVAL An event name was longer than 15 significant characters.

ENOENT The *Command* parameter is **AUDIT_SET**, and the parent directory of one of the file–system objects does not exist.

ENOSPC The value of the *Command* parameter is **AUDIT_GET** or **AUDIT_LOCK**, and the size of the buffer as specified by the *ObjSize* parameter is not large enough to hold the list of event structures and names. If this occurs, the first word of the buffer is set to the required buffer size.

ENOMEM	Memory allocation failed.
EBUSY	Another process has called the **auditobj** subroutine with **AUDIT_LOCK**.
EPERM	The caller does not have root user authority.

Implementation Specifics

This subroutine is part of Base Operating System (BOS) Runtime.

Related Information

The **audit** subroutine, **auditbin** subroutine, **auditevents** subroutine, **auditlog** subroutine, **auditproc** subroutine.

The **audit** command.

The **audit.h** file.

List of Security and Auditing Subroutines and Subroutines Overview in *AIX General Programming Concepts : Writing and Debugging Programs*.

auditpack Subroutine

Purpose

Compresses and uncompresses audit bins.

Library

Security Library (**libc.a**)

Syntax

```
#include <sys/audit.h>
#include <stdio.h>

char *auditpack (Expand, Buffer)
int Expand;
char *Buffer;
```

Description

The **auditpack** subroutine can be used to compress or uncompress bins of audit records.

Parameters

Expand Specifies the operation. Valid values, as defined in the **sys/audit.h** header file, are one of the following:

AUDIT_PACK Performs standard compression on the audit bin.

AUDIT_UNPACK Unpacks the compressed audit bin.

Buffer Specifies the buffer containing the bin to be compressed or uncompressed. This buffer must contain a standard bin as described in the **audit.h** file.

Return Values

If the **auditpack** subroutine is successful, a pointer to a buffer containing the processed audit bin is returned. If unsuccessful, a null pointer is returned and the **errno** global variable is set to indicate the error.

Error Codes

The **auditpack** subroutine fails if one or more of the following values is true:

EINVAL The *Expand* parameter is not one of the valid values (**AUDIT_PACK** or **AUDIT_UNPACK**).

EINVAL The *Expand* parameter is **AUDIT_UNPACK** and the packed data in *Buffer* does not unpack to its original size.

EINVAL The *Expand* parameter is **AUDIT_PACK** and the bin in the *Buffer* parameter is already compressed, or the *Expand* parameter is **AUDIT_UNPACK** and the bin in the *Buffer* parameter is already unpacked.

ENOSPC The **auditpack** subroutine is unable to allocate space for a new buffer.

Implementation Specifics

This subroutine is part of Base Operating System (BOS) Runtime.

Related Information

The **auditread** subroutine.

The **auditcat** command.

List of Security and Auditing Subroutines and Subroutines Overview in *AIX General Programming Concepts : Writing and Debugging Programs*.

auditproc Subroutine

Purpose

Gets or sets the audit state of a process.

Library

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

Syntax

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

int auditproc (ProcessID, Command, Argument, Length)
int ProcessID;
int Command;
char * Argument;
int Length;
```

Description

The **auditproc** subroutine queries or sets the auditing state of a process. There are two parts to the auditing state of a process:

- The list of classes to be audited for this process. Classes are defined by the **auditevents** subroutine. Each class includes a set of audit events. When a process causes an audit event, that event may be logged in the audit trail if it is included in one or more of the audit classes of the process.
- The audit status of the process. Auditing for a process may be suspended or resumed. Functions that generate an audit record can first check to see whether auditing is suspended. If process auditing is suspended, no audit events are logged for a process. For more information, see the **auditlog** subroutine.

Parameters

ProcessID The process ID of the process to be affected. If *ProcessID* is 0, the **auditproc** subroutine affects the current process.

Command The action to be taken. Defined in the **audit.h** file, valid values include:

AUDIT_KLIST_EVENTS
Sets the list of audit classes to be audited for the process and also sets the user's default audit classes definition within the kernel. The *Argument* parameter is a pointer to a list of null–terminated audit class names. The *Length* parameter is the length of this list, including null bytes.

AUDIT_QEVENTS Returns the list of audit classes defined for the current process if *ProcessID* is 0. Otherwise, it returns the list of audit classes defined for the specified process ID. The *Argument* parameter is a pointer to a character buffer. The *Length* parameter specifies the size of this buffer. On return, this buffer contains a list of null–terminated audit class names. A null name terminates the list.

AUDIT_EVENTS Sets the list of audit classes to be audited for the process. The *Argument* parameter is a pointer to a list of null–terminated audit class names. The *Length* parameter is the length of this list, including null bytes.

AUDIT_QSTATUS Returns the audit status of the current process. You can only check the status of the current process. If the *ProcessID* parameter is nonzero, a –1 is returned and the **errno** global variable is set to **EINVAL**. The *Length* and *Argument* parameters are ignored. A return value of **AUDIT_SUSPEND** indicates that auditing is suspended. A return value of **AUDIT_RESUME** indicates normal auditing for this process.

AUDIT_STATUS Sets the audit status of the current process. The *Length* parameter is ignored, and the *ProcessID* parameter must be zero. If *Argument* is **AUDIT_SUSPEND**, the audit status is set to suspend event auditing for this process. If the *Argument* parameter is **AUDIT_RESUME**, the audit status is set to resume event auditing for this process.

Argument A character pointer for the audit class buffer for an **AUDIT_EVENT** or **AUDIT_QEVENTS** value of the *Command* parameter or an integer defining the audit status to be set for an **AUDIT_STATUS** operation.

Length Size of the audit class character buffer.

Return Values

The **auditproc** subroutine returns the following values upon successful completion:

- The previous audit status (**AUDIT_SUSPEND** or **AUDIT_RESUME**), if the call queried or set the audit status (the *Command* parameter specified **AUDIT_QSTATUS** or **AUDIT_STATUS**)
- A value of 0 if the call queried or set audit events (the *Command* parameter specified **AUDIT_QEVENTS** or **AUDIT_EVENTS**)

Error Codes

If the **auditproc** subroutine fails if one or more of the following are true:

EINVAL	An invalid value was specified for the *Command* parameter.
EINVAL	The *Command* parameter is set to the **AUDIT_QSTATUS** or **AUDIT_STATUS** value and the **pid** value is nonzero.
EINVAL	The *Command* parameter is set to the **AUDIT_STATUS** value and the *Argument* parameter is not set to **AUDIT_SUSPEND** or **AUDIT_RESUME**.
ENOSPC	The *Command* parameter is **AUDIT_QEVENTS**, and the buffer size is insufficient. In this case, the first word of the *Argument* parameter is set to the required size.
EFAULT	The *Command* parameter is **AUDIT_QEVENTS** or **AUDIT_EVENTS** and the *Argument* parameter points to a location outside of the process' allocated address space.
ENOMEM	Memory allocation failed.
EPERM	The caller does not have root user authority.

Implementation Specifics

This subroutine is part of Base Operating System (BOS) Runtime.

Related Information

The **audit** subroutine, **auditbin** subroutine, **auditevents** subroutine, **auditlog** subroutine, **auditobj** subroutine, **auditwrite** subroutine.

List of Security and Auditing Subroutines and Subroutines Overview in *AIX General Programming Concepts : Writing and Debugging Programs*.

auditread, auditread_r Subroutines

Purpose

Reads an audit record.

Library

Security Library (**libc.a**)

Syntax

```
#include <sys/audit.h>
#include <stdio.h>
char *auditread (FilePointer,AuditRecord)
FILE *FilePointer;
struct aud_rec *AuditRecord;


char *auditread_r (FilePointer,AuditRecord, RecordSize,
StreamInfo)
FILE *FilePointer;
struct aud_rec *AuditRecord;
size_t RecordSize;
void **StreamInfo;
```

Description

The **auditread** subroutine reads the next audit record from the specified file descriptor. Bins on this input stream are unpacked and uncompressed if necessary.

The **auditread** subroutine can not be used on more than one *FilePointer* as the results can be unpredictable. Use the **auditread_r** subroutine instead.

The **auditread_r** subroutine reads the next audit from the specified file descriptor. This subroutine is thread safe and can be used to handle multiple open audit files simultaneously by multiple threads of execution.

The **auditread_r** subroutine is able to read multiple versions of audit records. The version information contained in an audit record is used to determine the correct size and format of the record. When an input record header is larger than *AuditRecord*, an error is returned. In order to provide for binary compatibility with previous versions, if *RecordSize* is the same size as the original (**struct aud_rec**), the input record is converted to the original format and returned to the caller.

Parameters

FilePointer	Specifies the file descriptor from which to read.
AuditRecord	Specifies the buffer to contain the header. The first short in this buffer must contain a valid number for the header.
RecordSize	The size of the buffer referenced by *AuditRecord*.
StreamInfo	A pointer to an opaque datatype used to hold information related to the current value of *FilePointer*. For each new value of *FilePointer*, a new *StreamInfo* pointer must be used. *StreamInfo* must be initialized to NULL by the user and is initialized by **auditread_r** when first used. When *FilePointer* has been closed, the value of *StreamInfo* can be passed to the **free** subroutine to be deallocated.

Return Values

If the **auditread** subroutine completes successfully, a pointer to a buffer containing the tail of the audit record is returned. The length of this buffer is returned in the `ah_length` field of the header file. If this subroutine is unsuccessful, a null pointer is returned and the **errno** global variable is set to indicate the error.

Error Codes

The **auditread** subroutine fails if one or more of the following is true:

EBADF	The *FilePointer* value is not valid.
ENOSPC	The **auditread** subroutine is unable to allocate space for the tail buffer.

Other error codes are returned by the **read** subroutine.

Implementation Specifics

This subroutine is part of Base Operating System (BOS) Runtime.

Related Information

The **auditpack** subroutine.

List of Security and Auditing Subroutines and Subroutines Overview in *AIX General Programming Concepts : Writing and Debugging Programs*.

auditwrite Subroutine

Purpose

Writes an audit record.

Library

Security Library (**libc.a**)

Syntax

```
#include <sys/audit.h>
#include <stdio.h>

int auditwrite (Event, Result, Buffer1, Length1, Buffer2, Length2
, ...)
char *Event;
int Result;
char *Buffer1, *Buffer2 ...;
int Length1, Length2 ...;
```

Description

The **auditwrite** subroutine builds the tail of an audit record and then writes it with the **auditlog** subroutine. The tail is built by gathering the specified buffers. The last buffer pointer must be a null.

If the **auditwrite** subroutine is to be called from a program invoked from the **inittab** file, the **setpcred** subroutine should be called first to establish the process' credentials.

Parameters

Event	Specifies the name of the event to be logged.
Result	Specifies the audit status of the event. Valid values are defined in the **sys/audit.h** file and are listed in the **auditlog** subroutine.
Buffer1, Buffer2	Specifies the character buffers containing audit tail information. Note that numerical values must be passed by reference. The correct size can be computed with the **sizeof** C function.
Length1, Length2	Specifies the lengths of the corresponding buffers.

Return Values

If the **auditwrite** subroutine completes successfully, a value of 0 is returned. Otherwise, a value of –1 is returned and the **errno** global variable is set to indicate the error.

Error Codes

The **auditwrite** subroutine fails if the following is true:

ENOSPC	The **auditwrite** subroutine is unable to allocate space for the tail buffer.

Other error codes are returned by the **auditlog** subroutine.

Implementation Specifics

This subroutine is part of Base Operating System (BOS) Runtime.

Related Information

The **auditlog** subroutine, **setpcred** subroutine.

The **inittab** file.

List of Security and Auditing Subroutines and Subroutines Overview in *AIX General Programming Concepts : Writing and Debugging Programs*.

authenticate Subroutine

Purpose

Verifies a user's name and password.

Library

Security Library (**libc.a**)

Syntax

#include <stddef.h>

int authenticate (*UserName***,** *Response***,** *Reenter***,** *Message***)**
wchar_t ***UserName*;**
wchar_t ***Response*;**
int ***Reenter*;**
wchar_t *****Message*;**

Description

The **authenticate** subroutine maintains requirements users must satisfy to be authenticated to the system. It is a recallable interface that prompts for the user's name and password. The user must supply a character string at the prompt issued by the *Message* parameter. The *Response* parameter returns the user's response to the **authenticate** subroutine. The calling program makes no assumptions about the number of prompt messages the user must satisfy for authentication.

The *Reenter* parameter remains a nonzero value until the user satisfies all prompt messages or answers incorrectly. Once the *Reenter* parameter is zero, the return code signals whether authentication passed or failed.

The **authenticate** subroutine ascertains the authentication domains the user can attempt. The subroutine reads the **SYSTEM** line from the user's stanza in the **/etc/security/user** file. Each token that appears in the **SYSTEM** line corresponds to a method that can be dynamically loaded and processed. Likewise, the system can provide multiple or alternate authentication paths.

The **authenticate** routine maintains internal state information concerning the next prompt message presented to the user. If the calling program supplies a different user name before all prompts are complete for the user, the internal state information is reset and prompt messages begin again.

If the user has no defined password, or the **SYSTEM** grammar explicitly specifies no authentication required, the user is not required to respond to any prompt messages. Otherwise, the user is always initially prompted to supply a password.

The **authenticate** subroutine can be called initially with the cleartext password in the *Response* parameter. If the user supplies a password during the initial invocation but does not have a password, authentication fails. If the user wants the **authenticate** subroutine to supply a prompt message, the *Response* parameter is a null pointer on initial invocation.

The **authenticate** subroutine sets the **AUTHSTATE** environment variable used by name resolution subroutines, such as the **getpwnam** subroutine. This environment variable indicates the registry to which to user authenticated. Values for the **AUTHSTATE** environment variable include **DCE**, **compat**, and token names that appear in a **SYSTEM** grammar. A null value can exist if the **cron** daemon or other utilities that do not require authentication is called.

Parameters

UserName	Points to the user's name that is to be authenticated.
Response	Specifies a character string containing the user's response to an authentication prompt.
Reenter	Points to a Boolean value that signals whether the **authenticate** subroutine has completed processing. If the *Reenter* parameter is a nonzero value, the **authenticate** subroutine expects the user to satisfy the prompt message provided by the *Message* parameter. If the *Reenter* parameter is 0, the **authenticate** subroutine has completed processing.
Message	Points to a pointer that the **authenticate** subroutine allocates memory for and fills in. This string is suitable for printing and issues prompt messages (if the *Reenter* parameter is a nonzero value). It also issues informational messages such as why the user failed authentication (if the *Reenter* parameter is 0). The calling application is responsible for freeing this memory.

Return Values

Upon successful completion, the **authenticate** subroutine returns a value of 0. If this subroutine fails, it returns a value of 1.

Error Codes

The **authenticate** subroutine is unsuccessful if one of the following values is true:

ENOENT	Indicates that the user is unknown to the system.
ESAD	Indicates that authentication is denied.
EINVAL	Indicates that the parameters are not valid.
ENOMEN	Indicates that memory allocation (malloc) failed.

Note: The DCE mechanism requires credentials on successful authentication that apply only to the authenticate process and its children.

Implementation Specifics

This subroutine is part of Base Operating System (BOS) Runtime.

Related Information

The **ckuserID** subroutine.

basename Subroutine

Purpose

Return the last element of a path name.

Library

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

Syntax#include <libgen.h>char *basename (char *path)

Description

Given a pointer to a character string that contains a path name, the **basename** subroutine deletes trailing "/" characters from *path*, and then returns a pointer to the last component of *path*. The "/" character is defined as trailing if it is not the first character in the string.

If *path* is a null pointer or points to an empty string, a pointer to a static constant "." is returned.

Return Values

The **basename** function returns a pointer to the last component of *path*.

The **basename** function returns a pointer to a static constant "." if *path* is a null pointer or points to an empty string.

The **basename** function may modify the string pointed to by *path* and may return a pointer to static storage that may then be overwritten by a subsequent call to the **basename** subroutine.

Examples

Input string	Output string
"/usr/lib"	"lib"
"/usr/"	"usr"
"/"	"/"

Implementation Specifics

This subroutine is part of Base Operating System (BOS) Runtime.

Related Information

The **dirname** subroutine.

bcopy, bcmp, bzero or ffs Subroutine

Purpose

Performs bit and byte string operations.

Library

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

Syntax

#include <strings.h>

void bcopy (*Source***,** *Destination***,** *Length***)**
const void ***Source*,**
char ***Destination*;**
size_t *Length***;**

int bcmp (*String1***,** *String2***,** *Length***)**
const void ***String1*, ***String2***;**
size_t *Length***;**

void bzero (*String***,***Length***)**
char ***String*;**
int *Length***;**

int ffs (*Index***)**
int *Index***;**

Description

Note: The **bcopy** subroutine takes parameters backwards from the **strcpy** subroutine.

The **bcopy**, **bcmp**, and **bzero** subroutines operate on variable length strings of bytes. They do not check for null bytes as do the **string** routines.

The **bcopy** subroutine copies the value of the *Length* parameter in bytes from the string in the *Source* parameter to the string in the *Destination* parameter.

The **bcmp** subroutine compares the byte string in the *String1* parameter against the byte string of the *String2* parameter, returning a zero value if the two strings are identical and a nonzero value otherwise. Both strings are assumed to be *Length* bytes long.

The **bzero** subroutine zeroes out the string in the *String* parameter for the value of the *Length* parameter in bytes.

The **ffs** subroutine finds the first bit set in the *Index* parameter passed to it and returns the index of that bit. Bits are numbered starting at 1. A return value of 0 indicates that the value passed is 0.

Implementation Specifics

These subroutines are part of Base Operating System (BOS) Runtime.

Related Information

The **memcmp**, **memccpy**, **memchr**, **memcpy**, **memmove**, **memset** subroutines, **strcat**, **strncat**, **strxfrm**, **strcpy**, **strncpy**, or **strdup** subroutine, **strcmp**, **strncmp**, **strcasecmp**, **strncasecmp**, or **strcoll** subroutine, **strlen**, **strchr**, **strrchr**, **strpbrk**, **strspn**, **strcspn**, **strstr**, or **strtok** subroutine, **swab** subroutine.

List of String Manipulation Services and Subroutines Overview in *AIX General Programming Concepts : Writing and Debugging Programs*.

bessel: j0, j1, jn, y0, y1, or yn Subroutine

Purpose

Computes Bessel functions.

Libraries

```
IEEE Math Library (libm.a)
or System V Math Library (libmsaa.a)
```

Syntax

```
#include <math.h>

double j0 (x)
double x;

double j1 (x)
double x;

double jn (n, x)
int n;
double x;

double y0 (x)
double x;

double y1 (x)
double x;

double yn (n, x)
int n;
double x;
```

Description

Bessel functions are used to compute wave variables, primarily in the field of communications.

The **j0** subroutine and **j1** subroutine return Bessel functions of *x* of the first kind, of orders 0 and 1, respectively. The **jn** subroutine returns the Bessel function of *x* of the first kind of order *n*.

The **y0** subroutine and **y1** subroutine return the Bessel functions of *x* of the second kind, of orders 0 and 1, respectively. The **yn** subroutine returns the Bessel function of *x* of the second kind of order *n*. The value of *x* must be positive.

Note: Compile any routine that uses subroutines from the **libm.a** library with the **–lm** flag. To compile the **j0.c** file, for example:

```
cc j0.c –lm
```

Parameters

x	Specifies some double–precision floating–point value.
n	Specifies some integer value.

Return Values

When using **libm.a** (**–lm**), if *x* is negative, **y0**, **y1**, and **yn** return the value NaNQ. If *x* is 0, **y0**, **y1**, and **yn** return the value **–HUGE_VAL**.

When using **libmsaa.a** (**–lmsaa**), values too large in magnitude cause the functions **j0**, **j1**, **y0**, and **y1** to return 0 and to set the **errno** global variable to ERANGE. In addition, a message indicating TLOSS error is printed on the standard error output.

Nonpositive values cause **y0**, **y1**, and **yn** to return the value **–HUGE** and to set the **errno** global variable to **EDOM**. In addition, a message indicating argument DOMAIN error is printed on the standard error output.

These error–handling procedures may be changed with the **matherr** subroutine when using **libmsaa.a** (**–lmsaa**).

Implementation Specifics

These subroutines are part of Base Operating System (BOS) Runtime.

Related Information

The **matherr** subroutine.

Subroutines Overview in *AIX General Programming Concepts : Writing and Debugging Programs*.

bindprocessor Subroutine

Purpose

Binds kernel threads to a processor.

Library

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

Syntax

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

int bindprocessor (What, Who, Where)
int What;
int Who;
cpu_t Where;
```

Description

The **bindprocessor** subroutine binds a single kernel thread, or all kernel threads in a process, to a processor, forcing the bound threads to be scheduled to run on that processor. It is important to understand that a process itself is not bound, but rather its kernel threads are bound. Once kernel threads are bound, they are always scheduled to run on the chosen processor, unless they are later unbound. When a new thread is created, it has the same bind properties as its creator. This applies to the initial thread in the new process created by the **fork** subroutine: the new thread inherits the bind properties of the thread which called **fork**. When the **exec** subroutine is called, thread properties are left unchanged.

Parameters

What Specifies whether a process or a thread is being bound to a processor. The *What* parameter can take one of the following values:

BINDPROCESS A process is being bound to a processor.

BINDTHREAD A thread is being bound to a processor.

Who Indicates a process or thread identifier, as appropriate for the *What* parameter, specifying the process or thread which is to be bound to a processor.

Where If the *Where* parameter is a logical processor identifier, it specifies the processor to which the process or thread is to be bound. A value of **PROCESSOR_CLASS_ANY** unbinds the specified process or thread, which will then be able to run on any processor.

The **sysconf** subroutine can be used to retrieve information about the number of processors in the system.

Return Values

On successful completion, the **bindprocessor** subroutine returns 0. Otherwise, a value of –1 is returned, and the **errno** global variable is set to indicate the error.

Error Codes

The **bindprocessor** subroutine is unsuccessful if one of the following is true:

EINVAL	The *What* parameter is invalid, or the *Where* parameter indicates an invalid processor number or a processor class which is not currently available.
ESRCH	The specified process or thread does not exist.
EPERM	The caller does not have root user authority, and the *Who* parameter specifies either a process, or a thread belonging to a process, having a real or effective user ID different from that of the calling process.

Implementation Specifics

The **bindprocessor** subroutine is part of the Base Operating System (BOS) Runtime.

Related Information

The **bindprocessor** command.

The **exec** subroutine, **fork** subroutine, **sysconf** subroutine, **thread_self** subroutine.

Controlling Processor Use in *AIX General Programming Concepts : Writing and Debugging Programs*.

brk or sbrk Subroutine

Purpose

Changes data segment space allocation.

Library

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

Syntax

```
#include <unistd .h>


int brk (EndDataSegment)
char *EndDataSegment;

void *sbrk (Increment)
intptr_t Increment;
```

Description

The **brk** and **sbrk** subroutines dynamically change the amount of space allocated for the data segment of the calling process. (For information about segments, see the **exec** subroutine. For information about the maximum amount of space that can be allocated, see the **ulimit** and **getrlimit** subroutines.)

The change is made by resetting the break value of the process, which determines the maximum space that can be allocated. The break value is the address of the first location beyond the current end of the data region. The amount of available space increases as the break value increases. The available space is initialized to a value of 0 at the time it is used. The break value can be automatically rounded up to a size appropriate for the memory management architecture.

The **brk** subroutine sets the break value to the value of the *EndDataSegment* parameter and changes the amount of available space accordingly.

The **sbrk** subroutine adds to the break value the number of bytes contained in the *Increment* parameter and changes the amount of available space accordingly. The *Increment* parameter can be a negative number, in which case the amount of available space is decreased.

Parameters

EndDataSegment	Specifies the effective address of the maximum available data.
Increment	Specifies any integer.

Return Values

Upon successful completion, the **brk** subroutine returns a value of 0, and the **sbrk** subroutine returns the old break value. If either subroutine is unsuccessful, a value of –1 is returned and the **errno** global variable is set to indicate the error.

Error Codes

The **brk** subroutine and the **sbrk** subroutine are unsuccessful and the allocated space remains unchanged if one or more of the following are true:

ENOMEM The requested change allocates more space than is allowed by a system–imposed maximum. (For information on the system–imposed maximum on memory space, see the **ulimit** system call.)

ENOMEM The requested change sets the break value to a value greater than or equal to the start address of any attached shared–memory segment. (For information on shared memory operations, see the **shmat** subroutine.)

Implementation Specifics

These subroutines are part of Base Operating System (BOS) Runtime.

Related Information

The **exec** subroutines, **getrlimit** subroutine, **shmat** subroutine, **shmdt** subroutine, **ulimit** subroutine.

The **_end**, **_etext**, or **_edata** identifier.

Subroutine Overview in *AIX General Programming Concepts : Writing and Debugging Programs*.

bsearch Subroutine

Purpose

Performs a binary search.

Library

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

Syntax

```
#include <stdlib.h>

void *bsearch (Key, Base, NumberOfElements, Size,
ComparisonPointer)

const void *Key;
const void *Base;
size_t NumberOfElements;
size_t Size;
int (*ComparisonPointer) (const void *, const void *);
```

Description

The **bsearch** subroutine is a binary search routine.

The **bsearch** subroutine searches an array of *NumberOfElements* objects, the initial member of which is pointed to by the *Base* parameter, for a member that matches the object pointed to by the *Key* parameter. The size of each member in the array is specified by the *Size* parameter.

The array must already be sorted in increasing order according to the provided comparison function *ComparisonPointer* parameter.

Parameters

Key	Points to the object to be sought in the array.
Base	Points to the element at the base of the table.
NumberOfElements	Specifies the number of elements in the array.
ComparisonPointer	Points to the comparison function, which is called with two arguments that point to the *Key* parameter object and to an array member, in that order.
Size	Specifies the size of each member in the array.

Return Values

If the *Key* parameter value is found in the table, the **bsearch** subroutine returns a pointer to the element found.

If the *Key* parameter value is not found in the table, the **bsearch** subroutine returns the null value. If two members compare as equal, the matching member is unspecified.

For the *ComparisonPointer* parameter, the comparison function compares its parameters and returns a value as follows:

- If the first parameter is less than the second parameter, the *ComparisonPointer* parameter returns a value less than 0.
- If the first parameter is equal to the second parameter, the *ComparisonPointer* parameter returns a value of 0.

- If the first parameter is greater than the second parameter, the *ComparisonPointer* parameter returns a value greater than 0.

The comparison function need not compare every byte, so arbitrary data can be contained in the elements in addition to the values being compared.

The *Key* and *Base* parameters should be of type pointer–to–element and cast to type pointer–to–character. Although declared as type pointer–to–character, the value returned should be cast into type pointer–to–element.

Implementation Specifics

This subroutine is part of Base Operating System (BOS) Runtime.

Related Information

The **hsearch** subroutine, **lsearch** subroutine, **qsort** subroutine.

Knuth, Donald E.; *The Art of Computer Programming*, Volume 3. Reading, Massachusetts, Addison–Wesley, 1981.

Searching and Sorting Example Program and Subroutines Overview in *AIX General Programming Concepts : Writing and Debugging Programs*.

btowc Subroutine

Purpose

Single–byte to wide–character conversion.

Library

Standard Library (**libc.a**)

Syntax

```
#include <stdio.h>
#include <wchar.h>

wint_t btowc (int c);
```

Description

The *btowc* function determines whether c constitutes a valid (one–byte) character in the initial shift state.

The behavior of this function is affected by the LC_CTYPE category of the current locale.

Return Values

The btowc function returns WEOF if c has the value EOF or if (unsigned char) c does not constitute a valid (one–byte) character in the initial shift state. Otherwise, it returns the wide–character representation of that character.

Implementation Specifics

This subroutine is part of Base Operating System (BOS) subroutine.

Related Information

The wctob subroutine, the **wchar.h** file.

_check_lock Subroutine

Purpose

Conditionally updates a single word variable atomically.

Library

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

Syntax

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

boolean_t _check_lock (word_addr, old_val, new_val)
atomic_p word_addr;
int old_val;
int new_val;
```

Parameters

word_addr	Specifies the address of the single word variable.
old_val	Specifies the old value to be checked against the value of the single word variable.
new_val	Specifies the new value to be conditionally assigned to the single word variable.

Description

The **_check_lock** subroutine performs an atomic (uninterruptible) sequence of operations. The **compare_and_swap** subroutine is similar, but does not issue synchronization instructions and therefore is inappropriate for updating lock words.

Note: The word variable must be aligned on a full word boundary.

Return Values

FALSE	Indicates that the single word variable was equal to the old value and has been set to the new value.
TRUE	Indicates that the single word variable was not equal to the old value and has been left unchanged.

Related Information

The **_clear_lock** subroutine, **_safe_fetch** subroutine.

_clear_lock Subroutine

Purpose

Stores a value in a single word variable atomically.

Library

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

Syntax

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

void _clear_lock (word_addr, value)
atomic_p word_addr;
int value
```

Parameters

word_addr	Specifies the address of the single word variable.
value	Specifies the value to store in the single word variable.

Description

The **_clear_lock** subroutine performs an atomic (uninterruptible) sequence of operations.

This subroutine has no return values.

Note: The word variable must be aligned on a full word boundary.

Related Information

The **_check_lock** subroutine, **_safe_fetch** subroutine.

catclose Subroutine

Purpose

Closes a specified message catalog.

Library

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

Syntax

#include <nl_types.h>

```
int catclose (CatalogDescriptor)
nl_catd CatalogDescriptor;
```

Description

The **catclose** subroutine closes a specified message catalog. If your program accesses several message catalogs and you reach the maximum number of opened catalogs (specified by the **NL_MAXOPEN** constant), you must close some catalogs before opening additional ones. If you use a file descriptor to implement the **nl_catd** data type, the **catclose** subroutine closes that file descriptor.

The **catclose** subroutine closes a message catalog only when the number of calls it receives matches the total number of calls to the **catopen** subroutine in an application. All message buffer pointers obtained by prior calls to the **catgets** subroutine are not valid when the message catalog is closed.

Parameters

CatalogDescriptor	Points to the message catalog returned from a call to the **catopen** subroutine.

Return Values

The **catclose** subroutine returns a value of 0 if it closes the catalog successfully, or if the number of calls it receives is fewer than the number of calls to the **catopen** subroutine.

The **catclose** subroutine returns a value of –1 if it does not succeed in closing the catalog. The **catclose** subroutine is unsuccessful if the number of calls it receives is greater than the number of calls to the **catopen** subroutine, or if the value of the *CatalogDescriptor* parameter is not valid.

Implementation Specifics

This subroutine is part of Base Operating System (BOS) Runtime.

Related Information

The **catgets** subroutine, **catopen** subroutine.

For more information about the Message Facility, see Message Facility Overview for Programming in *AIX General Programming Concepts : Writing and Debugging Programs*.

For more information about subroutines and libraries, see Subroutines Overview in *AIX General Programming Concepts : Writing and Debugging Programs*.

catgets Subroutine

Purpose

Retrieves a message from a catalog.

Library

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

Syntax

```
#include <nl_types>

char *catgets (CatalogDescriptor, SetNumber, MessageNumber, Strin
g)
nl_catd CatalogDescriptor;
int SetNumber, MessageNumber;
const char *String;
```

Description

The **catgets** subroutine retrieves a message from a catalog after a successful call to the **catopen** subroutine. If the **catgets** subroutine finds the specified message, it loads it into an internal character string buffer, ends the message string with a null character, and returns a pointer to the buffer.

The **catgets** subroutine uses the returned pointer to reference the buffer and display the message. However, the buffer can not be referenced after the catalog is closed.

Parameters

CatalogDescriptor	Specifies a catalog description that is returned by the **catopen** subroutine.
SetNumber	Specifies the set ID.
MessageNumber	Specifies the message ID. The *SetNumber* and *MessageNumber* parameters specify a particular message to retrieve in the catalog.
String	Specifies the default character–string buffer.

Return Values

If the **catgets** subroutine is unsuccessful for any reason, it returns the user–supplied default message string specified by the *String* parameter.

Implementation Specifics

This subroutine is part of Base Operating System (BOS) Runtime.

Related Information

The **catclose** subroutine, **catopen** subroutine.

For more information about the Message Facility, see Message Facility Overview for Programming in *AIX General Programming Concepts : Writing and Debugging Programs*.

For more information about subroutines and libraries, see Subroutines Overview in *AIX General Programming Concepts : Writing and Debugging Programs*.

catopen Subroutine

Purpose

Opens a specified message catalog.

Library

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

Syntax

```
#include <nl_types.h>

nl_catd catopen (CatalogName, Parameter)
const char *CatalogName;
int Parameter;
```

Description

The **catopen** subroutine opens a specified message catalog and returns a catalog descriptor used to retrieve messages from the catalog. The contents of the catalog descriptor are complete when the **catgets** subroutine accesses the message catalog. The **nl_catd** data type is used for catalog descriptors and is defined in the **nl_types.h** file.

If the catalog file name referred to by the *CatalogName* parameter contains a leading / (slash), it is assumed to be an absolute path name. If the catalog file name is not an absolute path name, the user environment determines which directory paths to search. The **NLSPATH** environment variable defines the directory search path. When this variable is used, the **setlocale** subroutine must be called before the **catopen** subroutine.

A message catalog descriptor remains valid in a process until that process or a successful call to one of the **exec** functions closes it.

You can use two special variables, **%N** and **%L**, in the **NLSPATH** environment variable. The **%N** variable is replaced by the catalog name referred to by the call that opens the message catalog. The **%L** variable is replaced by the value of the **LC_MESSAGES** category.

The value of the **LC_MESSAGES** category can be set by specifying values for the **LANG**, **LC_ALL**, or **LC_MESSAGES** environment variable. The value of the **LC_MESSAGES** category indicates which locale–specific directory to search for message catalogs. For example, if the **catopen** subroutine specifies a catalog with the name `mycmd`, and the environment variables are set as follows:

```
NLSPATH=../%N:./%N:/system/nls/%L/%N:/system/nls/%N LANG=fr_FR
```

then the application searches for the catalog in the following order:

```
../mycmd
./mycmd
/system/nls/fr_FR/mycmd
/system/nls/mycmd
```

If you omit the **%N** variable in a directory specification within the **NLSPATH** environment variable, the application assumes that it defines a catalog name and opens it as such and will not traverse the rest of the search path.

If the **NLSPATH** environment variable is not defined, the **catopen** subroutine uses the default path. See the **/etc/environment** file for the **NLSPATH** default path. If the **LC_MESSAGES** category is set to the default value C, and the **LC__FASTMSG** environment variable is set to `true`, then subsequent calls to the **catgets** subroutine generate pointers to the program–supplied default text.

The **catopen** subroutine treats the first file it finds as a message file. If you specify a non–message file in a **NLSPATH**, for example, **/usr/bin/ls**, **catopen** treats **/usr/bin/ls** as a message catalog. Thus no messages are found and default messages are returned. If you specify /**tmp** in a **NLSPATH**, /**tmp** is opened and searched for messages and default messages are displayed.

Parameters

CatalogName	Specifies the catalog file to open.
Parameter	Determines the environment variable to use in locating the message catalog. If the value of the *Parameter* parameter is 0, use the **LANG** environment variable without regard to the **LC_MESSAGES** category to locate the catalog. If the value of the *Parameter* parameter is the **NL_CAT_LOCALE** macro, use the **LC_MESSAGES** category to locate the catalog.

Return Values

The **catopen** subroutine returns a catalog descriptor. If the **LC_MESSAGES** category is set to the default value C, and the **LC__FASTMSG** environment variable is set to `true`, the **catopen** subroutine returns a value of –1.

If the **LC_MESSAGES** category is not set to the default value C but the **catopen** subroutine returns a value of –1, an error has occurred during creation of the structure of the **nl_catd** data type or the catalog name referred to by the *CatalogName* parameter does not exist.

Implementation Specifics

This subroutine is part of Base Operating System (BOS) Runtime.

Related Information

The **catclose** subroutine, **catgets** subroutine, **exec** subroutines, **setlocale** subroutine.

The **environment** file.

For more information about the Message Facility, see the Message Facility Overview for Programming in *AIX General Programming Concepts : Writing and Debugging Programs*.

For more information about subroutines and libraries, see the Subroutines Overview in *AIX General Programming Concepts : Writing and Debugging Programs*.

ccsidtocs or cstoccsid Subroutine

Purpose

Provides conversion between coded character set IDs (CCSID) and code set names.

Library

The iconv Library (**libiconv.a**)

Syntax

```
#include <iconv.h>
```

```
CCSID cstoccsid (*Codeset)
const char *Codeset;
```

```
char *ccsidtocs (CCSID)
CCSID CCSID;
```

Description

The **cstoccsid** subroutine returns the CCSID of the code set specified by the *Codeset* parameter. The **ccsidtocs** subroutine returns the code set name of the CCSID specified by *CCSID* parameter. CCSIDs are registered Bull coded character set IDs.

Parameters

Codeset	Specifies the code set name to be converted to its corresponding CCSID.
CCSID	Specifies the CCSID to be converted to its corresponding code set name.

Return Values

If the code set is recognized by the system, the **cstoccsid** subroutine returns the corresponding CCSID. Otherwise, null is returned.

If the CCSID is recognized by the system, the **ccsidtocs** subroutine returns the corresponding code set name. Otherwise, a null pointer is returned.

Implementation Specifics

These subroutines are part of Base Operating System (BOS) Runtime.

Related Information

For more information about code set conversion, see Converters Overview for Programming in *AIX General Programming Concepts : Writing and Debugging Programs*.

The National Language Support Overview for Programming in *AIX General Programming Concepts : Writing and Debugging Programs*.

Subroutines Overview in *AIX General Programming Concepts : Writing and Debugging Programs*.

cfgetospeed, cfsetospeed, cfgetispeed, or cfsetispeed Subroutine

Purpose

Gets and sets input and output baud rates.

Library

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

Syntax

```
#include <termios.h>

speed_t cfgetospeed (TermiosPointer)
const struct termios *TermiosPointer;

int cfsetospeed (TermiosPointer, Speed)
struct termios *TermiosPointer;
speed_t Speed;

speed_t cfgetispeed (TermiosPointer)
const struct termios *TermiosPointer;

int cfsetispeed (TermiosPointer, Speed)
struct termios *TermiosPointer;
speed_t Speed;
```

Description

The baud rate subroutines are provided for getting and setting the values of the input and output baud rates in the **termios** structure. The effects on the terminal device described below do not become effective and not all errors are detected until the **tcsetattr** function is successfully called.

The input and output baud rates are stored in the **termios** structure. The supported values for the baud rates are shown in the table that follows this discussion.

The **termios.h** file defines the type **speed_t** as an unsigned integral type.

The **cfgetospeed** subroutine returns the output baud rate stored in the **termios** structure pointed to by the *TermiosPointer* parameter.

The **cfsetospeed** subroutine sets the output baud rate stored in the **termios** structure pointed to by the *TermiosPointer* parameter to the value specified by the *Speed* parameter.

The **cfgetispeed** subroutine returns the input baud rate stored in the **termios** structure pointed to by the *TermiosPointer* parameter.

The **cfsetispeed** subroutine sets the input baud rate stored in the **termios** structure pointed to by the *TermiosPointer* parameter to the value specified by the *Speed* parameter.

Certain values for speeds have special meanings when set in the **termios** structure and passed to the **tcsetattr** function. These values are discussed in the **tcsetattr** subroutine**.**

The following table lists possible baud rates:

Baud Rate Values			
Name	**Description**	**Name**	**Description**
B0	Hang up	B600	600 baud
B5	50 baud	B1200	1200 baud
B75	75 baud	B1800	1800 baud
B110	110 baud	B2400	2400 baud
B134	134 baud	B4800	4800 baud
B150	150 baud	B9600	9600 baud
B200	200 baud	B19200	19200 baud
B300	300 baud	B38400	38400 baud

The **termios.h** file defines the name symbols of the table.

Parameters

TermiosPointer Points to a **termios** structure.

Speed Specifies the baud rate.

Return Values

The **cfgetospeed** and **cfgetispeed** subroutines return exactly the value found in the **termios** data structure, without interpretation.

Both the **cfsetospeed** and **cfsetispeed** subroutines return a value of 0 if successful and –1 if unsuccessful.

Examples

To set the output baud rate to 0 (which forces modem control lines to stop being asserted), enter:

```
cfsetospeed (&my_termios, B0);
tcsetattr (stdout, TCSADRAIN, &my_termios);
```

Implementation Specifics

These subroutines are part of Base Operating System (BOS) Runtime.

Related Information

The **tcsetattr** subroutine.

The **termios.h** file.

Input and Output Handling Programmer's Overview in *AIX General Programming Concepts : Writing and Debugging Programs*.

chacl or fchacl Subroutine

Purpose

Changes the permissions on a file.

Library

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

Syntax

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

int chacl (Path, ACL, ACLSize)
char *Path;
struct acl *ACL;
int ACLSize;

int fchacl (FileDescriptor, ACL, ACLSize)
int FileDescriptor;
struct acl *ACL;
int ACLSize;
```

Description

The **chacl** and **fchacl** subroutines set the access control attributes of a file according to the Access Control List (ACL) structure pointed to by the *ACL* parameter.

Parameters

Path
: Specifies the path name of the file.

ACL
: Specifies the ACL to be established on the file. The format of an ACL is defined in the **sys/acl.h** file and contains the following members:

`acl_len`
: Specifies the size of the ACL (Access Control List) in bytes, including the base entries.

Note: The entire ACL for a file cannot exceed one memory page (4096 bytes).

`acl_mode`
: Specifies the file mode.

The following bits in the **acl_mode** member are defined in the **sys/mode.h** file and are significant for this subroutine:

S_ISUID
: Enables the **setuid** attribute on an executable file.

S_ISGID
: Enables the **setgid** attribute on an executable file. Enables the group–inheritance attribute on a directory.

S_ISVTX
: Enables linking restrictions on a directory.

S_IXACL
: Enables extended ACL entry processing. If this attribute is not set, only the base entries (owner, group, and default) are used for access authorization checks.

Other bits in the mode, including the following, are ignored:

u_access
: Specifies access permissions for the file owner.

g_access
: Specifies access permissions for the file group.

o_access
: Specifies access permissions for the default class of *others*.

acl_ext[]	Specifies an array of the extended entries for this access control list.

The members for the base ACL (owner, group, and others) can contain the following bits, which are defined in the **sys/access.h** file:

R_ACC	Allows read permission.
W_ACC	Allows write permission.
X_ACC	Allows execute or search permission.

FileDescriptor	Specifies the file descriptor of an open file.
ACLSize	Specifies the size of the buffer containing the ACL.

Note: The **chacl** subroutine requires the *Path*, *ACL*, and *ACLSize* parameters. The **fchacl** subroutine requires the *FileDescriptor*, *ACL*, and *ACLSize* parameters.

ACL Data Structure for chacl

Each access control list structure consists of one **struct acl** structure containing one or more **struct acl_entry** structures with one or more **struct** *ace_id* structures.

If the **struct ace_id** structure has *id_type* set to **ACEID_USER** or **ACEID_GROUP**, there is only one *id_data* element. To add multiple IDs to an ACL you must specify multiple **struct ace_id** structures when *id_type* is set to **ACEID_USER** or **ACEID_GROUP**. In this case, no error is returned for the multiple elements, and the access checking examines only the first element. Specifically, the errno value **EINVAL** is not returned for *acl_len* being incorrect in the ACL structure although more than one uid or gid is specified.

Return Values

Upon successful completion, the **chacl** and **fchacl** subroutines return a value of 0. If the **chacl** or **fchacl** subroutine fails, a value of –1 is returned, and the **errno** global variable is set to indicate the error.

Error Codes

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

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

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

EROFS	The file specified by the *Path* parameter resides on a read–only file system.
EFAULT	The *ACL* parameter points to a location outside of the allocated address space of the process.
EINVAL	The *ACL* parameter does not point to a valid ACL.
EINVAL	The `acl_len` member in the ACL is not valid.
EIO	An I/O error occurred during the operation.
ENOSPC	The size of the *ACL* parameter exceeds the system limit of one memory page (4KB).
EPERM	The effective user ID does not match the ID of the owner of the file, and the invoker does not have root user authority.

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

EBADF	The file descriptor *FileDescriptor* is not valid.

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

ETIMEDOUT	The connection timed out.

Security

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

Auditing Events:

Event	Information
chacl	*Path*
fchacl	*FileDescriptor*

Implementation Specifics

These subroutines are part of Base Operating System (BOS) Runtime.

Related Information

The **acl_chg** subroutine, **acl_get** subroutine, **acl_put** subroutine, **acl_set** subroutine, **chmod** subroutine, **stat** subroutine, **statacl** subroutine.

The **aclget** command, **aclput** command.

List of Security and Auditing Subroutines and Subroutines Overview in *AIX General Programming Concepts : Writing and Debugging Programs*.

chdir Subroutine

Purpose

Changes the current directory.

Library

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

Syntax

```
#include <unistd.h>

int chdir (Path)
const char *Path;
```

Description

The **chdir** subroutine changes the current directory to the directory indicated by the *Path* parameter.

Parameters

Path	A pointer to the path name of the directory. If the *Path* parameter refers to a symbolic link, the **chdir** subroutine sets the current directory to the directory pointed to by the symbolic link. If Network File System (NFS) is installed on the system, this path can cross into another node.

The current directory, also called the current working directory, is the starting point of searches for path names that do not begin with a / (slash). The calling process must have search access to the directory specified by the *Path* parameter.

Return Values

Upon successful completion, the **chdir** subroutine returns a value of 0. Otherwise, a value of –1 is returned and the **errno** global variable is set to identify the error.

Error Codes

The **chdir** subroutine fails and the current directory remains unchanged if one or more of the following are true:

EACCES	Search access is denied for the named directory.
ENOENT	The named directory does not exist.
ENOTDIR	The path name is not a directory.

The **chdir** subroutine can also be unsuccessful for other reasons. See ”Appendix A. Base Operating System Error Codes for Services That Require Path–Name Resolution”, on page A-1 for a list of additional error codes.

If NFS is installed on the system, the **chdir** subroutine can also fail if the following is true:

ETIMEDOUT	The connection timed out.

Implementation Specifics

This subroutine is part of Base Operating System (BOS) Runtime.

Related Information

The **chroot** subroutine.

The **cd** command.

Base Operating System Error Codes for Services That Require Path–Name Resolution, on page A-1.

Files, Directories, and File Systems for Programmers in *AIX General Programming Concepts : Writing and Debugging Programs*.

chmod or fchmod Subroutine

Purpose

Changes file access permissions.

Library

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

Syntax

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

int chmod (Path, Mode)
const char *Path;
mode_t Mode;

int fchmod (FileDescriptor, Mode)
int FileDescriptor;
mode_t Mode;
```

Description

The **chmod** subroutine sets the access permissions of the file specified by the *Path* parameter. If Network File System (NFS) is installed on your system, this path can cross into another node.

Use the **fchmod** subroutine to set the access permissions of an open file pointed to by the *FileDescriptor* parameter.

The access control information is set according to the *Mode* parameter.

Parameters

FileDescriptor — Specifies the file descriptor of an open file.

Mode — Specifies the bit pattern that determines the access permissions. The *Mode* parameter is constructed by logically ORing one or more of the following values, which are defined in the **sys/mode.h** file:

S_ISUID — Enables the **setuid** attribute for an executable file. A process executing this program acquires the access rights of the owner of the file.

S_ISGID — Enables the **setgid** attribute for an executable file. A process executing this program acquires the access rights of the group of the file. Also, enables the group–inheritance attribute for a directory. Files created in this directory have a group equal to the group of the directory.

The following attributes apply only to files that are directly executable. They have no meaning when applied to executable text files such as shell scripts and **awk** scripts.

S_ISVTX — Enables the **link/unlink** attribute for a directory. Files cannot be linked to in this directory. Files can only be unlinked if the requesting process has write permission for the directory and is either the owner of the file or the directory.

S_ISVTX — Enables the **save text** attribute for an executable file. The program is not unmapped after usage.

S_ENFMT Enables enforcement–mode record locking for a regular file. File locks requested with the **lockf** subroutine are enforced.

S_IRUSR Permits the file's owner to read it.

S_IWUSR Permits the file's owner to write to it.

S_IXUSR Permits the file's owner to execute it (or to search the directory).

S_IRGRP Permits the file's group to read it.

S_IWGRP Permits the file's group to write to it.

S_IXGRP Permits the file's group to execute it (or to search the directory).

S_IROTH Permits others to read the file.

S_IWOTH Permits others to write to the file.

S_IXOTH Permits others to execute the file (or to search the directory).

Other mode values exist that can be set with the **mknod** subroutine but not with the **chmod** subroutine.

Path Specifies the full path name of the file.

Return Values

Upon successful completion, the **chmod** subroutine and **fchmod** subroutines return a value of 0. If the **chmod** subroutine or **fchmod** subroutine is unsuccessful, a value of –1 is returned, and the **errno** global variable is set to identify the error.

Error Codes

The **chmod** subroutine is unsuccessful and the file permissions remain unchanged if one of the following is true:

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

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

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

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

ENOENT The named file does not exist.

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

The **fchmod** subroutine is unsuccessful and the file permissions remain unchanged if the following is true:

EBADF The value of the *FileDescriptor* parameter is not valid.

The **chmod** or **fchmod** subroutine is unsuccessful and the access control information for a file remains unchanged if one of the following is true:

EPERM The effective user ID does not match the owner of the file, and the process does not have appropriate privileges.

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

EIO An I/O error occurred during the operation.

If NFS is installed on your system, the **chmod** and **fchmod** subroutines can also be unsuccessful if the following is true:

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

ETIMEDOUT The connection timed out.

Security

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

If you receive the **EBUSY** error, toggle the **enforced locking** attribute in the *Mode* parameter and retry your operation. The **enforced locking** attribute should never be used on a file that is part of the Trusted Computing Base.

Implementation Specifics

These subroutines are part of Base Operating System (BOS) Runtime.

Related Information

The **acl_chg** subroutine, **acl_get** subroutine, **acl_put** subroutine, **acl_set** subroutine, **chacl** subroutine, **statacl** subroutine, **stat** subroutine.

The **aclget** command, **aclput** command, **chmod** command.

List of Security and Auditing Subroutines and Subroutines Overview in *AIX General Programming Concepts : Writing and Debugging Programs*.

chown, fchown, lchown, chownx, or fchownx Subroutine

Purpose

Changes file ownership.

Library

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

Syntax

Syntax for the **chown**, **fchown**, and **lchown** Subroutines: **#include <sys/types.h>**
#include <unistd.h>

int chown (*Path***,** *Owner***,** *Group***)**
const char ****Path*;**
uid_t *Owner***;**
gid_t *Group***;**

int fchown (*FileDescriptor***,** *Owner***,** *Group***)**
int *FileDescriptor***;**
uid_t *Owner***;**
gid_t *Group***;**

int lchown (*Path***,** *Owner***,** *Group***)**
const char **fname*
uid_t *uid*
gid_t*gid*

Syntax for the **chownx** and **fchownx** Subroutines: **#include <sys/types.h>**
#include <sys/chownx.h>

int chownx (*Path***,** *Owner***,** *Group***,** *Flags***)**
char ****Path*;**
uid_t *Owner***;**
gid_t *Group***;**
int *Flags***;**

int fchownx (*FileDescriptor***,** *Owner***,** *Group***,** *Flags***)**
int *FileDescriptor***;**
uid_t *Owner***;**
gid_t *Group***;**
int *Flags***;**

Description

The **chown**, **chownx**, **fchown**, **fchownx**, and **lchown** subroutines set the file owner and group IDs of the specified file system object. Root user authority is required to change the owner of a file.

A function **lchown** function sets the owner ID and group ID of the named file similarity to **chown** function except in the case where the named file is a symbolic link. In this case **lchown** function changes the ownership of the symbolic link file itself, while **chown** function changes the ownership of the file or directory to which the symbolic link refers.

Parameters

FileDescriptor	Specifies the file descriptor of an open file.
Flags	Specifies whether the file owner ID or group ID should be changed. This parameter is constructed by logically ORing the following values:

T_OWNER_AS_IS	Ignores the value specified by the *Owner* parameter and leaves the owner ID of the file unaltered.
T_GROUP_AS_IS	Ignores the value specified by the *Group* parameter and leaves the group ID of the file unaltered.

Group	Specifies the new group of the file. If this value is –1, the group is not changed. (A value of –1 indicates only that the group is not changed; it does not indicate a group that is not valid. An owner or group ID cannot be invalid.)
Owner	Specifies the new owner of the file. If this value is –1, the owner is not changed. (A value of –1 indicates only that the group is not changed; it does not indicate a group that is not valid. An owner or group ID cannot be invalid.)
Path	Specifies the full path name of the file. If *Path* resolves to a symbolic link, the ownership of the file or directory pointed to by the symbolic link is changed.

Return Values

Upon successful completion, the **chown**, **chownx**, **fchown**, **fchownx**, and **lchown** subroutines return a value of 0. If the **chown**, **chownx**, **fchown**, **fchownx**, or **lchown** subroutine is unsuccessful, a value of –1 is returned and the **errno** global variable is set to indicate the error.

Error Codes

The **chown, chownx**, or **lchown** subroutine is unsuccessful and the owner and group of a file remain unchanged if one of the following is true:

EACCESS	Search permission is denied on a component of the *Path* parameter.
EDQUOT	The new group for the file system object cannot be set because the group's quota of disk blocks or i–nodes has been exhausted on the file system.
EFAULT	The *Path* parameter points to a location outside of the allocated address space of the process.
EINVAL	The owner or group ID supplied is not valid.
ELOOP	Too many symbolic links were encountered in translating the *Path* parameter.
ENAMETOOLONG	A component of the *Path* parameter exceeded 255 characters, or the entire *Path* parameter exceeded 1023 characters.
ENOENT	A symbolic link was named, but the file to which it refers does not exist; or a component of the *Path* parameter does not exist; or the process has the **disallow truncation** attribute set; or the *Path* parameter is null.
ENOTDIR	A component of the path prefix is not a directory.
EPERM	The effective user ID does not match the owner of the file, and the calling process does not have the appropriate privileges.

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

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

The **fchown** or **fchownx** subroutine is unsuccessful and the file owner and group remain unchanged if one of the following is true:

EBADF The named file resides on a read–only file system.

EDQUOT The new group for the file system object cannot be set because the group's quota of disk blocks or i–nodes has been exhausted on the file system.

EIO An I/O error occurred during the operation.

Security

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

chpass Subroutine

Purpose

Changes file access permissions.

Library

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

Thread Safe Security Library (**libs_r.a**)

Syntax

```
#include <stddef.h>

int chpass (UserName, Response, Reenter, Message)
wchar_t *UserName;
wchar_t *Response;
int *Reenter;
wchar_t **Message;
```

Description

The **chpass** subroutine maintains the requirements that the user must meet to change a password. This subroutine is the basic building block for changing passwords and handles password changes for local, NIS, and DCE user passwords.

The *Message* parameter provides a series of messages asking for old and new passwords, or providing informational messages, such as the reason for a password change failing. The first *Message* prompt is a prompt for the old password. This parameter does not prompt for the old password if the user has a real user ID of 0 (zero) and is changing a local user, or if the user has no current password. The **chpass** subroutine does not prompt a user with root authority for an old password. It informs the program that no message was sent and that it should invoke **chpass** again. If the user satisfies the first *Message* parameter's prompt, the system prompts the user to enter the new password. Each message is contained in the *Message* parameter and is displayed to the user. The *Response* parameter returns the user's response to the **chpass** subroutine.

The *Reenter* parameter remains a nonzero value until the user satisfies all of the prompt messages or until the user incorrectly responds to a prompt message. Once the *Reenter* parameter is 0, the return code signals whether the password change completed or failed.

The **chpass** subroutine maintains internal state information concerning the next prompt message to present to the user. If the calling program supplies a different user name before all prompt messages are complete for the user, the internal state information is reset and prompt messages begin again.

The **chpass** subroutine determines the administration domain to use during password changes. It determines if the user is defined locally, defined in Network Information Service (NIS), or defined in Distributed Computing Environment (DCE). Password changes occur only in these domains. System administrators may override this convention with the registry value in the **/etc/security/user** file. If the registry value is defined, the password change can only occur in the specified domain. System administrators can use this registry value if the user is administered on a remote machine that periodically goes down. If the user is allowed to log in through some other authentication method while the server is down, password changes remain to follow only the primary server.

The **chpass** subroutine allows the user to change passwords in two ways. For normal (non–administrative) password changes, the user must supply the old password, either on the first call to the **chpass** subroutine or in response to the first message from **chpass**. If the user is root, real user ID of 0, local administrative password changes are handled by supplying a null pointer for the *Response* parameter during the initial call

Users that are not administered locally are always queried for their old password.

The **chpass** subroutine is always in one of three states, entering the old password, entering the new password, or entering the new password again. If any of these states need do not need to be complied with, the **chpass** subroutine returns a null challenge.

Parameters

UserName	Specifies the user's name whose password is to be changed.
Response	Specifies a character string containing the user's response to the last prompt.
Reenter	Points to a Boolean value used to signal whether **chpass** subroutine has completed processing. If the *Reenter* parameter is a nonzero value, the **chpass** subroutine expects the user to satisfy the prompt message provided by the *Message* parameter. If the *Reenter* parameter is 0, the **chpass** subroutine has completed processing.
Message	Points to a pointer that the **chpass** subroutine allocates memory for and fills in. This replacement string is then suitable for printing and issues challenge messages (if the *Reenter* parameter is a nonzero value). The string can also issue informational messages such as why the user failed to change the password (if the *Reenter* parameter is 0). The calling application is responsible for freeing this memory.

Return Values

Upon successful completion, the **chpass** subroutine returns a value of 0. If the **chpass** subroutine is unsuccessful, it returns the following values:

–1	Indicates the call failed in the thread safe library **libs_r.a**. **ERRNO** will indicate the failure code.
1	Indicates that the password change was unsuccessful and the user should attempt again. This return value occurs if a password restriction is not met, such as if the password is not long enough.
2	Indicates that the password change was unsuccessful and the user should not attempt again. This return value occurs if the user enters an incorrect old password or if the network is down (the password change cannot occur).

Error Codes

The **chpass** subroutine is unsuccessful if one of the following values is true:

ENOENT	Indicates that the user cannot be found.
ESAD	Indicates that the user did not meet the criteria to change the password.
EPERM	Indicates that the user did not have permission to change the password.
EINVAL	Indicates that the parameters are not valid.
ENOMEM	Indicates that memory allocation (malloc) failed.

Implementation Specifics

This subroutine is part of Base Operating System (BOS) Runtime.

Related Information

The **authenticate** subroutine.

chroot Subroutine

Purpose

Changes the effective root directory.

Library

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

Syntax

```
#include <unistd.h>


int chroot (const char *Path)
char *Path;
```

Description

The **chroot** subroutine causes the directory named by the *Path* parameter to become the effective root directory. If the *Path* parameter refers to a symbolic link, the **chroot** subroutine sets the effective root directory to the directory pointed to by the symbolic link. If Network File System (NFS) is installed on your system, this path can cross into another node.

The effective root directory is the starting point when searching for a file's path name that begins with / (slash). The current directory is not affected by the **chroot** subroutine.

The calling process must have root user authority in order to change the effective root directory. The calling process must also have search access to the new effective root directory.

The .. (double period) entry in the effective root directory is interpreted to mean the effective root directory itself. Thus, this directory cannot be used to access files outside the subtree rooted at the effective root directory.

Parameters

Path	Pointer to the new effective root directory.

Return Values

Upon successful completion, a value of 0 is returned. Otherwise, a value of –1 is returned and the **errno** global variable is set to indicate the error.

Error Codes

The **chroot** subroutine fails and the effective root directory remains unchanged if one or more of the following are true:

ENOENT	The named directory does not exist.
EACCES	The named directory denies search access.
EPERM	The process does not have root user authority.

The **chroot** subroutine can be unsuccessful for other reasons. See Appendix A. Base Operating System Error Codes for Services that Require Path–Name Resolution, on page A-1 for a list of additional errors.

If NFS is installed on the system, the **chroot** subroutine can also fail if the following is true:

ETIMEDOUT	The connection timed out.

Implementation Specifics

This subroutine is part of Base Operating System (BOS) Runtime.

Related Information

The **chdir** subroutine.

The **chroot** command.

Base Operating System Error Codes for Services that Require Path–Name Resolution.

Appendix A. Base Operating System Error Codes for Services that Require Path–Name Resolution, on page A-1.

Files, Directories, and File Systems for Programmers in *AIX General Programming Concepts : Writing and Debugging Programs*.

chssys Subroutine

Purpose

Modifies the subsystem objects associated with the *SubsystemName* parameter.

Library

System Resource Controller Library (**libsrc.a**)

Syntax

```
#include <sys/srcobj.h>
#include <spc.h>

int chssys(SubsystemName, SRCSubsystem)
char *SubsystemName;
struct SRCsubsys *SRCSubsystem;
```

Description

The **chssys** subroutine modifies the subsystem objects associated with the specified subsystem with the values in the **SRCsubsys** structure. This action modifies the objects associated with subsystem in the following object classes:

- Subsystem Environment
- Subserver Type
- Notify

The Subserver Type and Notify object classes are updated only if the subsystem name has been changed.

The **SRCsubsys** structure is defined in the **/usr/include/sys/srcobj.h** file.

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

Parameters

SRCSubsystem	Points to the **SRCsubsys** structure.
SubsystemName	Specifies the name of the subsystem.

Return Values

Upon successful completion, the **chssys** subroutine returns a value of 0. Otherwise, it returns a value of –1 and the **odmerrno** variable is set to indicate the error, or a System Resource Controller (SRC) error code is returned.

Error Codes

The **chssys** subroutine is unsuccessful if one or more of the following are true:

SRC_NONAME	No subsystem name is specified.
SRC_NOPATH	No subsystem path is specified.
SRC_BADNSIG	Invalid stop normal signal.
SRC_BADFSIG	Invalid stop force signal.
SRC_NOCONTACT	Contact not signal, sockets, or message queues.
SRC_SSME	Subsystem name does not exist.
SRC_SUBEXIST	New subsystem name is already on file.

SRC_SYNEXIST	New subsystem synonym name is already on file.
SRC_NOREC	The specified **SRCsubsys** record does not exist.
SRC_SUBSYS2BIG	Subsystem name is too long.
SRC_SYN2BIG	Synonym name is too long.
SRC_CMDARG2BIG	Command arguments are too long.
SRC_PATH2BIG	Subsystem path is too long.
SRC_STDIN2BIG	stdin path is too long.
SRC_STDOUT2BIG	stdout path is too long.
SRC_STDERR2BIG	stderr path is too long.
SRC_GRPNAM2BIG	Group name is too long.

Security

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

```
SET_PROC_AUDIT kernel privilege
```

Files Accessed:

Mode	File
644	**/etc/objrepos/SRCsubsys**
644	**/etc/objrepos/SRCsubsvr**
644	**/etc/objrepos/SRCnotify**

Auditing Events:

Event	**Information**
SRC_Chssys	

Implementation Specifics

This subroutine is part of Base Operating System (BOS) Runtime.

Files

/etc/objrepos/SRCsubsys	SRC Subsystem Configuration object class.
/etc/objrepos/SRCsubsvr	SRC Subserver Configuration object class.
/etc/objrepos/SRCnotify	SRC Notify Method object class.
/dev/SRC	Specifies the **AF_UNIX** socket file.
/dev/.SRC–unix	Specifies the location for temporary socket files.

Related Information

The **addssys** subroutine, **delssys** subroutine.

The **chssys** command, **mkssys** command, **rmssys** command.

System Resource Controller Overview in *AIX 4.3 System Management Guide: Operating System and Devices*.

Defining Your Subsystem to the SRC, List of SRC Subroutines, System Resource Controller (SRC) Overview for Programmers in *AIX General Programming Concepts : Writing and Debugging Programs*.

ckuseracct Subroutine

Purpose

Checks the validity of a user account.

Library

Security Library (**libc.a**)

Syntax

```
#include <login.h>

int ckuseracct (Name, Mode, TTY)
char *Name;
int Mode;
char *TTY;
```

Description

Note: This subroutine is obsolete and is provided only for backwards compatibility. Use the **loginrestrictions** subroutine, which performs a superset of the functions of the **ckuseracct** subroutine, instead.

The **ckuseracct** subroutine checks the validity of the user account specified by the *Name* parameter. The *Mode* parameter gives the mode of the account usage, and the *TTY* parameter defines the terminal being used for the access. The **ckuseracct** subroutine checks for the following conditions:

- Account existence
- Account expiration

The *Mode* parameter specifies other mode–specific checks.

Parameters

Name — Specifies the login name of the user whose account is to be validated.

Mode — Specifies the manner of usage. Valid values as defined in the **login.h** file are listed below. The *Mode* parameter must be one of these or 0:

S_LOGIN	Verifies that local logins are permitted for this account.
S_SU	Verifies that the **su** command is permitted and that the current process has a group ID that can invoke the **su** command to switch to the account.
S_DAEMON	Verifies the account can be used to invoke daemon or batch programs using the **src** or **cron** subsystems.
S_RLOGIN	Verifies the account can be used for remote logins using the **rlogind** or **telnetd** programs.

TTY — Specifies the terminal of the originating activity. If this parameter is a null pointer or a null string, no TTY origin checking is done.

Security

Files Accessed:

Mode	File
r	**/etc/passwd**
r	**/etc/security/user**

Return Values

If the account is valid for the specified usage, the **ckuseracct** subroutine returns a value of 0. Otherwise, a value of –1 is returned and the **errno** global variable is set to the appropriate error code.

Error Codes

The **ckuseracct** subroutine fails if one or more of the following are true:

ENOENT	The user specified in the *Name* parameter does not have an account.
ESTALE	The user's account is expired.
EACCES	The specified terminal does not have access to the specified account.
EACCES	The *Mode* parameter is **S_SU**, and the current process is not permitted to use the **su** command to access the specified user.
EACCES	Access to the account is not permitted in the specified *Mode*.
EINVAL	The *Mode* parameter is not one of **S_LOGIN**, **S_SU**, **S_DAEMON**, **S_RLOGIN**.

Implementation Specifics

This subroutine is part of Base Operating System (BOS) Runtime.

Related Information

The **ckuserID** subroutine, **getpcred** subroutine, **getpenv** subroutine, **setpcred** subroutine, **setpenv** subroutine.

The **login** command, **rlogin** command, **su** command, **telnet** command.

The **cron** daemon.

List of Security and Auditing Subroutines and Subroutines Overview in *AIX General Programming Concepts : Writing and Debugging Programs*.

ckuserID Subroutine

Purpose

Authenticates the user.

Note: This subroutine is obsolete and is provided for backwards compatibility. Use the **authenticate** subroutine, instead.

Library

Security Library (**libc.a**)

Syntax

```
#include <login.h>
int ckuserID (User, Mode)
int Mode;
char *User;
```

Description

The **ckuserID** subroutine authenticates the account specified by the *User* parameter. The mode of the authentication is given by the *Mode* parameter. The **login** and **su** commands continue to use the **ckuserID** subroutine to process the **/etc/security/user auth1** and **auth2** authentication methods.

The **ckuserID** subroutine depends on the **authenticate** subroutine to process the **SYSTEM** attribute in the **/etc/security/user** file. If authentication is successful, the **passwdexpired** subroutine is called.

Errors caused by grammar or load modules during a call to the **authenticate** subroutine are displayed to the user if the user was authenticated. These errors are audited with the **USER_Login** audit event if the user failed authentication.

Parameters

User — Specifies the name of the user to be authenticated.

Mode — Specifies the mode of authentication. This parameter is a bit mask and may contain one or more of the following values, which are defined in the **login.h** file:

S_PRIMARY — The primary authentication methods defined for the *User* parameter are checked. All primary authentication checks must be passed.

S_SECONDARY — The secondary authentication methods defined for the *User* parameter are checked. Secondary authentication checks are not required to be successful.

Primary and secondary authentication methods for each user are set in the **/etc/security/user** file by defining the **auth1** and **auth2** attributes. If no primary methods are defined for a user, the **SYSTEM** attribute is assumed. If no secondary methods are defined, there is no default.

Security

Files Accessed:

Mode	File
r	**/etc/passwd**
r	**/etc/security/passwd**
r	**/etc/security/user**
r	**/etc/security/login.cfg**

Return Values

If the account is valid for the specified usage, the **ckuserID** subroutine returns a value of 0. Otherwise, a value of –1 is returned and the **errno** global variable is set to indicate the error.

Error Codes

The **ckuserID** subroutine fails if one or more of the following are true:

ESAD	Security authentication failed for the user.
EINVAL	The *Mode* parameter is neither **S_PRIMARY** nor **S_SECONDARY** or the *Mode* parameter is both **S_PRIMARY** and **S_SECONDARY**.

Implementation Specifics

This subroutine is part of Base Operating System (BOS) Runtime.

Related Information

The **authenticate** subroutine, **ckuseracct** subroutine, **getpcred** subroutine,**getpenv** subroutine, **passwdexpired** subroutine, **setpcred** subroutine, **setpenv** subroutine.

The **login** command, **su** command.

List of Security and Auditing Subroutines and Subroutines Overview in *AIX General Programming Concepts : Writing and Debugging Programs*.

class, _class, finite, isnan, or unordered Subroutines

Purpose

Determines classifications of floating–point numbers.

Libraries

IEEE Math Library (**libm.a**)
or System V Math Library (**libmsaa.a**)

Syntax

```
#include <math.h>
#include <float.h>

int
class(x)
double x;

#include <math.h>
#include <float.h>

int
_class(x)
double x;

#include <math.h>

int finite(x)
double x;

#include <math.h>

int isnan(x)
double x;


#include <math.h>

int unordered(x, y)
double x, y;
```

Description

The **class** subroutine, **_class** subroutine, **finite** subroutine, **isnan** subroutine, and **unordered** subroutine determine the classification of their floating–point value. The **unordered** subroutine determines if a floating–point comparison involving *x* and *y* would generate the IEEE floating–point unordered condition (such as whether *x* or *y* is a NaN).

The **class** subroutine returns an integer that represents the classification of the floating–point *x* parameter. Since **class** is a reversed key word in C++. The **class** subroutine can not be invoked in a C++ program. The **_class** subroutine is an interface for C++ program using the **class** subroutine. The interface and the return value for class and **_class** subroutines are identical. The values returned by the **class** subroutine are defined in the **float.h** header file. The return values are the following:

FP_PLUS_NORM	Positive normalized, nonzero *x*
FP_MINUS_NORM	Negative normalized, nonzero *x*
FP_PLUS_DENORM	Positive denormalized, nonzero *x*
FP_MINUS_DENORM	Negative denormalized, nonzero *x*
FP_PLUS_ZERO	*x* = +0.0
FP_MINUS_ZERO	*x* = –0.0
FP_PLUS_INF	*x* = +INF
FP_MINUS_INF	*x* = –INF
FP_NANS	*x* = Signaling Not a Number (NaNS)
FP_NANQ	*x* = Quiet Not a Number (NaNQ)

Since class is a reserved keyword in C++, the **class** subroutine cannot be invoked in a C++ program. The **_class** subroutine is an interface for the C++ program using the **class** subroutine. The interface and the return values for **class** and **_class** subroutines are identical.

The **finite** subroutine returns a nonzero value if the *x* parameter is a finite number; that is, if *x* is not +–, INF, NaNQ, or NaNS.

The **isnan** subroutine returns a nonzero value if the *x* parameter is an NaNS or a NaNQ. Otherwise, it returns 0.

The **unordered** subroutine returns a nonzero value if a floating–point comparison between *x* and *y* would be unordered. Otherwise, it returns 0.

Note: Compile any routine that uses subroutines from the **libm.a** library with the **–lm** flag. To compile the **class.c** file, for example, enter:

```
cc class.c -lm
```

Parameters

x	Specifies some double–precision floating–point value.
y	Specifies some double–precision floating–point value.

Error Codes

The **finite**, **isnan**, and **unordered** subroutines neither return errors nor set bits in the floating–point exception status, even if a parameter is an NaNS.

Implementation Specifics

These subroutines are part of Base Operating System (BOS) Runtime.

Related Information

IEEE Standard for Binary Floating–Point Arithmetic (ANSI/IEEE Standards 754–1985 and 854–1987).

List of Numerical Manipulation Services and Subroutines Overview in *AIX General Programming Concepts : Writing and Debugging Programs*.

clock Subroutine

Purpose

Reports central processing unit (CPU) time used.

Library

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

Syntax

```
#include <time.h>

clock_t clock (void);
```

Description

The **clock** subroutine reports the amount of CPU time used. The reported time is the sum of the CPU time of the calling process and its terminated child processes for which it has executed **wait**, **system**, or **pclose** subroutines. To measure the amount of time used by a program, the **clock** subroutine should be called at the beginning of the program, and that return value should be subtracted from the return value of subsequent calls to the **clock** subroutine. To find the time in seconds, divide the value returned by the **clock** subroutine by the value of the macro **CLOCKS_PER_SEC**, which is defined in the **time.h** file.

Return Values

The **clock** subroutine returns the amount of CPU time used.

Implementation Specifics

This subroutine is part of Base Operating System (BOS) Runtime.

Related Information

The **getrusage, times** subroutine, **pclose** subroutine, **system** subroutine, **vtimes** subroutine, **wait, waitpid, wait3** subroutine.

Subroutines Overview in *AIX General Programming Concepts : Writing and Debugging Programs*.

close Subroutine

Purpose

Closes the file associated with a file descriptor.

Syntax

```
#include <unistd.h>



int close (
FileDescriptor)
int FileDescriptor;
```

Description

The **close** subroutine closes the file associated with the *FileDescriptor* parameter. If Network File System (NFS) is installed on your system, this file can reside on another node.

All file regions associated with the file specified by the *FileDescriptor* parameter that this process has previously locked with the **lockf** or **fcntl** subroutine are unlocked. This occurs even if the process still has the file open by another file descriptor.

If the *FileDescriptor* parameter resulted from an **open** subroutine that specified **O_DEFER**, and this was the last file descriptor, all changes made to the file since the last **fsync** subroutine are discarded.

If the *FileDescriptor* parameter is associated with a mapped file, it is unmapped. The **shmat** subroutine provides more information about mapped files.

The **close** subroutine attempts to cancel outstanding asynchronous I/O requests on this file descriptor. If the asynchronous I/O requests cannot be canceled, the application is blocked until the requests have completed.

The **close** subroutine is blocked when another thread of the same process is using the file descriptor.

When all file descriptors associated with a pipe or FIFO special file have been closed, any data remaining in the pipe or FIFO is discarded. If the link count of the file is 0 when all file descriptors associated with the file have been closed, the space occupied by the file is freed, and the file is no longer accessible.

> **Note:** If the *FileDescriptor* parameter refers to a device and the **close** subroutine actually results in a device **close**, and the device **close** routine returns an error, the error is returned to the application. However, the *FileDescriptor* parameter is considered closed and it may not be used in any subsequent calls.

All open file descriptors are closed when a process exits. In addition, file descriptors may be closed during the **exec** subroutine if the **close–on–exec** flag has been set for that file descriptor.

Parameters

FileDescriptor	Specifies a valid open file descriptor.

Return Values

Upon successful completion, a value of 0 is returned. Otherwise, a value of –1 is returned and the **errno** global variable is set to identify the error. If the **close** subroutine is interrupted by a signal that is caught, it returns a value of –1, the **errno** global variable is set to **EINTR** and the state of the *FileDescriptor* parameter is closed.

Error Codes

The **close** subroutine is unsuccessful if the following is true:

EBADF	The *FileDescriptor* parameter does not specify a valid open file descriptor.
EINTR	Specifies that the **close** subroutine was interrupted by a signal.

The **close** subroutine may also be unsuccessful if the file being closed is NFS–mounted and the server is down under the following conditions:

- The file is on a hard mount.
- The file is locked in any manner.

The **close** subroutine may also be unsuccessful if NFS is installed and the following is true:

ETIMEDOUT	The connection timed out.

Implementation Specifics

This subroutine is part of Base Operating System (BOS) Runtime.

Related Information

The **exec** subroutines, **fcntl** subroutine, **ioctl** subroutine, **lockfx** subroutine, **open**, **openx**, or **creat** subroutine, **pipe** subroutine, **socket** subroutine.

The Input and Output Handling Programmer's Overview in *AIX General Programming Concepts : Writing and Debugging Programs*.

compare_and_swap Subroutine

Purpose

Conditionally updates or returns a single word variable atomically.

Library

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

Syntax

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

boolean_t compare_and_swap (word_addr, old_val_addr, new_val)
atomic_p word_addr;
int *old_val_addr;
int new_val;
```

Description

The **compare_and_swap** subroutine performs an atomic operation which compares the contents of a single word variable with a stored old value. If the values are equal, a new value is stored in the single word variable and **TRUE** is returned; otherwise, the old value is set to the current value of the single word variable and **FALSE** is returned.

The **compare_and_swap** subroutine is useful when a word value must be updated only if it has not been changed since it was last read.

Note: The word containing the single word variable must be aligned on a full word boundary.

Note: If **compare_and_swap** is used as a locking primitive, insert an **isync** at the start of any critical sections.

Parameters

word_addr	Specifies the address of the single word variable.
old_val_addr	Specifies the address of the old value to be checked against (and conditionally updated with) the value of the single word variable.
new_val	Specifies the new value to be conditionally assigned to the single word variable.

Return Values

TRUE	Indicates that the single word variable was equal to the old value, and has been set to the new value.
FALSE	Indicates that the single word variable was not equal to the old value, and that its current value has been returned in the location where the old value was previously stored.

Implementation Specifics

The **compare_and_swap** subroutine is part of the Base Operating System (BOS) Runtime.

Related Information

The **fetch_and_add** subroutine, **fetch_and_and** subroutine, **fetch_and_or** subroutine.

compile, step, or advance Subroutine

Purpose

Compiles and matches regular–expression patterns.

Note: AIX commands use the **regcomp**, **regexec**, **regfree**, and **regerror** subroutines for the functions described in this article.

Library

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

Syntax

```
#define INIT declarations
#define GETC( ) getc_code
#define PEEKC( ) peekc_code
#define UNGETC(c) ungetc_code
#define RETURN(pointer) return_code
#define ERROR(val) error_code


#include <regexp.h>
#include <NLregexp.h>

char *compile (InString, ExpBuffer, EndBuffer, EndOfFile)
char *ExpBuffer;
char  *InString, *EndBuffer;
int EndOfFile;

int step (String, ExpBuffer)
const char *String, *ExpBuffer;

int advance (String, ExpBuffer)
const char *String, *ExpBuffer;
```

Description

The **/usr/include/regexp.h** file contains subroutines that perform regular–expression pattern matching. Programs that perform regular–expression pattern matching use this source file. Thus, only the **regexp.h** file needs to be changed to maintain regular expression compatibility between programs.

The interface to this file is complex. Programs that include this file define the following six macros before the **#include <regexp.h>** statement. These macros are used by the **compile** subroutine:

INIT This macro is used for dependent declarations and initializations. It is placed right after the declaration and opening { (left brace) of the **compile** subroutine. The definition of the **INIT** buffer must end with a ; (semicolon). **INIT** is frequently used to set a register variable to point to the beginning of the regular expression so that this register variable can be used in the declarations for the **GETC**, **PEEKC**, and **UNGETC** macros. Otherwise, you can use **INIT** to declare external variables that **GETC**, **PEEKC**, and **UNGETC** require.

GETC( ) This macro returns the value of the next character in the regular expression pattern. Successive calls to the **GETC** macro should return successive characters of the pattern.

PEEKC( ) This macro returns the next character in the regular expression. Successive calls to the **PEEKC** macro should return the same character, which should also be the next character returned by the **GETC** macro.

UNGETC(*c***)** This macro causes the parameter *c* to be returned by the next call to the **GETC** and **PEEKC** macros. No more than one character of pushback is ever needed, and this character is guaranteed to be the last character read by the **GETC** macro. The return value of the **UNGETC** macro is always ignored.

RETURN(*pointer***)** This macro is used for normal exit of the **compile** subroutine. The *pointer* parameter points to the first character immediately following the compiled regular expression. This is useful for programs that have memory allocation to manage.

ERROR(*val***)** This macro is used for abnormal exit from the **compile** subroutine. It should never contain a **return** statement. The *val* parameter is an error number. The error values and their meanings are:

Error	Meaning
11	Interval end point too large
16	Bad number
25	\ *digit* out of range
36	Illegal or missing delimiter
41	No remembered search String
42	\ (?\) imbalance
43	Too many \.(
44	More than two numbers given in \{ \}
45	} expected after \.
46	First number exceeds second in \{ \}
49	[ ] imbalance
50	Regular expression overflow
70	Invalid endpoint in range

The **compile** subroutine compiles the regular expression for later use. The *InString* parameter is never used explicitly by the **compile** subroutine, but you can use it in your macros. For example, you can use the **compile** subroutine to pass the string containing the pattern as the *InString* parameter to **compile** and use the **INIT** macro to set a pointer to the beginning of this string. The example in the **Examples** section uses this technique. If your macros do not use *InString*, then call **compile** with a value of **((char *) 0)** for this parameter.

The *ExpBuffer* parameter points to a character array where the compiled regular expression is to be placed. The *EndBuffer* parameter points to the location that immediately follows the character array where the compiled regular expression is to be placed. If the compiled expression cannot fit in (*EndBuffer–ExpBuffer*) bytes, the call **ERROR(***50***)** is made.

The *EndOfFile* parameter is the character that marks the end of the regular expression. For example, in the **ed** command, this character is usually / (slash).

The **regexp.h** file defines other subroutines that perform actual regular–expression pattern matching. One of these is the **step** subroutine.

The *String* parameter of the **step** subroutine is a pointer to a null–terminated string of characters to be checked for a match.

The *Expbuffer* parameter points to the compiled regular expression, obtained by a call to the **compile** subroutine.

The **step** subroutine returns the value 1 if the given string matches the pattern, and 0 if it does not match. If it matches, then **step** also sets two global character pointers: **loc1**, which points to the first character that matches the pattern, and **loc2**, which points to the character immediately following the last character that matches the pattern. Thus, if the regular expression matches the entire string, **loc1** points to the first character of the *String* parameter and **loc2** points to the null character at the end of the *String* parameter.

The **step** subroutine uses the global variable **circf**, which is set by the **compile** subroutine if the regular expression begins with a ^ (circumflex). If this variable is set, **step** only tries to match the regular expression to the beginning of the string. If you compile more than one regular expression before executing the first one, save the value of **circf** for each compiled expression and set **circf** to that saved value before each call to **step**.

Using the same parameters that were passed to it, the **step** subroutine calls a subroutine named **advance**. The **step** function increments through the *String* parameter and calls the **advance** subroutine until it returns a 1, indicating a match, or until the end of *String* is reached. To constrain the *String* parameter to the beginning of the string in all cases, call the **advance** subroutine directly instead of calling the **step** subroutine.

When the **advance** subroutine encounters an * (asterisk) or a \{ \} sequence in the regular expression, it advances its pointer to the string to be matched as far as possible and recursively calls itself, trying to match the rest of the string to the rest of the regular expression. As long as there is no match, the **advance** subroutine backs up along the string until it finds a match or reaches the point in the string that initially matched the * or \{ \}. You can stop this backing–up before the initial point in the string is reached. If the **locs** global character is equal to the point in the string sometime during the backing–up process, the **advance** subroutine breaks out of the loop that backs up and returns 0. This is used for global substitutions on the whole line so that expressions such as `s/y*//g` do not loop forever.

Note:	In 64–bit mode, these interfaces are not supported: they fail with a return code of 0. In order to use the 64–bit version of this functionality, applications should migrate to the **fnmatch**, **glob**, **regcomp**, and **regexec** functions which provide full internationalized regular expression functionality compatible with ISO 9945–1:1996 (IEEE POSIX 1003.1) and with the UNIX98 specification.

Parameters

InString	Specifies the string containing the pattern to be compiled. The *InString* parameter is not used explicitly by the **compile** subroutine, but it may be used in macros.
ExpBuffer	Points to a character array where the compiled regular expression is to be placed.
EndBuffer	Points to the location that immediately follows the character array where the compiled regular expression is to be placed.
EndOfFile	Specifies the character that marks the end of the regular expression.
String	Points to a null–terminated string of characters to be checked for a match.

Examples

The following is an example of the regular expression macros and calls:

```
#define INIT         register char *sp=instring;
#define GETC()         (*sp++)
#define PEEKC()         (*sp)
#define UNGETC(c)      (--sp)
#define RETURN(c)      return;
#define ERROR(c)      regerr()

#include <regexp.h>
 . . .
compile (patstr,expbuf, &expbuf[ESIZE], '\0');
 . . .
if (step (linebuf, expbuf))
   succeed( );
 . . .
```

Implementation Specifics

These subroutines are part of Base Operating System (BOS) Runtime.

Related Information

The **regcmp** or **regex** subroutine, **regcomp** subroutine, **regerror** subroutine, **regexec** subroutine, **regfree** subroutine.

confstr Subroutine

Purpose

Gets configurable variables.

Library

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

Syntax

#include <unistd.h>

size_t confstr (int *name*, *char * buf*, size_t *len* **);**

Description

The **confstr** subroutine determines the current setting of certain system parameters, limits, or options that are defined by a string value. It is mainly used by applications to find the system default value for the **PATH** environment variable. Its use and purpose are similar to those of the **sysconf** subroutine, but it returns string values rather than numeric values.

If the *Len* parameter is not 0 and the *Name* parameter has a system–defined value, the **confstr** subroutine copies that value into a *Len*–byte buffer pointed to by the *Buf* parameter. If the string returns a value longer than the value specified by the *Len* parameter, including the terminating null byte, then the **confstr** subroutine truncates the string to *Len*–1 bytes and adds a terminating null byte to the result. The application can detect that the string was truncated by comparing the value returned by the **confstr** subroutine with the value specified by the *Len* parameter.

Parameters

Name	Specifies the system variable setting to be returned. Valid values for the *Name* parameter are defined in the **unistd.h** file.
Buf	Points to the buffer into which the **confstr** subroutine copies the value of the *Name* parameter.
Len	Specifies the size of the buffer storing the value of the *Name* parameter.

Return Values

If the value specified by the *Name* parameter is system–defined, the **confstr** subroutine returns the size of the buffer needed to hold the entire value. If this return value is greater than the value specified by the *Len* parameter, the string returned as the *Buf* parameter is truncated.

If the value of the *Len* parameter is set to 0 and the *Buf* parameter is a null value, the **confstr** subroutine returns the size of the buffer needed to hold the entire system–defined value, but does not copy the string value. If the value of the *Len* parameter is set to 0 but the *Buf* parameter is not a null value, the result is unspecified.

Error Codes

The **confstr** subroutine will fail if:

EINVAL	The value of the name argument is invalid.

Example

To find out what size buffer is needed to store the string value of the *Name* parameter, enter:

```
confstr(_CS_PATH, NULL, (size_t) 0)
```

The **confstr** subroutine returns the size of the buffer.

Implementation Specifics

This subroutine is part of Base Operating System (BOS) Runtime.

Files

/usr/include/limits.h	Contains system–defined limits.
/usr/include/unistd.h	Contains system–defined environment variables.

Related Information

The **pathconf** subroutine, **sysconf** subroutine.

The **unistd.h** header file.

The XCU specification of getconf.

Subroutines Overview in *AIX General Programming Concepts : Writing and Debugging Programs*.

conv Subroutines

Purpose

Translates characters.

Library

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

Syntax

```
#include <ctype.h>

int toupper (Character)
int Character;

int tolower (Character)
int Character;

int _toupper (Character)
int Character;

int _tolower (Character)
int Character;

int toascii (Character)
int Character;

int NCesc (Pointer, CharacterPointer)
NLchar *Pointer;
char *CharacterPointer;

int NCtoupper (Xcharacter)
int Xcharacter;

int NCtolower (Xcharacter)
int Xcharacter;

int _NCtoupper (Xcharacter)
int Xcharacter;

int _NCtolower (Xcharacter)
int Xcharacter;

int NCtoNLchar (Xcharacter)
int Xcharacter;

int NCunesc (CharacterPointer, Pointer)
char *CharacterPointer;
NLchar *Pointer;

int NCflatchr (Xcharacter)
int Xcharacter;
```

Description

The **toupper** and the **tolower** subroutines have as domain an **int**, which is representable as an unsigned **char** or the value of **EOF**: –1 through 255.

If the parameter of the **toupper** subroutine represents a lowercase letter and there is a corresponding uppercase letter (as defined by **LC_CTYPE**), the result is the corresponding uppercase letter. If the parameter of the **tolower** subroutine represents an uppercase letter, and there is a corresponding lowercase letter (as defined by **LC_CTYPE**), the result is the corresponding lowercase letter. All other values in the domain are returned unchanged. If case–conversion information is not defined in the current locale, these subroutines determine character case according to the "C" locale.

The **_toupper** and **_tolower** subroutines accomplish the same thing as the **toupper** and **tolower** subroutines, but they have restricted domains. The **_toupper** routine requires a lowercase letter as its parameter; its result is the corresponding uppercase letter. The **_tolower** routine requires an uppercase letter as its parameter; its result is the corresponding lowercase letter. Values outside the domain cause undefined results.

The **NC***xxxxxx* subroutines translate all characters, including extended characters, as code points. The other subroutines translate traditional ASCII characters only. The **NC***xxxxxx* subroutines are obsolete and should not be used if portability and future compatibility are a concern.

The value of the *Xcharacter* parameter is in the domain of any legal **NLchar** data type. It can also have a special value of –1, which represents the end of file **(EOF)**.

If the parameter of the **NCtoupper** subroutine represents a lowercase letter according to the current collating sequence configuration, the result is the corresponding uppercase letter. If the parameter of the **NCtolower** subroutine represents an uppercase letter according to the current collating sequence configuration, the result is the corresponding lowercase letter. All other values in the domain are returned unchanged.

The **_NCtoupper** and **_NCtolower** routines are macros that perform the same function as the **NCtoupper** and **NCtolower** subroutines, but have restricted domains and are faster. The **_NCtoupper** macro requires a lowercase letter as its parameter; its result is the corresponding uppercase letter. The **_NCtolower** macro requires an uppercase letter as its parameter; its result is the corresponding lowercase letter. Values outside the domain cause undefined results.

The **NCtoNLchar** subroutine yields the value of its parameter with all bits turned off that are not part of an **NLchar** data type.

The **NCesc** subroutine converts the **NLchar** value of the *Pointer* parameter into one or more ASCII bytes stored in the character array pointed to by the *CharacterPointer* parameter. If the **NLchar** data type represents an extended character, it is converted into a printable ASCII escape sequence that uniquely identifies the extended character. **NCesc** returns the number of bytes it wrote. The display symbol table lists the escape sequence for each character.

The opposite conversion is performed by the **NCunesc** macro, which translates an ordinary ASCII byte or escape sequence starting at *CharacterPointer* into a single **NLchar** at *Pointer*. **NCunesc** returns the number of bytes it read.

The **NCflatchr** subroutine converts its parameter value into the single ASCII byte that most closely resembles the parameter character in appearance. If no ASCII equivalent exists, it converts the parameter value to a ? (question mark).

Note: The **setlocale** subroutine may affect the conversion of the decimal point symbol and the thousands separator.

Parameters

Character	Specifies the character to be converted.
Xcharacter	Specifies an **NLchar** value to be converted.
CharacterPointer	Specifies a pointer to a single–byte character array.
Pointer	Specifies a pointer to an escape sequence.

Implementation Specifics

These subroutines are part of Base Operating System (BOS) Runtime.

Related Information

The Japanese **conv** subroutines, **ctype** subroutines, **getc, fgetc, getchar,** or **getw** subroutine, **getwc**, **fgetwc**, or **getwchar** subroutine, **setlocale** subroutine.

List of Character Manipulation Services, National Language Support Overview for Programming, Subroutines Overview in *AIX General Programming Concepts : Writing and Debugging Programs*.

copysign, nextafter, scalb, logb, or ilogb Subroutine

Purpose

Computes certain binary floating–point arithmetic functions.

Libraries

IEEE Math Library (**libm.a**)
or System V Math Library (**libmsaa.a**)

Syntax

```
#include <math.h>
#include <float.h>

double copysign (x, y)
double x, y;

double nextafter (x, y)
double x, y;

double scalb(x, y)
double x, y;

double logb(x)
double x;

int ilogb (x)
double x;
```

Description

These subroutines compute certain functions recommended in the *IEEE Standard for Binary Floating–Point Arithmetic*. The other such recommended function is provided in the **class** subroutine.

The **copysign** subroutine returns the *x* parameter with the same sign as the *y* parameter.

The **nextafter** subroutine returns the next representable neighbor of the *x* parameter in the direction of the *y* parameter. If *x* equals *y*, the result is the *x* parameter.

The **scalb** subroutine returns the value of the *x* parameter times 2 to the power of the y parameter.

The **logb** subroutine returns a floating–point double that is equal to the unbiased exponent of the *x* parameter. Special cases are:

```
logb (NaN) = NaNQ
logb (infinity) = +INF
logb (0) = –INF
```

Note: When the *x* parameter is finite and not zero, then the **logb** (*x)* subroutine satisfies the following equation:

```
1 < = scalb (|x|, –(int) logb (x)) < 2
```

The **ilogb** subroutine returns an integer that is equal to the unbiased exponent of the *x* parameter. Special cases are:

```
ilogb (NaN) = LONG_MIN
ilogb (INF) = LONG_MAX
ilogb (0) = LONG_MIN
```

Compile any routine that uses subroutines from the **libm.a** library with the **–lm** flag. For example: to compile the **copysign.c** file, enter:

```
cc copysign.c –lm
```

Parameters

x	Specifies a double–precision floating–point value.
y	Specifies a double–precision floating–point value.

Return Values

The **nextafter** subroutine sets the overflow bit in the floating–point exception status when the *x* parameter is finite but the **nextafter (***x***,** *y***)** subroutine is infinite. Similarly, when the **nextafter** subroutine is denormalized, the underflow exception status flag is set.

The **logb(**0**)** subroutine returns an **–INF** value and sets the division–by–zero exception status flag.

The **ilogb(**0**)** subroutine returns a **LONG_MIN** value and sets the division–by–zero exception status flag.

Error Codes

If the correct value would overflow, the **scalb** subroutine returns +/–INF (depending on a negative or positive value of the x parameter) and sets **errno** to **ERANGE**.

If the correct value would underflow, the **scalb** subroutine returns a value of 0 and sets **errno** to **ERANGE**.

The **logb** function returns **–HUGE_VAL** when the x parameter is set to a value of 0 and sets **errno** to **EDOM**.

For the **nextafter** subroutine, if the *x* parameter is finite and the correct function value would overflow, **HUGE_VAL** is returned and **errno** is set to **ERANGE**.

crypt, encrypt, or setkey Subroutine

Purpose

Encrypts or decrypts data.

Library

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

Syntax

```
char *crypt (PW, Salt)
const char *PW, *Salt;

void encrypt (Block, EdFlag)
char Block[64];
int EdFlag;

void setkey (Key)
const char *Key;
```

Description

The **crypt** and **encrypt** subroutines encrypt or decrypt data. The **crypt** subroutine performs a one–way encryption of a fixed data array with the supplied *PW* parameter. The subroutine uses the *Salt* parameter to vary the encryption algorithm.

The **encrypt** subroutine encrypts or decrypts the data supplied in the *Block* parameter using the key supplied by an earlier call to the **setkey** subroutine. The data in the *Block* parameter on input must be an array of 64 characters. Each character must be an char 0 or char 1.

If you need to statically bind functions from **libc.a** for **crypt** do the following:

1. Create a file and add the following:

   ```
   #!
   ___setkey
   ___encrypt
   ___crypt
   ```

2. Perform the linking.
3. Add the following to the make file:

   ```
   –bI:YourFileName
   ```

   where *YourFileName* is the name of the file you created in step 1. It should look like the following:

   ```
   LDFLAGS=bnoautoimp –bI:/lib/syscalls.exp –bI:YourFileName –lc
   ```

Parameters

Block	Identifies a 64–character array containing the values (**char**) 0 and (**char**) 1. Upon return, this buffer contains the encrypted or decrypted data.
EdFlag	Determines whether the subroutine encrypts or decrypts the data. If this parameter is 0, the data is encrypted. If this is a nonzero value, the data is decrypted. If the **/usr/lib/libdes.a** file does not exist and the *EdFlag* parameter is set to nonzero, the **encrypt** subroutine returns the **ENOSYS** error code.
Key	Specifies an 64–element array of 0's and 1's cast as a **const char** data type. The *Key* parameter is used to encrypt or decrypt data.

PW	Specifies an 8–character string used to change the encryption algorithm. The first two characters of the *PW* parameter are the same as the *Salt* parameter.
Salt	Specifies a 2–character string chosen from the following:

A–**Z**	Uppercase alpha characters
0–9	Numeric characters
.	Period
/	Slash

The *Salt* parameter is used to vary the hashing algorithm in one of 4096 different ways.

Return Values

The **crypt** subroutine returns a pointer to the encrypted password. The static area this pointer indicates may be overwritten by subsequent calls.

Error Codes

The **encrypt** subroutine returns the following:

ENOSYS	The **encrypt** subroutine was called with the *EdFlag* parameter which was set to a nonzero value. Also, the /**usr**/**lib**/**libdes.a** file does not exist.

Implementation Specifics

These subroutines are part of Base Operating System (BOS) Runtime.

These subroutines are provided for compatibility with UNIX system implementations.

Related Information

The **newpass** subroutine.

The **login** command, **passwd** command, **su** command.

List of Security and Auditing Subroutines and Subroutines Overview in *AIX General Programming Concepts : Writing and Debugging Programs*.

cs Subroutine

Purpose

Compares and swaps data.

Library

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

Syntax

```
int cs (Destination, Compare, Value)

int *Destination;
int Compare;
int Value;
```

Description

Note: The **cs** subroutine is only provided to support binary compatibility with AIX Version 3 applications. When writing new applications, it is not recommended to use this subroutine; it may cause reduced performance in the future. Applications should use the **compare_and_swap** subroutine, unless they need to use unaligned memory locations.

The **cs** subroutine compares the *Compare* value with the integer pointed to by *Destination* address. If they are equal, *Value* is stored in the integer pointed to by the *Destination* address and **cs** returns 0. If the values are different, the **cs** subroutine returns 1, and the value pointed to by *Destination* address is not affected. The compare and store operations are executed atomically. Therefore, no process switches occur between them.

The **cs** subroutine can be used to implement interprocess communication facilities or to manipulate data structures shared among several processes, such as linked lists stored in shared memory.

The following example shows how a new element can be inserted in a null–terminated list that is stored in shared memory and maintained by several processes:

```
struct elem {
       struct elem  *next;
        ...
};
struct elem   *list, *new_elem;
do
       new_elem->next = list;
while (cs((int *)&list, (int)(new_elem->next),
            (int)new_elem));
```

Parameters

Destination	Specifies the address of the integer to be compared with the *Compare* value, and if need be, where *Value* will be stored.
Compare	Specifies the value to be compared with the integer pointed by *Destination* parameter address.
Value	Specifies the value stored in the integer pointed to by the *Destination* address if the *Destination* and *Compare* values are equal.

Return Codes

The **cs** subroutine returns a value of 0 if the two values compared are equal. If the values are not equal, the **cs** subroutine returns a value of 1.

Error Codes

If the integer pointed by the *Destination* parameter references memory that does not belong to the process address space, the **SIGSEGV** signal is sent to the process.

Implementation Specifics

This subroutine is part of Base Operating System (BOS) Runtime.

Related Information

The **shmat** subroutine, **shmctl** subroutine, **shmdt** subroutine, **shmget** subroutine, **sigaction**, **signal**, or **sigvec**.

Program Address Space Overview in *AIX General Programming Concepts : Writing and Debugging Programs*.

Subroutines Overview in *AIX General Programming Concepts : Writing and Debugging Programs*.

csid Subroutine

Purpose

Returns the character set ID (charsetID) of a multibyte character.

Library

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

Syntax

```
#include <stdlib.h>

int csid (String)
const char *String;
```

Description

The **csid** subroutine returns the charsetID of the multibyte character pointed to by the *String* parameter. No validation of the character is performed. The parameter must point to a value in the character range of the current code set defined in the current locale.

Parameters

String	Specifies the character to be tested.

Return Values

Successful completion returns an integer value representing the charsetID of the character. This integer can be a number from 0 through *n*, where *n* is the maximum character set defined in the `CHARSETID` field of the **charmap**. See "Understanding the Character Set Description (charmap) Source File" in *AIX 4.3 System Management Guide: Operating System and Devices* for more information.

Implementation Specifics

This subroutine is part of Base Operating System (BOS) Runtime.

Related Information

The **mbstowcs** subroutine, **wcsid** subroutine.

National Language Support Overview for Programming and Understanding the Character Set Description (charmap) Source File in *AIX 4.3 System Management Guide: Operating System and Devices*.

Subroutines Overview in *AIX General Programming Concepts : Writing and Debugging Programs*.

ctermid Subroutine

Purpose

Generates the path name of the controlling terminal.

Library

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

Syntax

```
#include <stdio.h>
char *ctermid (String)
char *String;
```

Description

The **ctermid** subroutine generates the path name of the controlling terminal for the current process and stores it in a string.

Note: File access permissions depend on user access. Access to a file whose path name the **ctermid** subroutine has returned is not guaranteed.

The difference between the **ctermid** and **ttyname** subroutines is that the **ttyname** subroutine must be handed a file descriptor and returns the actual name of the terminal associated with that file descriptor. The **ctermid** subroutine returns a string (the /**dev**/**tty** file) that refers to the terminal if used as a file name. Thus, the **ttyname** subroutine is useful only if the process already has at least one file open to a terminal.

Parameters

String	If the *String* parameter is a null pointer, the string is stored in an internal static area and the address is returned. The next call to the **ctermid** subroutine overwrites the contents of the internal static area. If the *String* parameter is not a null pointer, it points to a character array of at least `L_ctermid` elements as defined in the **stdio.h** file. The path name is placed in this array and the value of the *String* parameter is returned.

Implementation Specifics

This subroutine is part of Base Operating System (BOS) Runtime.

Related Information

The **isatty** or **ttyname** subroutine.

Input and Output Handling Programmer's Overview in *AIX General Programming Concepts : Writing and Debugging Programs*.

ctime, localtime, gmtime, mktime, difftime, asctime, or tzset Subroutine

Purpose

Converts the formats of date and time representations.

Library

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

Syntax

```
#include <time.h>

char *ctime (Clock)
const time_t *Clock;

struct tm *localtime (Clock)
const time_t *Clock;

struct tm *gmtime (Clock)
const time_t *Clock;

time_t mktime(Timeptr)
struct tm *Timeptr;

double difftime(Time1, Time0)
time_t Time0, Time1;

char *asctime (Tm)
const struct tm *Tm;

void tzset ( )
extern long int timezone;
extern int daylight;
extern char *tzname[];
```

Description

Attention: Do not use the **tzset** subroutine when linking with both **libc.a** and **libbsd.a**. The **tzset** subroutine sets the global external variable called **timezone**, which conflicts with the **timezone** subroutine in **libbsd.a**. This name collision may cause unpredictable results.

Attention: Do not use the **ctime**, **localtime**, **gmtime**, or **asctime** subroutine in a multithreaded environment. See the multithread alternatives in the **ctime_r**, **localtime_r**, **gmtime_r**, or **asctime_r** subroutine article.

The **ctime** subroutine converts a time value pointed to by the *Clock* parameter, which represents the time in seconds since 00:00:00 Coordinated Universal Time (UTC), January 1, 1970, into a 26–character string in the following form:

```
Sun Sept 16 01:03:52 1973\n\0
```

The width of each field is always the same as shown here.

The **ctime** subroutine adjusts for the time zone and daylight saving time, if it is in effect.

The **localtime** subroutine converts the long integer pointed to by the *Clock* parameter, which contains the time in seconds since 00:00:00 UTC, 1 January 1970, into a **tm** structure. The **localtime** subroutine adjusts for the time zone and for daylight–saving time, if it is in effect. Use the time–zone information as though **localtime** called **tzset**.

The **gmtime** subroutine converts the long integer pointed to by the *Clock* parameter into a **tm** structure containing the Coordinated Universal Time (UTC), which is the time standard the operating system uses.

Note: UTC is the international time standard intended to replace GMT.

The **tm** structure is defined in the **time.h** file, and it contains the following members:

```
int tm_sec;      /* Seconds (0 - 59) */
int tm_min;      /* Minutes (0 - 59) */
int tm_hour;     /* Hours (0 - 23) */
int tm_mday;     /* Day of month (1 - 31) */
int tm_mon;      /* Month of year (0 - 11) */
int tm_year;     /* Year - 1900 */
int tm_wday;     /* Day of week (Sunday = 0) */
int tm_yday;     /* Day of year (0 - 365) */
int tm_isdst;    /* Nonzero = Daylight saving time */
```

The **mktime** subroutine is the reverse function of the **localtime** subroutine. The **mktime** subroutine converts the **tm** structure into the time in seconds since 00:00:00 UTC, 1 January 1970. The `tm_wday` and `tm_yday` fields are ignored, and the other components of the **tm** structure are not restricted to the ranges specified in the **/usr/include/time.h** file. The value of the `tm_isdst` field determines the following actions of the **mktime** subroutine:

0	Initially presumes that Daylight Savings Time (DST) is not in effect.
>0	Initially presumes that DST is in effect.
–1	Actively determines whether DST is in effect from the specified time and the local time zone. Local time zone information is set by the **tzset** subroutine.

Upon successful completion, the **mktime** subroutine sets the values of the `tm_wday` and `tm_yday` fields appropriately. Other fields are set to represent the specified time since January 1, 1970. However, the values are forced to the ranges specified in the **/usr/include/time.h** file. The final value of the `tm_mday` field is not set until the values of the `tm_mon` and `tm_year` fields are determined.

The **difftime** subroutine computes the difference between two calendar times: the *Time1* and *–Time0* parameters.

The **asctime** subroutine converts a **tm** structure to a 26–character string of the same format as **ctime**.

If the **TZ** environment variable is defined, then its value overrides the default time zone, which is the U.S. Eastern time zone. The **environment** facility contains the format of the time zone information specified by **TZ**. **TZ** is usually set when the system is started with the value that is defined in either the **/etc/environment** or **/etc/profile** files. However, it can also be set by the user as a regular environment variable for performing alternate time zone conversions.

The **tzset** subroutine sets the **timezone**, **daylight**, and **tzname** external variables to reflect the setting of **TZ**. The **tzset** subroutine is called by **ctime** and **localtime**, and it can also be called explicitly by an application program.

The **timezone** external variable contains the difference, in seconds, between UTC and local standard time. For example, the value of **timezone** is 5 * 60 * 60 for U.S. Eastern Standard Time.

The **daylight** external variable is nonzero when a daylight–saving time conversion should be applied. By default, this conversion follows the standard U.S. conventions; other conventions can be specified. The default conversion algorithm adjusts for the peculiarities of U.S. daylight saving time in 1974 and 1975.

The **tzname** external variable contains the name of the standard time zone (**tzname[0]**) and of the time zone when Daylight Savings Time is in effect (**tzname[1]**). For example:

```
char *tzname[2] = {"EST", "EDT"};
```

The **time.h** file contains declarations of all these subroutines and externals and the **tm** structure.

Parameters

Clock	Specifies the pointer to the time value in seconds.
Timeptr	Specifies the pointer to a **tm** structure.
Time1	Specifies the pointer to a **time_t** structure.
Time0	Specifies the pointer to a **time_t** structure.
Tm	Specifies the pointer to a **tm** structure.

Return Values

Attention: The return values point to static data that is overwritten by each call.

The **tzset** subroutine returns no value.

The **mktime** subroutine returns the specified time in seconds encoded as a value of type **time_t**. If the time cannot be represented, the function returns the value (**time_t**)–1.

The **localtime** and **gmtime** subroutines return a pointer to the **struct tm**.

The **ctime** and **asctime** subroutines return a pointer to a 26–character string.

The **difftime** subroutine returns the difference expressed in seconds as a value of type **double**.

Implementation Specifics

These subroutines are part of Base Operating System (BOS) Runtime.

Related Information

The **getenv** subroutine, **gettimer** subroutine, **strftime** subroutine.

List of Time Data Manipulation Services in *AIX 4.3 System Management Guide: Operating System and Devices*.

National Language Support Overview for Programming in *AIX General Programming Concepts : Writing and Debugging Programs*.

Subroutines Overview in *AIX General Programming Concepts : Writing and Debugging Programs*.

ctime_r, localtime_r, gmtime_r, or asctime_r Subroutine

Purpose

Converts the formats of date and time representations.

Library

Thread–Safe C Library (**libc_r.a**)

Syntax

```
#include <time.h>

char *ctime_r(Timer, BufferPointer)
const time_t *Timer;
char *BufferPointer;

struct tm *localtime_r(Timer, CurrentTime)
const time_t *Timer;
struct tm *CurrentTime;

struct tm *gmtime_r(Timer, XTime)
const time_t *Timer;
struct tm *XTime;

char *asctime_r(TimePointer, BufferPointer)
const struct tm *TimePointer;
char *BufferPointer;
```

Description

The **ctime_r** subroutine converts a time value pointed to by the *Timer* parameter, which represents the time in seconds since 00:00:00 Coordinated Universal Time (UTC), January 1, 1970, into the character array pointed to by the *BufferPointer* parameter. The character array should have a length of at least 26 characters so the converted time value fits without truncation. The converted time value string takes the form of the following example:

```
Sun Sep 16 01:03:52 1973\n\0
```

The width of each field is always the same as shown here.

The **ctime_r** subroutine adjusts for the time zone and daylight saving time, if it is in effect.

The **localtime_r** subroutine converts the **time_t** structure pointed to by the *Timer* parameter, which contains the time in seconds since 00:00:00 UTC, January 1, 1970, into the **tm** structure pointed to by the *CurrentTime* parameter. The **localtime_r** subroutine adjusts for the time zone and for daylight saving time, if it is in effect.

The **gmtime_r** subroutine converts the **time_t** structure pointed to by the *Timer* parameter into the **tm** structure pointed to by the *XTime* parameter.

The **tm** structure is defined in the **time.h** header file. The **time.h** file contains declarations of these subroutines, externals, and the **tm** structure.

The **asctime_r** subroutine converts the **tm** structure pointed to by the *TimePointer* parameter into a 26–character string in the same format as the **ctime_r** subroutine. The results are placed into the character array, *BufferPointer*. The *BufferPointer* parameter points to the resulting character array, which takes the form of the following example:

```
char *tzname[2] = {"EST", "EDT"};
```

Parameters

Timer	Points to a **time_t** structure, which contains the number of seconds since 00:00:00 UTC, January 1, 1970.
BufferPointer	Points to a character array at least 26 characters long.
CurrentTime	Points to a **tm** structure. The result of the **localtime_r** subroutine is placed here.
XTime	Points to a **tm** structure used for the results of the **gmtime_r** subroutine.
TimePointer	Points to a **tm** structure used as input to the **asctime_r** subroutine.

Return Values

Attention: The return values point to static data that is overwritten by each call.

The **localtime_r** and **gmtime_r** subroutines return a pointer to the **tm** structure.

The **ctime_r** and **asctime_r** subroutines return a pointer to a 26–character string.

Implementation Specifics

These subroutines are part of Base Operating System (BOS) Runtime.

Programs using this subroutine must link to the **libpthreads.a** library.

Files

/usr/include/time.h

Defines time macros, data types, and structures.

Related Information

The **ctime**, **localtime**, **gmtime**, **mktime**, **difftime**, **asctime**, **tzset**, or **timezone** subroutine.

Subroutines Overview, List of Time Data Manipulation Services, List of Multithread Subroutines, and National Language Support Overview for Programming in *AIX General Programming Concepts : Writing and Debugging Programs*.

ctype Subroutines

Purpose

Classifies characters.

Library

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

Syntax

```
#include <ctype.h>

int isalpha (Character)
int Character;

int isupper (Character)
int Character;

int islower (Character)
int Character;

int isdigit (Character)
int Character;

int isxdigit (Character)
int Character;

int isalnum (Character)
int Character;

int isspace (Character)
int Character;

int ispunct (Character)
int Character;

int isprint (Character)
int Character;

int isgraph (Character)
int Character;

int iscntrl (Character)
int Character;

int isascii (Character)
int Character;
```

Description

The **ctype** subroutines classify character–coded integer values specified in a table. Each of these subroutines returns a nonzero value for True and 0 for False.

Note: The **ctype** subroutines should only be used on character data that can be represented by a single byte value (0 through 255). Attempting to use the **ctype** subroutines on multi–byte locale data may give inconsistent results. Wide character classification routines (such as **iswprint**, **iswlower**, etc.) should be used with dealing with multi–byte character data.

Locale Dependent Character Tests

The following subroutines return nonzero (True) based upon the character class definitions for the current locale.

isalnum Returns nonzero for any character for which the **isalpha** or **isdigit** subroutine would return nonzero. The **isalnum** subroutine tests whether the character is of the **alpha** or **digit** class.

isalpha Returns nonzero for any character for which the **isupper** or **islower** subroutines would return nonzero. The **isalpha** subroutine also returns nonzero for any character defined as an alphabetic character in the current locale, or for a character for which *none* of the **iscntrl**, **isdigit**, **ispunct**, or **isspace** subroutines would return nonzero. The **isalpha** subroutine tests whether the character is of the **alpha** class.

isupper Returns nonzero for any uppercase letter [A through Z]. The **isupper** subroutine also returns nonzero for any character defined to be uppercase in the current locale. The **isupper** subroutine tests whether the character is of the **upper** class.

islower Returns nonzero for any lowercase letter [a through z]. The **islower** subroutine also returns nonzero for any character defined to be lowercase in the current locale. The **islower** subroutine tests whether the character is of the **lower** class.

isspace Returns nonzero for any white–space character (space, form feed, new line, carriage return, horizontal tab or vertical tab). The **isspace** subroutine tests whether the character is of the **space** class.

ispunct Returns nonzero for any character for which the **isprint** subroutine returns nonzero, except the space character and any character for which the **isalnum** subroutine would return nonzero. The **ispunct** subroutine also returns nonzero for any locale–defined character specified as a punctuation character. The **ispunct** subroutine tests whether the character is of the **punct** class.

isprint Returns nonzero for any printing character. Returns nonzero for any locale–defined character that is designated a printing character. This routine tests whether the character is of the **print** class.

isgraph Returns nonzero for any character for which the **isprint** character returns nonzero, except the space character. The **isgraph** subroutine tests whether the character is of the **graph** class.

iscntrl Returns nonzero for any character for which the **isprint** subroutine returns a value of False (0) and any character that is designated a control character in the current locale. For the C locale, control characters are the ASCII delete character (0177 or 0x7F), or an ordinary control character (less than 040 or 0x20). The **iscntrl** subroutine tests whether the character is of the **cntrl** class.

Locale Independent Character Tests

The following subroutines return nonzero for the same characters, regardless of the locale:

isdigit	*Character* is a digit in the range [0 through 9].
isxdigit	*Character* is a hexadecimal digit in the range [0 through 9], [A through F], or [a through f].
isascii	*Character* is an ASCII character whose value is in the range 0 through 0177 (0 through 0x7F), inclusive.

Parameter

Character	Indicates the character to be tested (integer value).

Return Codes

The **ctype** subroutines return nonzero (True) if the character specified by the *Character* parameter is a member of the selected character class; otherwise, a 0 (False) is returned.

Implementation Specifics

These subroutines are part of Base Operating System (BOS) Runtime.

Related Information

The **setlocale** subroutine.

List of Character Manipulation Services and Subroutines Overview in *AIX General Programming Concepts : Writing and Debugging Programs*.

cuserid Subroutine

Purpose

Gets the alphanumeric user name associated with the current process.

Library

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

Use the **libc_r.a** library to access the thread–safe version of this subroutine.

Syntax

```
#include <stdio.h>

char *cuserid (Name)
char *Name;
```

Description

The **cuserid** subroutine gets the alphanumeric user name associated with the current process. This subroutine generates a character string representing the name of a process's owner.

Note: The **cuserid** subroutine duplicates functionality available with the **getpwuid** and **getuid** subroutines. Present applications should use the **getpwuid** and **getuid** subroutines.

If the *Name* parameter is a null pointer, then a character string of size `L_cuserid` is dynamically allocated with **malloc**, and the character string representing the name of the process owner is stored in this area. The **cuserid** subroutine then returns the address of this area. Multithreaded application programs should use this functionality to obtain thread specific data, and then continue to use this pointer in subsequent calls to the **curserid** subroutine. In any case, the application program must deallocate any dynamically allocated space with the **free** subroutine when the data is no longer needed.

If the *Name* parameter is not a null pointer, the character string is stored into the array pointed to by the *Name* parameter. This array must contain at least the number of characters specified by the constant `L_cuserid`. This constant is defined in the **stdio.h** file.

If the user name cannot be found, the **cuserid** subroutine returns a null pointer; if the *Name* parameter is not a null pointer, a null character ('\0') is stored in *Name* [0].

Parameter

Name	Points to a character string representing a user name.

Implementation Specifics

This subroutine is part of Base Operating System (BOS) Runtime.

Related Information

The **endpwent** subroutine, **getlogin**, **getpwent**, **getpwnam**, **getpwuid**, or **putpwent** subroutine.

Input and Output Handling Programmer's Overview in *AIX General Programming Concepts : Writing and Debugging Programs*.

defssys Subroutine

Purpose

Initializes the **SRCsubsys** structure with default values.

Library

System Resource Controller Library (**libsrc.a**)

Syntax

```
#include <sys/srcobj.h>
#include <spc.h>


void defssys(SRCSubsystem)
struct SRCsubsys *SRCSubsystem;
```

Description

The **defssys** subroutine initializes the **SRCsubsys** structure of the **/usr/include/sys/srcobj.h** file with the following default values:

Field	Value
`display`	SRCYES
`multi`	SRCNO
`contact`	SRCSOCKET
`waittime`	TIMELIMIT
`priority`	20
`action`	ONCE
`standerr`	**/dev/console**
`standin`	**/dev/console**
`standout`	**/dev/console**

All other numeric fields are set to 0, and all other alphabetic fields are set to an empty string.

This function must be called to initialize the **SRCsubsys** structure before an application program uses this structure to add records to the subsystem object class.

Parameters

SRCSubsystem	Points to the **SRCsubsys** structure.

Implementation Specifics

This subroutine is part of Base Operating System (BOS) Runtime.

Related Information

The **addssys** subroutine.

Defining Your Subsystem to the SRC, List of SRC Subroutines, System Resource Controller (SRC) Overview for Programmers in *AIX General Programming Concepts : Writing and Debugging Programs*.

delssys Subroutine

Purpose

Removes the subsystem objects associated with the *SubsystemName* parameter.

Library

System Resource Controller Library (**libsrc.a**)

Syntax

```
#include <sys/srcobj.h>
#include <spc.h>

int delssys (SubsystemName)
char *SubsystemName;
```

Description

The **delssys** subroutine removes the subsystem objects associated with the specified subsystem. This removes all objects associated with that subsystem from the following object classes:

- Subsystem
- Subserver Type
- Notify

The program running with this subroutine must be running with the group **system**.

Parameter

SubsystemName	Specifies the name of the subsystem.

Return Values

Upon successful completion, the **delssys** subroutine returns a positive value. If no record is found, a value of 0 is returned. Otherwise, –1 is returned and the **odmerrno** variable is set to indicate the error. See ”Appendix B. ODM Error Codes”, on page B-1 for a description of possible **odmerrno** values.

Security

Privilege Control:

SET_PROC_AUDIT kernel privilege

Files Accessed:

Mode	**File**
644	**/etc/objrepos/SRCsubsys**
644	**/etc/objrepos/SRCsubsvr**
644	**/etc/objrepos/SRCnotify**

Auditing Events:

Event	**Information**
SRC_Delssys	Lists in an audit log the name of the subsystem being removed.

Implementation Specifics

This subroutine is part of Base Operating System (BOS) Runtime.

Files

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

Related Information

The **addssys** subroutine, **chssys** subroutine.

The **chssys** command, **mkssys** command, **rmssys** command.

List of SRC Subroutines and System Resource Controller (SRC) Overview for Programmers in *AIX General Programming Concepts : Writing and Debugging Programs*.

dirname Subroutine

Purpose

Report the parent directory name of a file path name.

Library

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

Syntax

#include <libgen.h>

char *dirname (*path*)
**char ***path*

Description

Given a pointer to a character string that contains a file system path name, the **dirname** subroutine returns a pointer to a string that is the parent directory of that file. Trailing "/" characters in the path are not counted as part of the path.

If *path* is a null pointer or points to an empty string, a pointer to a static constant "." is returned.

The **dirname** and **basename** subroutines together yield a complete path name. **dirname** (*path*) is the directory where **basename** (*path*) is found.

Parameters

path	Character string containing a file system path name.

Return Values

The **dirname** subroutine returns a pointer to a string that is the parent directory of *path.* If *path* or **path* is a null pointer or points to an empty string, a pointer to a string "." is returned. The **dirname** subroutine may modify the string pointed to by *path* and may return a pointer to static storage that may then be overwritten by sequent calls to the **dirname** subroutine.

Examples

A simple file name and the strings "." and ".." all have "." as their return value.

Input string	Output string
/usr/lib	/usr
/usr/	/
usr	.
/	/
.	.
..	.

The following code reads a path name, changes directory to the appropriate directory, and opens the file.

```
char path [MAXPATHEN],   *pathcopy;
int fd;
fgets (path, MAXPATHEN, stdin);
pathcopy = strdup (path);
chdir (dirname (pathcopy) );
fd = open (basename (path), O_RDONLY);
```

Implementation Specifics

This subroutine is part of Base Operating System (BOS) Runtime.

Related Information

The **basename** or **chdir** subroutine.

disclaim Subroutine

Purpose

Disclaims the content of a memory address range.

Syntax

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

int disclaim ( Address, Length, Flag)
char *Address;
unsigned int Length, Flag;
```

Description

The **disclaim** subroutine marks an area of memory having content that is no longer needed. The system then stops paging the memory area. This subroutine cannot be used on memory that is mapped to a file by the **shmat** subroutine.

Parameters

Address	Points to the beginning of the memory area.
Length	Specifies the length of the memory area in bytes.
Flag	Must be the value **ZERO_MEM**, which indicates that each memory location in the address range should be set to 0.

Return Values

When successful, the **disclaim** subroutine returns a value of 0.

Error Codes

If the **disclaim** subroutine is unsuccessful, it returns a value of –1 and sets the **errno** global variable to indicate the error. The **disclaim** subroutine is unsuccessful if one or more of the following are true:

EFAULT	The calling process does not have write access to the area of memory that begins at the *Address* parameter and extends for the number of bytes specified by the *Length* parameter.
EINVAL	The value of the *Flag* parameter is not valid.
EINVAL	The memory area is mapped to a file.

dlclose Subroutine

Purpose

Closes and unloads a module loaded by the **dlopen** subroutine.

Syntax

```
#include <dlfcn.h>

int dlclose(Data);
void *Data;
```

Description

The **dlclose** subroutine is used to remove access to a module loaded with the **dlopen** subroutine. In addition, access to dependent modules of the module being unloaded is removed as well.

Modules being unloaded with the **dlclose** subroutine will not be removed from the process's address space if they are still required by other modules. Nevertheless, subsequent uses of *Data* are invalid, and further uses of symbols that were exported by the module being unloaded result in undefined behavior.

Parameters

Data	A loaded module reference returned from a previous call to **dlopen**.

Return Values

Upon successful completion, `0` (zero) is returned. Otherwise, **errno** is set to **EINVAL**, and the return value is also **EINVAL**. Even if the **dlclose** subroutine succeeds, the specified module may still be part of the process's address space if the module is still needed by other modules.

Error Codes

EINVAL	The *Data* parameter does not refer to a module opened by **dlopen** that is still open. The parameter may be corrupt or the module may have been unloaded by a previous call to **dlclose**.

Implementation Specifics

This subroutine is part of Base Operating System (BOS) Runtime.

Related Information

The **dlerror** subroutine, **dlopen** subroutine, **dlsym** subroutine, **load** subroutine, **loadquery** subroutine, **unload** subroutine, **loadbind** subroutine.

The **ld** command.

The Shared Libraries and Shared Memory Overview and Subroutines Overview in *AIX General Programming Concepts : Writing and Debugging Programs*.

dlerror Subroutine

Purpose

Return a pointer to information about the last **dlopen**, **dlsym**, or **dlclose** error.

Syntax

```
#include <dlfcn.h>

char *dlerror(void);
```

Description

The **dlerror** subroutine is used to obtain information about the last error that occurred in a dynamic loading routine (that is, **dlopen** , **dlsym** , or **dlclose**). The returned value is a pointer to a null–terminated string without a final newline. Once a call is made to this function, subsequent calls without any intervening dynamic loading errors will return `NULL`.

Applications can avoid calling the **dlerror** subroutine, in many cases, by examining **errno** after a failed call to a dynamic loading routine. If **errno** is **ENOEXEC**, the **dlerror** subroutine will return additional information. In all other cases, **dlerror** will return the string corresponding to the value of **errno**.

The **dlerror** function may invoke **loadquery** to ascertain reasons for a failure. If a call is made to l**oad** or **unload** between calls to **dlopen** and **dlerror**, incorrect information may be returned.

Return Values

A pointer to a static buffer is returned; a `NULL` value is returned if there has been no error since the last call to **dlerror**. Applications should not write to this buffer; they should make a copy of the buffer if they wish to preserve the buffer's contents.

Implementation Specifics

This subroutine is part of Base Operating System (BOS) Runtime.

Related Information

The **load** subroutine, **loadbind** subroutine, **loadquery** subroutine, **unload** subroutine, **dlopen** subroutine, **dlclose** subroutine, **dlsym** subroutine.

The **ld** command.

The Shared Libraries and Shared Memory Overview and Subroutines Overview in *AIX General Programming Concepts : Writing and Debugging Programs*.

dlopen Subroutine

Purpose

Dynamically load a module into the calling process.

Syntax

```
#include <dlfcn.h>

void *dlopen (FilePath, Flags);
const char *FilePath;
int Flags;
```

Description

The **dlopen** subroutine loads the module specified by *FilePath* into the executing process's address space. Dependents of the module are automatically loaded as well. If the module is already loaded, i t is not loaded again, but a new, unique value will be returned by the **dlopen** subroutine.

The value returned by **dlopen** may be used in subsequent calls to **dlsym** and **dlclose**. If an error occurs during the operation, **dlopen** returns `NULL`.

If the main application was linked with the **–brtl** option, then the runtime linker is invoked by **dlopen**. If the module being loaded was linked with runtime linking enabled, both intra–module and inter–module references are overridden by any symbols available in the main application. If runtime linking was enabled, but the module was not built enabled, then all inter–module references will be overridden, but some intra–module references will not be overridden.

If the module being opened with **dlopen** or any of its dependents is being loaded for the first time, initialization routines for these newly–loaded routines are called (after runtime linking, if applicable) before **dlopen** returns. Initialization routines are the functions specified with the **–binitfini:** linker option when the module was built. (Refer to the **ld** command for more information about this option.)

Notes:

1. The initialization functions need not have any special names, and multiple functions per module are allowed.
2. If the module being loaded has read–other permission, the module is loaded into the global shared library segment. Modules loaded into the global shared library segment are not unloaded even if they are no longer being used. Use the **slibclean** command to remove unused modules from the global shared library segment.

Use the environment variable *LIBPATH* to specify a list of directories in which **dlopen** search es for the named module. The running application also contains a set of library search paths that were specified when the application was linked; these paths are searched after any paths found in *LIBPATH*. Also, the **setenv** subroutine

FilePath	Specifies the name of a file containing the loadable module. This parameter can be contain an absolute path, a relative path, or no path component. If *FilePath* contains a slash character, *FilePath* is used directly, and no directories are searched. If the *FilePath* parameter is `/unix`, **dlopen** returns a value that can be used to look up symbols in the current kernel image, including those symbols found in any kernel extension that was available at the time the process began execution. If the value of *FilePath* is `NULL`, a value for the main application is returned. This allows dynamically loaded objects to look up symbols in the main executable, or for an application to examine symbols available within itself.

Flags

Specifies variations of the behavior of **dlopen**. Either **RTLD_NOW** or **RTLD_LAZY** must always be specified. Other flags may be OR'ed with **RTLD_NOW** or **RTLD_LAZY**.

RTLD_NOW	Load all dependents of the module being loaded and resolve all symbols.
RTLD_LAZY	Specifies the same behavior as **RTLD_NOW**. In a future release of AIX, the behavior of the **RTLD_LAZY** may change so that loading of dependent modules is deferred of resolution of some symbols is deferred.
RTLD_GLOBAL	Allows symbols in the module being loaded to be visible when resolving symbols used by other **dlopen** calls. These symbols will also be visible when the main application is opened with **dlopen**(NULL, *mode*).
RTLD_LOCAL	Prevent symbols in the module being loaded from being used when resolving symbols used by other **dlopen** calls. Symbols in the module being loaded can only be accessed by calling **dlsym** subroutine. If neither **RTLD_GLOBAL** nor **RTLD_LOCAL** is specified, the default is **RTLD_LOCAL**. If both flags are specified, **RTLD_LOCAL** is ignored.
RTLD_MEMBER	The **dlopen** subroutine can be used to load a module that is a member of an archive. The **L_LOADMEMBER** flag is used when the **load** subroutine is called. The module name *FilePath* names the archive and archive member according to the rules outlined in the **load** subroutine.
RTLD_NOAUTODEFER	Prevents deferred imports in the module being loaded from being automatically resolved by subsequent loads. The **L_NOAUTODEFER** flag is used when the **load** subroutine is called. Ordinarily, modules built for use by the **dlopen** and **dlsym** sub routines will not contain deferred imports. However, deferred imports can be still used. A module opened with **dlopen** may provide definitions for deferred imports in the main application, for modules loaded with the **load** subroutine (if the **L_NOAUTODEFER** flag was not used), and for other modules loaded with the **dlopen** subroutine (if the **RTLD_NOAUTODEFER** flag was not used).

Return Values

Upon successful completion, **dlopen** returns a value that can be used in calls to the **dlsym** and **dlclose** subroutines. The value is not valid for use with the **loadbind** and **unload** subroutines.

If the **dlopen** call fails, `NULL` (a value of 0) is returned and the global variable **errno** is set. If **errno** contains the value `ENOEXEC`, further information is available via the **dlerror** function.

Error Codes

See the **load** subroutine for a list of possible **errno** values and their meanings.

Implementation Specifics

This subroutine is part of Base Operating System (BOS) Runtime.

Related Information

The **dlclose** subroutine, **dlerror** subroutine, **dlsym** subroutine, **load** subroutine, **loadbind** subroutine, **loadquery** subroutine, **unload** subroutine.

The **ld** command.

Subroutines Overview in *AIX General Programming Concepts : Writing and Debugging Programs*.

dlsym Subroutine

Purpose

Looks up the location of a symbol in a module that is loaded with **dlopen**.

Syntax

```
#include <dlfcn.h>

void *dlsym(Data, Symbol);
void *Data;
const char *Symbol;
```

Description

The dlsym subroutine looks up a named symbol exported from a module loaded by a previous call to the dlopen subroutine. Only exported symbols are found by dlsym. See the ld command to see how to export symbols from a module.

Data	Specifies a value returned by a previous call to **dlopen**.
Symbol	Specifies the name of a symbol exported from the referenced module. The form should be a NULL–terminated string.

Note: C++ symbol names should be passed to **dlsym** in mangled form; **dlsym** does not perform any name demangling on behalf of the calling application.

A search for the named symbol is based upon breadth–first ordering of the module and its dependants. If the module was constructed using the **–G** or **–brtl** linker option, the module's dependants will include all modules named on the **ld** command line, in the original order. The dependants of a module that was not linked with the **–G** or **–brtl** linker option will be listed in an unspecified order.

Return Values

If the named symbol is found, its address is returned. If the named symbol is not found, NULL is returned and **errno** is set to 0. If *Data* or *Symbol* are invalid, NULL is returned and **errno** is set to **EINVAL** .

If the first definition found is an export of an imported symbol, this definition will satisfy the search. The address of the imported symbol is returned. If the first definition is a deferred import, the definition is ignored and the search continues.

If the named symbol refers to a BSS symbol (uninitialized data structure), the search continues until an initialized instance of the symbol is found or the module and all of its dependants have been searched . If an initialized instance is found, its address is returned; otherwise, the address of the first uninitialized instance is returned.

Error Codes

EINVAL	If the *Data* parameter does not refer to a module opened by **dlopen** that is still loaded or if the *Symbol* parameter points to an invalid address, the **dlsym** subroutine returns NULL and **errno** is set to **EINVAL**.

Implementation Specifics

This subroutine is part of Base Operating System (BOS) Runtime.

Related Information

The **dlclose** subroutine, **dlerror** subroutine, **dlopen** subroutine, **load** subroutine, **loadbind** subroutine, **loadquery** subroutine, **unload** subroutine.

The **ld** command.

drand48, erand48, jrand48, lcong48, lrand48, mrand48, nrand48, seed48, or srand48 Subroutine

Purpose

Generate uniformly distributed pseudo–random number sequences.

Library

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

Syntax

```
#include <stdlib.h>

double drand48 (void)

double erand48 (xsubi)
unsigned short int xsubi[3];

long int jrand48 (xsubi)
unsigned short int xsubi[3];

void lcong48 (Parameter)
unsigned short int Parameter[7];

long int lrand48 (void)

long int mrand48 (void)

long int nrand48 (xsubi)
unsigned short int xsubi[3];

unsigned short int *seed48 (Seed16v)
unsigned short int Seed16v[3];

void srand48 (SeedValue)
long int SeedValue;
```

Description

Attention: Do not use the **drand48**, **erand48**, **jrand48**, **lcong48**, **lrand48**, **mrand48**, **nrand48**, **seed48**, or **srand48** subroutine in a multithreaded environment.

This family of subroutines generates pseudo–random numbers using the linear congruential algorithm and 48–bit integer arithmetic.

The **drand48** subroutine and the **erand48** subroutine return positive double–precision floating–point values uniformly distributed over the interval [0.0, 1.0).

The **lrand48** subroutine and the **nrand48** subroutine return positive long integers uniformly distributed over the interval [0,2**31).

The **mrand48** subroutine and the **jrand48** subroutine return signed long integers uniformly distributed over the interval [–2**31, 2**31).

The **srand48** subroutine, **seed48** subroutine, and **lcong48** subroutine initialize the random–number generator. Programs must call one of them before calling the **drand48**, **lrand48** or **mrand48** subroutines. (Although it is not recommended, constant default initializer values are supplied if the **drand48**, **lrand48** or **mrand48** subroutines are called without first calling an initialization subroutine.) The **erand48**, **nrand48**, and **jrand48** subroutines do not require that an initialization subroutine be called first.

The previous value pointed to by the **seed48** subroutine is stored in a 48–bit internal buffer, and a pointer to the buffer is returned by the **seed48** subroutine. This pointer can be ignored if it is not needed, or it can be used to allow a program to restart from a given point at a later time. In this case, the pointer is accessed to retrieve and store the last value pointed to by the **seed48** subroutine, and this value is then used to reinitialize, by means of the **seed48** subroutine, when the program is restarted.

All the subroutines work by generating a sequence of 48–bit integer values, *x*[*i*], according to the linear congruential formula:

```
x[n+1] = (ax[n] + c)mod m, n is > = 0
```

The parameter *m* = 248; hence 48–bit integer arithmetic is performed. Unless the **lcong48** subroutine has been called, the multiplier value *a* and the addend value *c* are:

```
a = 5DEECE66D base 16 = 273673163155 base 8
```

```
c = B base 16 = 13 base 8
```

Parameters

xsubi — Specifies an array of three shorts, which, when concatenated together, form a 48–bit integer.

SeedValue — Specifies the initialization value to begin randomization. Changing this value changes the randomization pattern.

Seed16v — Specifies another seed value; an array of three unsigned shorts that form a 48–bit seed value.

Parameter — Specifies an array of seven shorts, which specifies the initial *xsubi* value, the multiplier value *a* and the add–in value *c*.

Return Values

The value returned by the **drand48**, **erand48**, **jrand48, lrand48**, **nrand48**, and **mrand48** subroutines is computed by first generating the next 48–bit *x*[*i*] in the sequence. Then the appropriate number of bits, according to the type of data item to be returned, are copied from the high–order (most significant) bits of *x*[*i*] and transformed into the returned value.

The **drand48**, **lrand48**, and **mrand48** subroutines store the last 48–bit *x*[*i*] generated into an internal buffer; this is why they must be initialized prior to being invoked.

The **erand48**, **jrand48**, and **nrand48** subroutines require the calling program to provide storage for the successive *x*[*i*] values in the array pointed to by the *xsubi* parameter. This is why these routines do not have to be initialized; the calling program places the desired initial value of *x*[*i*] into the array and pass it as a parameter.

By using different parameters, the **erand48**, **jrand48**, and **nrand48** subroutines allow separate modules of a large program to generate independent sequences of pseudo–random numbers. In other words, the sequence of numbers that one module generates does not depend upon how many times the subroutines are called by other modules.

The **lcong48** subroutine specifies the initial *x*[*i*] value, the multiplier value *a*, and the addend value *c*. The *Parameter* array elements *Parameter*[0–2] specify *x*[*i*], *Parameter*[3–5] specify the multiplier *a*, and *Parameter*[6] specifies the 16–bit addend *c*. After **lcong48** has been called, a subsequent call to either the **srand48** or **seed48** subroutine restores the standard *a* and *c* specified before.

The initializer subroutine **seed48** sets the value of *x*[*i*] to the 48–bit value specified in the array pointed to by the *Seed16v* parameter. In addition, **seed48** returns a pointer to a 48–bit internal buffer that contains the previous value of *x*[*i*] that is used only by **seed48**. The returned pointer allows you to restart the pseudo–random sequence at a given point. Use the pointer to copy the previous *x*[*i*] value into a temporary array. Then call **seed48** with a pointer to this array to resume processing where the original sequence stopped.

The initializer subroutine **srand48** sets the high–order 32 bits of *x*[*i*] to the 32 bits contained in its parameter. The low order 16 bits of *x*[*i*] are set to the arbitrary value 330E16.

Implementation Specifics

These subroutines are part of Base Operating System (BOS) Runtime.

Related Information

The **rand, srand** subroutine, **random**, **srandom**, **initstate**, or **setstate** subroutine.

Subroutines Overview in *AIX General Programming Concepts : Writing and Debugging Programs*.

drem or remainder Subroutine

Purpose

Computes the IEEE Remainder as defined in the IEEE Floating–Point Standard.

Libraries

IEEE Math Library (**libm.a**)
or System V Math Library (**libmsaa.a**)

Syntax

```
#include <math.h>

double drem (x, y)
double x, y;

double remainder (double x, double y);
```

Description

The **drem** or **remainder** subroutines calculate the remainder r equal to x minus n to the x power multiplied by y (*r = x – n* * y), where the n parameter is the integer nearest the exact value of x divided by y `(x/y)`. If `|n -x/y| = 1/2`, then the *n* parameter is an even value. Therefore, the remainder is computed exactly, and the absolute value of r `(|r|)` is less than or equal to the absolute value of y divided by 2 `(|y|/2)`.

The IEEE Remainder differs from the **fmod** subroutine in that the IEEE Remainder always returns an *r* parameter such that `|r|` is less than or equal to `|y|/2`, while FMOD returns an *r* such that `|r|` is less than or equal to `|y|`. The IEEE Remainder is useful for argument reduction for transcendental functions.

Note: Compile any routine that uses subroutines from the **libm.a** library with the **–lm** flag. For example: compile the **drem.c** file:

```
cc drem.c -lm
```

Parameters

x	Specifies double–precision floating–point value.
y	Specifies a double–precision floating–point value.

Return Values

The **drem** or **remainder** subroutines return a NaNQ value for `(x, 0)` and `(+/-INF, y)`.

Error Codes

The **remainder** subroutine returns a NaNQ value for (x, 0.0) [x not equal to NaN] and (+/–INF, y) [y not equal to NaN] and set **errno** to **EDOM**.

Implementation Specifics

This subroutine is part of Base Operating System (BOS) Runtime.

Note: For new development, the **remainder** subroutine is the preferred interface.

Related Information

The **copysign**, **nextafter**, **scalb**, **logb**, or **ilog** subroutine, **floor**, **ceil**, **nearest**, **trunc**, **rint**, **itrunc**, **fmod**, **fabs**, or **uitruns** subroutine.

IEEE Standard for Binary Floating–Point Arithmetic (ANSI/IEEE Standards 754–1985 and 854–1987) describes the IEEE Remainder Function.

Subroutines Overview in *AIX General Programming Concepts : Writing and Debugging Programs*.

_end, _etext, or _edata Identifier

Purpose

Define the first addresses following the program, initialized data, and all data.

Syntax

```
extern _end;

extern _etext;

extern _edata;
```

Description

The external names **_end**, **_etext**, and **_edata** are defined by the loader for all programs. They are not subroutines but identifiers associated with the following addresses:

_etext	The first address following the program text.
_edata	The first address following the initialized data region.
_end	The first address following the data region that is not initialized. The name **end** (with no underscore) defines the same address as does **_end** (with underscore).

The break value of the program is the first location beyond the data. When a program begins running, this location coincides with **end**. However, many factors can change the break value, including:

- The **brk** or **sbrk** subroutine
- The **malloc** subroutine
- The standard I/O subroutines
- The **–p** flag with the **cc** command

Therefore, use the **brk** or **sbrk(0)** subroutine, not the **end** address, to determine the break value of the program.

Implementation Specifics

These identifiers are part of Base Operating System (BOS) Runtime.

Related Information

The **brk** or **sbrk** subroutine, **malloc** subroutine.

Subroutines Overview in *AIX General Programming Concepts : Writing and Debugging Programs*.

ecvt, fcvt, or gcvt Subroutine

Purpose

Converts a floating–point number to a string.

Library

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

Syntax

```
#include <stdlib.h>

char *ecvt (Value, NumberOfDigits, DecimalPointer, Sign;)
double Value;
int  NumberOfDigits, *DecimalPointer, *Sign;

char *fcvt (Value, NumberOfDigits, DecimalPointer, Sign;)
double Value;
int  NumberOfDigits, *DecimalPointer, *Sign;

char *gcvt (Value, NumberOfDigits, Buffer;)
double Value;
int NumberOfDigits;
char *Buffer;
```

Description

The **ecvt**, **fcvt**, and **gcvt** subroutines convert floating–point numbers to strings.

The **ecvt** subroutine converts the *Value* parameter to a null–terminated string and returns a pointer to it. The *NumberOfDigits* parameter specifies the number of digits in the string. The low–order digit is rounded according to the current rounding mode. The **ecvt** subroutine sets the integer pointed to by the *DecimalPointer* parameter to the position of the decimal point relative to the beginning of the string. (A negative number means the decimal point is to the left of the digits given in the string.) The decimal point itself is not included in the string. The **ecvt** subroutine also sets the integer pointed to by the *Sign* parameter to a nonzero value if the *Value* parameter is negative and sets a value of 0 otherwise.

The **fcvt** subroutine operates identically to the **ecvt** subroutine, except that the correct digit is rounded for C or FORTRAN F–format output of the number of digits specified by the *NumberOfDigits* parameter.

Note: In the F–format, the *NumberOfDigits* parameter is the number of digits desired after the decimal point. Large numbers produce a long string of digits before the decimal point, and then *NumberOfDigits* digits after the decimal point. Generally, the **gcvt** and **ecvt** subroutines are more useful for large numbers.

The **gcvt** subroutine converts the *Value* parameter to a null–terminated string, stores it in the array pointed to by the *Buffer* parameter, and then returns the *Buffer* parameter. The **gcvt** subroutine attempts to produce a string of the *NumberOfDigits* parameter significant digits in FORTRAN F–format. If this is not possible, the E–format is used. The **gcvt** subroutine suppresses trailing zeros. The string is ready for printing, complete with minus sign, decimal point, or exponent, as appropriate. The radix character is determined by the current locale (see **setlocale** subroutine). If the **setlocale** subroutine has not been called successfully, the default locale, POSIX, is used. The default locale specifies a **.** (period) as the radix character. The **LC_NUMERIC** category determines the value of the radix character within the current locale.

The **ecvt**, **fcvt**, and **gcvt** subroutines represent the following special values that are specified in ANSI/IEEE standards 754–1985 and 854–1987 for floating–point arithmetic:

Quiet NaN	Indicates a quiet not–a–number (NaNQ)
Signalling NaN	Indicates a signaling NaNS
Infinity	Indicates a INF value

The sign associated with each of these values is stored in the *Sign* parameter.

Note: A value of 0 can be positive or negative. In the IEEE floating–point, zeros also have signs and set the *Sign* parameter appropriately.

Attention: All three subroutines store the strings in a static area of memory whose contents are overwritten each time one of the subroutines is called.

Parameters

Value	Specifies some double–precision floating–point value.
NumberOfDigits	Specifies the number of digits in the string.
DecimalPointer	Specifies the position of the decimal point relative to the beginning of the string.
Sign	Specifies that the sign associated with the return value is placed in the *Sign* parameter. In IEEE floating–point, since 0 can be signed, the *Sign* parameter is set appropriately for signed 0.
Buffer	Specifies a character array for the string.

Implementation Specifics

These subroutines are part of Base Operating System (BOS) Runtime.

Related Information

The **atof**, **strtod**, **atoff**, or **strtof** subroutine, **fp_read_rnd**, or **fp_swap_rnd** subroutine, **printf** subroutine, **scanf** subroutine.

Subroutines Overview in *AIX General Programming Concepts : Writing and Debugging Programs*.

IEEE Standard for Binary Floating–Point Arithmetic (ANSI/IEEE Standards 754–1985 and 854–1987).

erf, erfl, erfc, or erfcl Subroutine

Purpose

Computes the error and complementary error functions.

Libraries

IEEE Math Library (**libm.a**)
or System V Math Library (**libmsaa.a**)

Syntax

```
#include <math.h>

double erf (x)
double x;

long double erfl (x)
long double x;

double erfc (x)
double x;

long double erfcl (x)
long double x;
```

Description

The **erf** and **erfl** subroutines return the error function of the *x* parameter, defined for the **erf** subroutine as the following:

```
erf(x) = (2/sqrt(pi) * (integral [0 to x] of exp(-(t**2)) dt)

erfc(x) = 1.0 - erf(x)
```

The **erfc** and **erfcl** subroutines are provided because of the extreme loss of relative accuracy if `erf(x)` is called for large values of the *x* parameter and the result is subtracted from 1. For example, 12 decimal places are lost when calculating `(1.0 - erf(5))`.

Note: Compile any routine that uses subroutines from the **libm.a** library with the **–lm** flag. To compile the **erf.c** file, for example, enter:

```
cc erf.c –lm
```

Parameters

x	Specifies a double–precision floating–point value.

Implementation Specifics

These subroutines are part of Base Operating System (BOS) Runtime.

Related Information

The **exp**, **expm1**, **log**, **log10**, **log1p**, or **pow** subroutine, **sqrt** or **cbrt** subroutine.

Subroutines Overview in *AIX General Programming Concepts : Writing and Debugging Programs*.

128–Bit long double Floating–Point Format in *AIX General Programming Concepts : Writing and Debugging Programs*.

errlog Subroutine

Purpose

Logs an application error to the system error log.

Library

Run–Time Services Library (**librts.a**)

Syntax

```
#include <sys/errids.h>
int errlog (ErrorStructure, Length)
void *ErrorStructure;
unsigned int Length;
```

Description

The **errlog** subroutine writes an error log entry to the /**dev**/**error** file. The **errlog** subroutine is used by application programs.

The transfer from the **err_rec** structure to the error log is by a **write** subroutine to the /**dev**/**error** special file.

The **errdemon** process reads from the /**dev**/**error** file and writes the error log entry to the system error log. The timestamp, machine ID, node ID, and Software Vital Product Data associated with the resource name (if any) are added to the entry before going to the log.

Parameters

ErrorStructure Points to an error record structure containing an error record. Valid error record structures are typed in the **/usr/include/sys/err_rec.h** file. The two error record structures available are **err_rec** and **err_rec0**. The **err_rec** structure is used when the `detail_data` field is required. When the `detail_data` field is not required, the **err_rec0** structure is used.

```
struct err_rec0 {
   unsigned int error_id;
   char resource_name[ERR_NAMESIZE];
};
struct err_rec {
   unsigned int error_id;
   char resource_name[ERR_NAMESIZE];
   char detail_data[1];
};
```

The fields of the structures **err_rec** and **err_rec0** are:

`error_id` Specifies an index for the system error template database, and is assigned by the **errupdate** command when adding an error template. Use the **errupdate** command with the **–h** flag to get a `#define` statement for this 8–digit hexadecimal index.

`resource_name` Specifies the name of the resource that has detected the error. For software errors, this is the name of a software component or an executable program. For hardware errors, this is the name of a device or system component. It does not indicate that the component is faulty or needs replacement instead, it is used to determine the appropriate diagnostic modules to be used to analyze the error.

`detail_data` Specifies an array from 0 to **ERR_REC_MAX** bytes of user–supplied data. This data may be displayed by the **errpt** command in hexadecimal, alphanumeric, or binary form, according to the `data_encoding` fields in the error log template for this `error_id` field.

Length Specifies the length in bytes of the **err_rec** structure, which is equal to the size of the `error_id` and `resource_name` fields plus the length in bytes of the `detail_data` field.

Return Values

0	The entry was logged successfully.
–1	The entry was not logged.

Implementation Specifics

The **errlog** subroutine is part of Base Operating System (BOS) Runtime.

Files

/dev/error	Provides standard device driver interfaces required by the error log component.
/usr/include/sys/errids.h	Contains definitions for error IDs.
/usr/include/sys/err_rec.h	Contains structures defined as arguments to the **errsave** kernel service and the **errlog** subroutine.
/var/adm/ras/errlog	Maintains the system error log.

Related Information

The **errclear** command, **errdead** command, **errinstall** command, **errlogger** command, **errmsg** command, **errpt** command, **errstop** command, **errupdate** command.

The /**dev**/**error** special file.

The **errdemon** daemon.

The **errsave** kernel service.

Error Logging Overview in *AIX Version 4.3 Problem Solving Guide and Reference*.

exec: execl, execle, execlp, execv, execve, execvp, or exect Subroutine

Purpose

Executes a file.

Library

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

Syntax

```
#include <unistd.h>



extern
char **environ;



int execl (
Path,
Argument0 [, Argument1, ...], 0)
const char *Path, *Argument0, *Argument
1, ...;



int execle (
Path,
Argument0 [, Argument1, ...], 0,

EnvironmentPointer)
const
char *Path, *Argument0, *Argum
ent
1, ...;
char *const EnvironmentPointer[ ];



int execlp (
File,
Argument0 [, Argument1
, ...], 0)
const char *File, *Argument0, *Argument
1, ...;



int execv (
Path,
ArgumentV)
const char *Path;
char *const ArgumentV[ ];
```

```
int execve (
Path,
ArgumentV,

EnvironmentPointer)
const char *Path;
char
*const ArgumentV[ ], *EnvironmentPointer

[ ];
```

```
int execvp (
File,
ArgumentV)
const char *File;
char *const ArgumentV[ ];
```

```
int exect (
Path,
ArgumentV,
EnvironmentPointer)
char *Path, *ArgumentV, *EnvironmentPointer [ ];
```

Description

The **exec** subroutine, in all its forms, executes a new program in the calling process. The **exec** subroutine does not create a new process, but overlays the current program with a new one, which is called the *new–process image*. The new–process image file can be one of three file types:

- An executable binary file in XCOFF file format. .
- An executable text file that contains a shell procedure (only the **execlp** and **execvp** subroutines allow this type of new–process image file).
- A file that names an executable binary file or shell procedure to be run.

The new–process image inherits the following attributes from the calling process image: session membership, supplementary group IDs, process signal mask, and pending signals.

The last of the types mentioned is recognized by a header with the following syntax:

```
#! Path [String]
```

The **#!** is the file *magic number*, which identifies the file type. The path name of the file to be executed is specified by the *Path* parameter. The *String* parameter is an optional character string that contains no tab or space characters. If specified, this string is passed to the new process as an argument in front of the name of the new–process image file. The header must be terminated with a new–line character. When called, the new process passes the *Path* parameter as *ArgumentV*[0]. If a *String* parameter is specified in the new process image file, the **exec** subroutine sets *ArgumentV*[0] to the *String* and *Path* parameter values concatenated together. The rest of the arguments passed are the same as those passed to the **exec** subroutine.

The **exec** subroutine attempts to cancel outstanding asynchronous I/O requests by this process. If the asynchronous I/O requests cannot be canceled, the application is blocked until the requests have completed.

The **exec** subroutine is similar to the **load** subroutine, except that the **exec** subroutine does not have an explicit library path parameter. Instead, the **exec** subroutine uses the **LIBPATH** environment variable. The **LIBPATH** variable is ignored when the program that the **exec** subroutine is run on has more privilege than the calling program, for example, the **suid** program.

The **exect** subroutine is included for compatibility with older programs being traced with the **ptrace** command. The program being executed is forced into hardware single–step mode.

Note: **exect** is not supported in 64–bit mode.

Parameters

Path	Specifies a pointer to the path name of the new–process image file. If Network File System (NFS) is installed on your system, this path can cross into another node. Data is copied into local virtual memory before proceeding.
File	Specifies a pointer to the name of the new–process image file. Unless the *File* parameter is a full path name, the path prefix for the file is obtained by searching the directories named in the **PATH** environment variable. The initial environment is supplied by the shell. **Note:** The **execlp** subroutine and the **execvp** subroutine take *File* parameters, but the rest of the **exec** subroutines take *Path* parameters. (For information about the environment, see the **environment** miscellaneous facility and the **sh** command.)
Argument0 [, *Argument1*, ...]	Point to null–terminated character strings. The strings constitute the argument list available to the new process. By convention, at least the *Argument0* parameter must be present, and it must point to a string that is the same as the *Path* parameter or its last component.
ArgumentV	Specifies an array of pointers to null–terminated character strings. These strings constitute the argument list available to the new process. By convention, the *ArgumentV* parameter must have at least one element, and it must point to a string that is the same as the *Path* parameter or its last component. The last element of the *ArgumentV* parameter is a null pointer.
EnvironmentPointer	An array of pointers to null–terminated character strings. These strings constitute the environment for the new process. The last element of the *EnvironmentPointer* parameter is a null pointer.

When a C program is run, it receives the following parameters:

```
main (ArgumentCount,
ArgumentV, EnvironmentPointer)
int ArgumentCount;
char *ArgumentV[ ], *EnvironmentPointer[
];
```

In this example, the *ArgumentCount* parameter is the argument count, and the *ArgumentV* parameter is an array of character pointers to the arguments themselves. By convention, the value of the *ArgumentCount* parameter is at least 1, and the *ArgumentV*[0] parameter points to a string containing the name of the new–process image file.

The **main** routine of a C language program automatically begins with a runtime start–off routine. This routine sets the **environ** global variable so that it points to the environment array passed to the program in *EnvironmentPointer*. You can access this global variable by including the following declaration in your program:

```
extern char **environ;
```

The **execl**, **execv**, **execlp**, and **execvp** subroutines use the **environ** global variable to pass the calling process current environment to the new process.

File descriptors open in the calling process remain open, except for those whose **close–on–exec** flag is set. For those file descriptors that remain open, the file pointer is unchanged. (For information about file control, see the **fcntl.h** file.)

The state–of–conversion descriptors and message–catalog descriptors in the new process image are undefined. For the new process, an equivalent of the **setlocale** subroutine, specifying the **LC_ALL** value for its category and the **"C"** value for its locale, is run at startup.

If the new program requires shared libraries, the **exec** subroutine finds, opens, and loads each of them into the new–process address space. The referenced counts for shared libraries in use by the issuer of the **exec** are decremented. Shared libraries are searched for in the directories listed in the **LIBPATH** environment variable. If any of these files is remote, the data is copied into local virtual memory.

The **exec** subroutines reset all caught signals to the default action. Signals that cause the default action continue to do so after the **exec** subroutines. Ignored signals remain ignored, the signal mask remains the same, and the signal stack state is reset. (For information about signals, see the **sigaction** subroutine.)

If the *SetUserID* mode bit of the new–process image file is set, the **exec** subroutine sets the effective user ID of the new process to the owner ID of the new–process image file. Similarly, if the *SetGroupID* mode bit of the new–process image file is set, the effective group ID of the new process is set to the group ID of the new–process image file. The real user ID and real group ID of the new process remain the same as those of the calling process. (For information about the *SetID* modes, see the **chmod** subroutine.)

At the end of the **exec** operation the saved user ID and saved group ID of the process are always set to the effective user ID and effective group ID, respectively, of the process.

When one or both of the set ID mode bits is set and the file to be executed is a remote file, the file user and group IDs go through outbound translation at the server. Then they are transmitted to the client node where they are translated according to the inbound translation table. These translated IDs become the user and group IDs of the new process.

Note: setuid and **setgid** bids on shell scripts do not affect user or group IDs of the process finally executed.

Profiling is disabled for the new process.

The new process inherits the following attributes from the calling process:

- Nice value (see the **getpriority** subroutine, **setpriority** subroutine, **nice** subroutine)
- Process ID
- Parent process ID
- Process group ID
- **semadj** values (see the **semop** subroutine)
- tty group ID (see the **exit**, **atexit**, or **_exit** subroutine, **sigaction** subroutine)

- **trace** flag (see request 0 of the **ptrace** subroutine)
- Time left until an alarm clock signal (see the **incinterval** subroutine, **setitimer** subroutine, and **alarm** subroutine)
- Current directory
- Root directory
- File–mode creation mask (see the **umask** subroutine)
- File size limit (see the **ulimit** subroutine)
- Resource limits (see the **getrlimit** subroutine, **setrlimit** subroutine, and **vlimit** subroutine)
- `tms_utime` , `tms_stime` , `tms_cutime` , and `tms_ctime` fields of the **tms** structure (see the **times** subroutine)
- Login user ID

Upon successful completion, the **exec** subroutines mark for update the `st_atime` field of the file.

Examples

1. To run a command and pass it a parameter, enter:

   ```
   execlp("li", "li", "-al", 0);
   ```

   The **execlp** subroutine searches each of the directories listed in the **PATH** environment variable for the **li** command, and then it overlays the current process image with this command. The **execlp** subroutine is not returned, unless the **li** command cannot be executed.

   Note: This example does not run the shell command processor, so operations interpreted by the shell, such as using wildcard characters in file names, are not valid.

2. To run the shell to interpret a command, enter:

   ```
   execl("/usr/bin/sh", "sh", "-c", "li -l *.c",
   0);
   ```

   This runs the **sh** command with the **–c** flag, which indicates that the following parameter is the command to be interpreted. This example uses the **execl** subroutine instead of the **execlp** subroutine because the full path name **/usr/bin/sh** is specified, making a path search unnecessary.

   Running a shell command in a child process is generally more useful than simply using the **exec** subroutine, as shown in this example. The simplest way to do this is to use the **system** subroutine.

3. The following is an example of a new–process file that names a program to be run:

   ```
   #! /usr/bin/awk -f
   { for (i = NF; i > 0; --i) print $i }
   ```

   If this file is named `reverse` , entering the following command on the command line:

   ```
   reverse chapter1 chapter2
   ```

   This command runs the following command:

   ```
   /usr/bin/awk -f reverse chapter1 chapter2
   ```

   Note: The **exec** subroutines use only the first line of the new–process image file and ignore the rest of it. Also, the **awk** command interprets the text that follows a # (pound sign) as a comment.

Return Values

Upon successful completion, the **exec** subroutines do not return because the calling process image is overlaid by the new–process image. If the **exec** subroutines return to the calling process, the value of –1 is returned and the **errno** global variable is set to identify the error.

Error Codes

If the **exec** subroutine is unsuccessful, it returns one or more of the following error codes:

EACCES	The new–process image file is not an ordinary file.
EACCES	The mode of the new–process image file denies execution permission.
ENOEXEC	The **exec** subroutine is neither an **execlp** subroutine nor an **execvp** subroutine. The new–process image file has the appropriate access permission, but the magic number in its header is not valid.
ENOEXEC	The new–process image file has a valid magic number in its header, but the header is damaged or is incorrect for the machine on which the file is to be run.
ETXTBSY	The new–process image file is a pure procedure (shared text) file that is currently open for writing by some process.
ENOMEM	The new process requires more memory than is allowed by the system–imposed maximum, the **MAXMEM** compiler option.
E2BIG	The number of bytes in the new–process argument list is greater than the system–imposed limit. This limit is defined as the **NCARGS** parameter value in the **sys/param.h** file.
EFAULT	The *Path*, *ArgumentV*, or *EnviromentPointer* parameter points outside of the process address space.
EPERM	The *SetUserID* or *SetGroupID* mode bit is set on the process image file. The translation tables at the server or client do not allow translation of this user or group ID.

If the **exec** subroutine is unsuccessful because of a condition requiring path name resolution, it returns one or more of the following error codes:

EACCES	Search permission is denied on a component of the path prefix. Access could be denied due to a secure mount.
EFAULT	The *Path* parameter points outside of the allocated address space of the process.
EIO	An input/output (I/O) error occurred during the operation.
ELOOP	Too many symbolic links were encountered in translating the *Path* parameter.
ENAMETOOLONG	A component of a path name exceeded 255 characters and the process has the **disallow truncation** attribute (see the **ulimit** subroutine), or an entire path name exceeded 1023 characters.
ENOENT	A component of the path prefix does not exist.
ENOENT	A symbolic link was named, but the file to which it refers does not exist.
ENOENT	The path name is null.
ENOTDIR	A component of the path prefix is not a directory.
ESTALE	The root or current directory of the process is located in a virtual file system that has been unmounted.

In addition, some errors can occur when using the new–process file after the old process image has been overwritten. These errors include problems in setting up new data and stack registers, problems in mapping a shared library, or problems in reading the new–process file. Because returning to the calling process is not possible, the system sends the **SIGKILL** signal to the process when one of these errors occurs.

If an error occurred while mapping a shared library, an error message describing the reason for error is written to standard error before the signal **SIGKILL** is sent to the process. If a shared library cannot be mapped, the subroutine returns one of the following error codes:

ENOENT	One or more components of the path name of the shared library file do not exist.
ENOTDIR	A component of the path prefix of the shared library file is not a directory.
ENAMETOOLONG	A component of a path name prefix of a shared library file exceeded 255 characters, or an entire path name exceeded 1023 characters.
EACCES	Search permission is denied for a directory listed in the path prefix of the shared library file.
EACCES	The shared library file mode denies execution permission.
ENOEXEC	The shared library file has the appropriate access permission, but a magic number in its header is not valid.
ETXTBSY	The shared library file is currently open for writing by some other process.
ENOMEM	The shared library requires more memory than is allowed by the system–imposed maximum.
ESTALE	The process root or current directory is located in a virtual file system that has been unmounted.

If NFS is installed on the system, the **exec** subroutine can also fail if the following is true:

ETIMEDOUT	The connection timed out.

Implementation Specifics

These subroutines are part of Base Operating System (BOS) Runtime.

Note: Currently, a Graphics Library program cannot be overlaid with another Graphics Library program. The overlaying program can be a nongraphics program. For additional information, see the **/usr/lpp/GL/README** file.

Related Information

The **alarm** or **incinterval** subroutine, **chmod** or **fchmod** subroutine, **exit** subroutine, **fcntl** subroutine, **fork** subroutine, **getrusage** or **times** subroutine, **nice** subroutine, **profil** subroutine, **ptrace** subroutine.

The **semop** subroutine, **settimer** subroutine, **sigaction**, **signal**, or **sigvec** subroutine, **shmat** subroutine, **system** subroutine, **ulimit** subroutine, **umask** subroutine.

The **awk** command, **ksh** command, **sh** command.

The **environment** file.

The XCOFF object (**a.out**) file format.

The **varargs** macros.

Asynchronous I/O Overview in *AIX Kernel Extensions and Device Support Programming Concepts*.

exit, atexit, or _exit Subroutine

Purpose

Terminates a process.

Library

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

Syntax

```
#include <stdlib.h>

void exit (Status)
int Status;

void _exit (Status)
int Status;

#include <sys/limits.h>

int atexit (Function)
void (*Function) (void);
```

Description

The **exit** subroutine terminates the calling process after calling the standard I/O library **_cleanup** function to flush any buffered output. Also, it calls any functions registered previously for the process by the **atexit** subroutine. The **atexit** subroutine registers functions called at normal process termination for cleanup processing. Normal termination occurs as a result of either a call to the **exit** subroutine or a **return** statement in the **main** function.

Each function a call to the **atexit** subroutine registers must return. This action ensures that all registered functions are called.

Finally, the **exit** subroutine calls the **_exit** subroutine, which completes process termination and does not return. The **_exit** subroutine terminates the calling process and causes the following to occur:

- The **_exit** subroutine attempts to cancel outstanding asynchronous I/O requests by this process. If the asynchronous I/O requests cannot be canceled, the application is blocked until the requests have completed.
- All of the file descriptors open in the calling process are closed. If Network File System (NFS) is installed on your system, some of these files can be remote. Because the **_exit** subroutine terminates the process, any errors encountered during these close operations go unreported.
- If the parent process of the calling process is running a **wait** call, it is notified of the termination of the calling process and the low–order 8 bits (that is, bits 0377 or 0xFF) of the *Status* parameter are made available to it.
- If the parent process is not running a **wait** call when the child process terminates, it may still do so later on, and the child's status is returned to it at that time.
- The parent process is sent a **SIGCHLD** signal when a child process terminates; however, since the default action for this signal is to ignore it, the signal is usually not seen.
- Terminating a process by exiting does not terminate its child processes.
- Each attached shared memory segment is detached and the **shm_nattch** value in the data structure associated with its shared memory identifier is decremented by 1.
- For each semaphore for which the calling process has set a **semadj** value, that **semadj** value is added to the **semval** of the specified semaphore. (The **semop** subroutine provides information about semaphore operations.)

- If the process has a process lock, text lock, or data lock, an **unlock** routine is performed. (See the **plock** subroutine.)
- An accounting record is written on the accounting file if the system accounting routine is enabled. (The **acct** subroutine provides information about enabling accounting routines.)
- Locks set by the **fcntl**, **lockf**, and **flock** subroutines are removed.
- If the parent process of the calling process is not ignoring a **SIGCHLD** signal, the calling process is transformed into a zombie process, and its parent process is sent a **SIGCHLD** signal to notify it of the end of a child process.
- A zombie process occupies a slot in the process table, but has no other space allocated to it either in user or kernel space. The process table slot that it occupies is partially overlaid with time–accounting information to be used by the **times** subroutine. (See the **sys/proc.h** file.)
- A process remains a zombie until its parent issues one of the **wait** subroutines. At this time, the zombie is *laid to rest* (deleted), and its process table entry is released.
- Terminating a process does not terminate its child processes. Instead, the parent process ID of all of the calling–process child processes and zombie child processes is set to the process ID of **init**. The **init** process inherits each of these processes, and catches their **SIGCHLD** signals and calls the **wait** subroutine for each of them.
- If the process is a controlling process, the **SIGHUP** signal is sent to each process in the foreground process group of the controlling terminal belonging to the calling process.
- If the process is a controlling process, the controlling terminal associated with the session is disassociated from the session, allowing it to be acquired by a new controlling process.
- If the exit of the process causes a process group to become orphaned, and if any member of the newly orphaned process group is stopped, a **SIGHUP** signal followed by a **SIGCONT** signal will be sent to each process in the newly orphaned process group.

Note: The system **init** process is used to assist cleanup of terminating processes. If the code for the **init** process is replaced, the program must be prepared to accept **SIGCHLD** signals and issue a **wait** call for each.

Parameters

Status Indicates the status of the process.

Function Specifies a function to be called at normal process termination for cleanup processing. You may specify a number of functions to the limit set by the **ATEXIT_MAX** function, which is defined in the **sys/limits.h** file. A pushdown stack of functions is kept so that the last function registered is the first function called.

Return Values

Upon successful completion, the **atexit** subroutine returns a value of 0. Otherwise, a nonzero value is returned. The **exit** and **_exit** subroutines do not return a value.

Implementation Specifics

These subroutines are part of Base Operating System (BOS) Runtime.

Related Information

The **acct** subroutine, **lockfx**, **lockf**, or **flock** subroutines, **sigaction**, **sigvec**, or **signal** subroutine, **times** subroutine, **wait**, **waitpid**, or **wait3** subroutine.

Asynchronous I/O Overview in *AIX Kernel Extensions and Device Support Programming Concepts*.

exp, expl, expm1, log, logl, log10, log10l, log1p, pow, or powl Subroutine

Purpose

Computes exponential, logarithm, and power functions.

Libraries

IEEE Math Library (**libm.a**)
or System V Math Library (**libmsaa.a**)

Syntax

```
#include <math.h>

double exp (x)
double x;

long double expl (x)
long double x;

double expm1 (x)
double x;

double log (x)
double x;

long double logl (x)
long double x;

double log10 (x)
double x;

long double log10l (x)
long double x;

double log1p (x)
double x;

double pow (x, y)
double x, y;

long double powl (x, y)
long double x, y;
```

Description

These subroutines are used to compute exponential, logarithm, and power functions.

The **exp** and **expl** subroutines returns `exp (x)`.

The **expm1** subroutine returns `exp (x)-1`.

The **log** and **logl** subroutines return the natural logarithm of the *x* parameter. The value of the *x* parameter must be positive.

The **log10** and **log10l** subroutines return the logarithm base 10 of the *x* parameter . The value of *x* must be positive.

The **log1p** subroutine returns `log (1 + x)`.

The **pow** and **powl** subroutines return x**y. If the *x* parameter is negative or 0, then the *y* parameter must be an integer. If the *y* parameter is 0, then the **pow** and **powl** subroutines return 1.0 for all the *x* parameters.

The **expm1** and **log1p** subroutines are useful to guarantee that financial calculations of ((1+x**n) –1) / x, are accurate when the *x* parameter is small (for example, when calculating small daily interest rates).

```
expm1(n * log1p(x))/x
```

These subroutines also simplify writing accurate inverse hyperbolic functions.

Note: Compile any routine that uses subroutines from the **libm.a** library with the **–lm** flags. For example: to compile the **pow.c** file, enter:

cc pow.c –lm

Parameters

x	Specifies some double–precision floating–point value.
y	Specifies some double–precision floating–point value.

Error Codes

When using the **libm.a** library:

exp	If the correct value would overflow, the **exp** subroutine returns a **HUGE_VAL** value and the **errno** global variable is set to a **ERANGE** value.
log	If the *x* parameter is less than 0, the **log** subroutine returns a **NaNQ** value and sets **errno** to **EDOM**. If x= 0, the **log** subroutine returns a –**HUGE_VAL** value but does not modify **errno**.
log10	If the *x* parameter is less than 0, the **log10** subroutine returns a **NaNQ** value and sets **errno** to **EDOM**. If x= 0, the **log10** subroutine returns a –**HUGE_VAL** value but does not modify **errno**.
pow	If the correct value overflows, the **pow**subroutine returns a **HUGE_VAL** value and sets **errno** to **ERANGE**. If the *x* parameter is negative and the *y* parameter is not an integer, the **pow** subroutine returns a **NaNQ** value and sets **errno** to **EDOM**. If `x=0` and the *y* parameter is negative, the **pow** subroutine returns a **HUGE_VAL** value but does not modify **errno.**
powl	If the correct value overflows, the **powl**subroutine returns a **HUGE_VAL** value and sets **errno** to **ERANGE**. If the *x* parameter is negative and the *y* parameter is not an integer, the **powl** subroutine returns a **NaNQ** value and sets **errno** to **EDOM**. If `x=0` and the *y* parameter is negative, the **powl** subroutine returns a **HUGE_VAL** value but does not modify **errno.**

When using **libmsaa.a**(**–lmsaa**):

exp	If the correct value would overflow, the **exp** subroutine returns a **HUGE_VAL** value. If the correct value would underflow, the **exp** subroutine returns 0. In both cases **errno** is set to **ERANGE**.
expl	If the correct value would overflow, the **expl** subroutine returns a **HUGE_VAL** value. If the correct value would underflow, the **expl** subroutine returns 0. In both cases **errno** is set to **ERANGE**.
log	If the *x* parameter is not positive, the **log** subroutine returns a –**HUGE_VAL** value, and sets **errno** to a **EDOM** value. A message indicating DOMAIN error (or SING error when *x* = 0) is output to standard error.
logl	If the *x* parameter is not positive, the **logl** subroutine returns the –**HUGE_VAL** value, and sets **errno** to **EDOM**. A message indicating DOMAIN error (or SING error when *x* = 0) is output to standard error.

log10 If the *x* parameter is not positive, the **log10** subroutine returns a –**HUGE_VAL** value and sets **errno** to **EDOM**. A message indicating DOMAIN error (or SING error when *x* = 0) is output to standard error.

log10l If the *x* parameter is not positive, the **log10l** subroutine returns a –**HUGE_VAL** value and sets **errno** to **EDOM**. A message indicating DOMAIN error (or SING error when *x* = 0) is output to standard error.

pow If `x=0` and the *y* parameter is not positive, or if the *x* parameter is negative and the *y* parameter is not an integer, the **pow** subroutine returns 0 and sets **errno** to **EDOM**. In these cases a message indicating DOMAIN error is output to standard error. When the correct value for the **pow** subroutine would overflow or underflow, the **pow** subroutine returns:

```
+HUGE_VAL

 OR

 -HUGE_VAL

 OR

 0
```

When using either the **libm.a** library or the **libsaa.a** library:

expl If the correct value overflows, the **expl** subroutine returns a **HUGE_VAL** value and **errno** is set to **ERANGE**.

logl If *x*<0, the **logl** subroutine returns a **NaNQ** value

log10l If *x* < 0, **log10l** returns the value **NaNQ** and sets **errno** to **EDOM**. If *x* equals 0, **log10l** returns the value –**HUGE_VAL** but does not modify **errno**.

powl If the correct value overflows, **powl** returns **HUGE_VAL** and **errno** to **ERANGE**. If *x* is negative and *y* is not an integer, **powl** returns **NaNQ** and sets **errno** to **EDOM**. If *x* = zero and *y* is negative, **powl** returns a **HUGE_VAL** value but does not modify **errno**.

These error–handling procedures may be changed with the **matherr** subroutine when using the **libmsaa.a** library.

Implementation Specifics

The **exp, expl, expm1, log, logl, log10, log10l, log1p, pow, or powl** subroutines are part of Base Operating System (BOS) Runtime.

The **expm1** and **log1p** subroutines are not part of the ANSI C Library.

Related Information

The **hypot** or **cabs** subroutine, **matherr** subroutine, **sinh**, **cosh**, or **tanh** subroutine.

Subroutines Overview in *AIX General Programming Concepts : Writing and Debugging Programs*.

128–Bit long double Floating–Point Format in *AIX General Programming Concepts : Writing and Debugging Programs*.

fattach Subroutine

Purpose

Attaches a STREAMS–based file descriptor to a file.

Library

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

Syntax

```
#include <stropts.h>
int fattach(int fildes, const char *path);
```

Description

The **fattach** subroutine attaches a STREAMS–based file descriptor to a file, effectively associating a pathname with *fildes*. The *fildes* argument must be a valid open file descriptor associated with a STREAMS file. The *path* argument points to a pathname of an existing file. The process must have appropriate privileges, or must be the owner of the file named by *path* and have write permission. A successful call to **fattach** subroutine causes all pathnames that name the file named by *path* to name the STREAMS file associated with *fildes*, until the STEAMS file is detached from the file. A STREAMS file can be attached to more than one file and can have several pathnames associated with it.

The attributes of the named STREAMS file are initialized as follows: the permissions, user ID, group ID, and times are set to those of the file named by *path*, the number of links is set to 1, and the size and device identifier are set to those of the STREAMS file associated with *fildes*. If any attributes of the named STREAMS file are subsequently changed (for example, by **chmod** subroutine), neither the attributes of the underlying file nor the attributes of the STREAMS file to which *fildes* refers are affected.

File descriptors referring to the underlying file, opened prior to an **fattach** subroutine, continue to refer to the underlying file.

Parameters

fildes	A file descriptor identifying an open STREAMS–based object.
path	An existing pathname which will be associated with *fildes*.

Return Value

0	Successful completion.
–1	Not successful and *errno* set to one of the following.

Errno Value

EACCES	Search permission is denied for a component of the path prefix, or the process is the owner of *path* but does not have write permission on the file named by *path*.
EBADF	The file referred to by *fildes* is not an open file descriptor.
ENOENT	A component of *path* does not name an existing file or *path* is an empty string.
ENOTDIR	A component of the path prefix is not a directory.
EPERM	The effective user ID of the process is not the owner of the file named by *path* and the process does not have appropriate privilege.

EBUSY	The file named by *path* is currently a mount point or has a STREAMS file attached to it.
ENAMETOOLONG	The size of *path* exceeds {**PATH_MAX**}, or a component of *path* is longer than {**NAME_MAX**}.
ELOOP	Too many symbolic links wer encountered in resolving *path*.
EINVAL	The *fildes* argument does not refer to a STREAMS file.
ENOMEM	Insufficient storage space is available.

Implementation Specifics

These subroutines are part of Base Operating System (BOS) Runtime.

Related Specifics

The **fdetach** subroutine, **isastream** subroutine.

fchdir Subroutine

Purpose

Directory pointed to by the file descriptor becomes the current working directory.

Library

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

Syntax

#include <unistd.h>

int fchdir (int *Fildes***)**

Description

The **fchdir** subroutine causes the directory specified by the *Fildes* parameter to become the current working directory.

Parameter

Fildes	A file descriptor identifying an open directory obtained from a call to the **open** subroutine.

Return Values

0	Successful completion
–1	Not successful and **errno** set to one of the following.

Error Codes

EACCES	Search access if denied.
EBADF	The file referred to by *Fildes* is not an open file descriptor.
ENOTDIR	The open file descriptor does not refer to a directory.

Related Information

The **chdir** subroutine, **chroot** subroutine, **open** subroutine.

fclear or fclear64 Subroutine

Purpose

Makes a hole in a file.

Library

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

Syntax

```
off_t fclear (FileDescriptor, NumberOfBytes)
int FileDescriptor;
off_t NumberOfBytes;
```

Note: The **fclear64** subroutine applies to Version 4.2 and later releases.

```
off64_t fclear64 (FileDescriptor, NumberOfBytes)
int FileDescriptor;
off64_t NumberOfBytes;
```

Description

Note: The **fclear64** subroutine applies to Version 4.2 and later releases.

The **fclear** and **fclear64** subroutines zero the number of bytes specified by the *NumberOfBytes* parameter starting at the current file pointer for the file specified in the *FileDescriptor* parameter. If Network File System (NFS) is installed on your system, this file can reside on another node.

The **fclear** subroutine can only clear up to **OFF_MAX** bytes of the file while **fclear64** can clear up to the maximum file size.

The **fclear** and **fclear64** subroutines cannot be applied to a file that a process has opened with the **O_DEFER** mode.

Successful completion of the **fclear** and **fclear64** subroutines clear the SetUserID bit (**S_ISUID**) of the file if any of the following are true:

- The calling process does not have root user authority.
- The effective user ID of the calling process does not match the user ID of the file.
- The file is executable by the group (**S_IXGRP**) or others (**S_IXOTH**).

This subroutine also clears the SetGroupID bit (**S_ISGID**) if:

- The file does not match the effective group ID or one of the supplementary group IDs of the process,

  OR

- The file is executable by the owner (**S_IXUSR**) or others (**S_IXOTH**).

Note: Clearing of the SetUserID and SetGroupID bits can occur even if the subroutine fails because the data in the file was modified before the error was detected.

In the large file enabled programming environment, **fclear** is redefined to be **fclear64**.

Parameters

FileDescriptor Indicates the file specified by the *FileDescriptor* parameter must be open for writing. The FileDescriptor is a small positive integer used instead of the file name to identify a file. This function differs from the logically equivalent write operation in that it returns full blocks of binary zeros to the file system, constructing holes in the file.

NumberOfBytes Indicates the number of bytes that the seek pointer is advanced. If you use the **fclear** and **fclear64** subroutines past the end of a file, the rest of the file is cleared and the seek pointer is advanced by *NumberOfBytes*. The file size is updated to include this new hole, which leaves the current file position at the byte immediately beyond the new end–of–file pointer.

Return Values

Upon successful completion, a value of *NumberOfBytes* is returned. Otherwise, a value of –1 is returned and the **errno** global variable is set to indicate the error.

Error Codes

The **fclear** and **fclear64** subroutines fail if one or more of the following are true:

EIO I/O error.

EBADF The *FileDescriptor* value is not a valid file descriptor open for writing.

EINVAL The file is not a regular file.

EMFILE The file is mapped **O_DEFER** by one or more processes.

EAGAIN The write operation in the **fclear** subroutine failed due to an enforced write lock on the file.

EFBIG The current offset plus *NumberOfBytes* is exceeds the offset maximum established in the open file description associated with *FileDescriptor*.

EFBIG An attempt was made to write a file that exceeds the process' file size limit or the maximum file size. If the user has set the environment variable **XPG_SUS_ENV=ON** prior to execution of the process, then the **SIGXFSZ** signal is posted to the process when exceeding the process' file size limit.

If NFS is installed on the system the **fclear** and **fclear64** subroutines can also fail if the following is true:

ETIMEDOUT The connection timed out.

Implementation Specifics

This subroutine is part of Base Operating System (BOS) Runtime.

Related Information

The **open**, **openx**, or **creat** subroutine, **truncate** or **ftruncate** subroutines.

Files, Directories, and File Systems for Programmers in *AIX General Programming Concepts : Writing and Debugging Programs*.

fclose or fflush Subroutine

Purpose

Closes or flushes a stream.

Library

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

Syntax

```
#include <stdio.h>

int fclose (Stream)
FILE *Stream;

int fflush (Stream)
FILE *Stream;
```

Description

The **fclose** subroutine writes buffered data to the stream specified by the *Stream* parameter, and then closes the stream. The **fclose** subroutine is automatically called for all open files when the **exit** subroutine is invoked.

The **fflush** subroutine writes any buffered data for the stream specified by the *Stream* parameter and leaves the stream open. The **fflush** subroutine marks the `st_ctime` and `st_mtime` fields of the underlying file for update.

If the *Stream* parameter is a null pointer, the **fflush** subroutine performs this flushing action on all streams for which the behavior is defined.

Parameters

Stream	Specifies the output stream.

Return Values

Upon successful completion, the **fclose** and **fflush** subroutines return a value of 0. Otherwise, a value of EOF is returned.

Error Codes

If the **fclose** and **fflush** subroutines are unsuccessful, the following errors are returned through the **errno** global variable:

EAGAIN	The **O_NONBLOCK** flag is set for the file descriptor underlying the *Stream* parameter and the process would be delayed in the write operation.
EBADF	The file descriptor underlying *Stream* is not valid.
EFBIG	An attempt was made to write a file that exceeds the process' file size limit or the maximum file size. See the **ulimit** subroutine.
EFBIG	The file is a regular file and an attempt was made to write at or beyond the offset maximum associated with the corresponding stream.
EINTR	The **fflush** subroutine was interrupted by a signal.

EIO	The process is a member of a background process group attempting to write to its controlling terminal, the **TOSTOP** signal is set, the process is neither ignoring nor blocking the **SIGTTOU** signal and the process group of the process is orphaned. This error may also be returned under implementation–dependent conditions.
ENOSPC	No free space remained on the device containing the file.
EPIPE	An attempt is made to write to a pipe or FIFO that is not open for reading by any process. A **SIGPIPE** signal is sent to the process.
ENXIO	A request was made of a non–existent device, or the request was outside the capabilities of the device

Implementation Specifics

These subroutines are part of Base Operating System (BOS) Runtime.

Related Information

The **close** subroutine, **exit**, **atexit**, or **_exit** subroutine, **fopen**, **freopen,** or **fdopen** subroutine, **setbuf**, **setvbuf**, **setbuffer**, or **setlinebuf** subroutine.

Input and Output Handling Programmer's Overview in *AIX General Programming Concepts : Writing and Debugging Programs*.

fcntl, dup, or dup2 Subroutine

Purpose

Controls open file descriptors.

Library

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

Syntax #include <fcntl.h>

int fcntl (*FileDescriptor***,** *Command***,** *Argument***)**
int *FileDescriptor***,** *Command***,** *Argument***;**

#include <unistd.h>

int dup2(*Old***,** *New***)**
int *Old***,** *New***;**

int dup(*FileDescriptor***)**
int *FileDescriptor***;**

Description

The **fcntl** subroutine performs controlling operations on the open file specified by the *FileDescriptor* parameter. If Network File System (NFS) is installed on your system, the open file can reside on another node. The **fcntl** subroutine is used to:

- Duplicate open file descriptors.
- Set and get the file–descriptor flags.
- Set and get the file–status flags.
- Manage record locks.
- Manage asynchronous I/O ownership.
- Close multiple files.

The **fcntl** subroutine can provide the same functions as the **dup** and **dup2** subroutines.

General Record Locking Information

A lock is either an *enforced* or *advisory lock* and either a *read* or a *write lock*.

> **Attention:** Buffered I/O does not work properly when used with file locking. Do not use the standard I/O package routines on files that are going to be locked.

For a lock to be an enforced lock, the Enforced Locking attribute of the file must be set; for example, the **S_ENFMT** bit must be set, but the **S_IXGRP**, **S_IXUSR**, and **S_IXOTH** bits must be clear. Otherwise, the lock is an advisory lock. A given file can have advisory or enforced locks, but not both. The description of the **sys/mode.h** file includes a description of file attributes.

When a process holds an enforced lock on a section of a file, no other process can access that section of the file with the **read** or **write** subroutine. In addition, the **open** and **ftruncate** subroutines cannot truncate the locked section of the file, and the **fclear** subroutine cannot modify the locked section of the file. If another process attempts to read or modify the locked section of the file, the process either sleeps until the section is unlocked or returns with an error indication.

When a process holds an advisory lock on a section of a file, no other process can lock that section of the file (or an overlapping section) with the **fcntl** subroutine. (No other subroutines are affected.) As a result, processes must voluntarily call the **fcntl** subroutine in order to make advisory locks effective.

When a process holds a read lock on a section of a file, other processes can also set read locks on that section or on subsets of it. Read locks are also called *shared* locks.

A read lock prevents any other process from setting a write lock on any part of the protected area. If the read lock is also an enforced lock, no other process can modify the protected area.

The file descriptor on which a read lock is being placed must have been opened with read access.

When a process holds a write lock on a section of a file, no other process can set a read lock or a write lock on that section. Write locks are also called *exclusive* locks. Only one write lock and no read locks can exist for a specific section of a file at any time.

If the lock is also an enforced lock, no other process can read or modify the protected area.

The following general rules about file locking apply:

- Changing or unlocking part of a file in the middle of a locked section leaves two smaller sections locked at each end of the originally locked section.
- If the calling process holds a lock on a file, that lock can be replaced by later calls to the **fcntl** subroutine.
- All locks associated with a file for a given process are removed when the process closes *any* file descriptor for that file.
- Locks are not inherited by a child process after a **fork** subroutine is run.

Note: Deadlocks due to file locks in a distributed system are not always detected. When such deadlocks can possibly occur, the programs requesting the locks should set time–out timers.

Locks can start and extend beyond the current end of a file but cannot be negative relative to the beginning of the file. A lock can be set to extend to the end of the file by setting the `l_len` field to 0. If such a lock also has the `l_start` and `l_whence` fields set to 0, the whole file is locked. The `l_len`, `l_start`, and `l_whence` locking fields are part of the **flock** structure.

Note: The following description applies to AIX Version 4.3 and later releases.

When an application locks a region of a file using the 32 bit locking interface (F_SETLK), and the last byte of the lock range includes MAX_OFF (2 Gb – 1), then the lock range for the unlock request will be extended to include MAX_END (2 ^ ^ 63 – 1).

Parameters

FileDescriptor	Specifies an open file descriptor obtained from a successful call to the **open**, **fcntl**, or **pipe** subroutine. File descriptors are small positive integers used (instead of file names) to identify files.
Argument	Specifies a variable whose value sets the function specified by the *Command* parameter. When dealing with file locks, the *Argument* parameter must be a pointer to the **FLOCK** structure.
Command	Specifies the operation performed by the **fcntl** subroutine. The **fcntl** subroutine can duplicate open file descriptors, set file–descriptor flags, set file descriptor locks, set process IDs, and close open file descriptors.

Duplicating File Descriptors

F_DUPFD Returns a new file descriptor as follows:

- Lowest–numbered available file descriptor greater than or equal to the *Argument* parameter
- Same object references as the original file
- Same file pointer as the original file (that is, both file descriptors share one file pointer if the object is a file)
- Same access mode (read, write, or read–write)
- Same file status flags (That is, both file descriptors share the same file status flags.)
- The **close–on–exec** flag (**FD_CLOEXEC** bit) associated with the new file descriptor is cleared

Setting File–Descriptor Flags

F_GETFD Gets the **close–on–exec** flag (**FD_CLOEXEC** bit) that is associated with the file descriptor specified by the *FileDescriptor* parameter. The *Argument* parameter is ignored. File descriptor flags are associated with a single file descriptor, and do not affect others associated with the same file.

F_SETFD Assigns the value of the *Argument* parameter to the **close–on–exec** flag (**FD_CLOEXEC** bit) that is associated with the *FileDescriptor* parameter. If the **FD_CLOEXEC** flag value is 0, the file remains open across any calls to **exec** subroutines; otherwise, the file will close upon the successful execution of an **exec** subroutine.

F_GETFL Gets the file–status flags and file–access modes for the open file description associated with the file descriptor specified by the *FileDescriptor* parameter. The open file description is set at the time the file is opened and applies only to those file descriptors associated with that particular call to the file. This open file descriptor does not affect other file descriptors that refer to the same file with different open file descriptions.

The file–status flags have the following values:

O_APPEND Set append mode.

O_NONBLOCK No delay.

The file–access modes have the following values:

O_RDONLY Open for reading only.

O_RDWR Open for reading and writing.

O_WRONLY Open for writing only.

The file access flags can be extracted from the return value using the **O_ACCMODE** mask, which is defined in the **fcntl.h** file.

F_SETFL Sets the file status flags from the corresponding bits specified by the *Argument* parameter. The file–status flags are set for the open file description associated with the file descriptor specified by the *FileDescriptor* parameter. The following flags may be set:

- **O_APPEND** or **FAPPEND**
- **O_NDELAY** or **FNDELAY**
- **O_NONBLOCK** or **FNONBLOCK**
- **O_SYNC** or **FSYNC**
- **FASYNC**

The **O_NDELAY** and **O_NONBLOCK** flags affect only operations against file descriptors derived from the same **open** subroutine. In BSD, these operations apply to all file descriptors that refer to the object.

Setting File Locks

F_GETLK Gets information on the first lock that blocks the lock described in the **flock** structure. The *Argument* parameter should be a pointer to a type **struct flock**, as defined in the **flock.h** file. The information retrieved by the **fcntl** subroutine overwrites the information in the **struct flock** pointed to by the *Argument* parameter. If no lock is found that would prevent this lock from being created, the structure is left unchanged, except for lock type (`l_type`) which is set to **F_UNLCK**.

F_SETLK Sets or clears a file–segment lock according to the lock description pointed to by the *Argument* parameter. The *Argument* parameter should be a pointer to a type **struct flock**, which is defined in the **flock.h** file. The **F_SETLK** option is used to establish read (or shared) locks (**F_RDLCK**), or write (or exclusive) locks (**F_WRLCK**), as well as to remove either type of lock (**F_UNLCK**). The lock types are defined by the **fcntl.h** file. If a shared or exclusive lock cannot be set, the **fcntl** subroutine returns immediately.

F_SETLKW Performs the same function as the **F_SETLK** option unless a read or write lock is blocked by existing locks, in which case the process sleeps until the section of the file is free to be locked. If a signal that is to be caught is received while the **fcntl** subroutine is waiting for a region, the **fcntl** subroutine is interrupted, returns a –1, sets the **errno** global variable to **EINTR**. The lock operation is not done.

Note: **F_GETLK64**, **F_SETLK64**, and **F_SETLKW64** apply to Version 4.2 and later releases.

F_GETLK64 Gets information on the first lock that blocks the lock described in the **flock64** structure. The *Argument* parameter should be a pointer to an object of the type **struct flock64**, as defined in the **flock.h** file. The information retrieved by the **fcntl** subroutine overwrites the information in the **struct flock64** pointed to by the *Argument* parameter. If no lock is found that would prevent this lock from being created, the structure is left unchanged, except for lock type (`l_type`) which is set to **F_UNLCK**.

F_SETLK64 Sets or clears a file–segment lock according to the lock description pointed to by the *Argument* parameter. The *Argument* parameter should be a pointer to a type **struct flock64**, which is defined in the **flock.h** file. The **F_SETLK** option is used to establish read (or shared) locks (**F_RDLCK**), or write (or exclusive) locks (**F_WRLCK**), as well as to remove either type of lock (**F_UNLCK**). The lock types are defined by the **fcntl.h** file. If a shared or exclusive lock cannot be set, the **fcntl** subroutine returns immediately.

F_SETLKW64 Performs the same function as the **F_SETLK** option unless a read or write lock is blocked by existing locks, in which case the process sleeps until the section of the file is free to be locked. If a signal that is to be caught is received while the **fcntl** subroutine is waiting for a region, the **fcntl** subroutine is interrupted, returns a –1, sets the **errno** global variable to **EINTR**. The lock operation is not done.

Setting Process ID

F_GETOWN Gets the process ID or process group currently receiving **SIGIO** and **SIGURG** signals. Process groups are returned as negative values.

F_SETOWN Sets the process or process group to receive **SIGIO** and **SIGURG** signals. Process groups are specified by supplying a negative *Argument* value. Otherwise, the *Argument* parameter is interpreted as a process ID.

Closing File Descriptors

F_CLOSEM Closes all file descriptors from *FileDescriptor* up to the number specified by the **OPEN_MAX** value.

Old Specifies an open file descriptor.

New Specifies an open file descriptor that is returned by the **dup2** subroutine.

Compatibility Interfaces

The lockfx Subroutine

The **fcntl** subroutine functions similar to the **lockfx** subroutine, when the *Command* parameter is **F_SETLK**, **F_SETLKW**, or **F_GETLK**, and when used in the following way:

fcntl (*FileDescriptor***,** *Command***,** *Argument***)**

is equivalent to:

lockfx (*FileDescriptor***,** *Command***,** *Argument***)**

The dup and dup2 Subroutines

The **fcntl** subroutine functions similar to the **dup** and **dup2** subroutines, when used in the following way:

```
dup (FileDescriptor)
```

is equivalent to:

```
fcntl (FileDescriptor, F_DUPFD, 0)
```

```
dup2 (Old, New)
```

is equivalent to:

```
close (New);
fcntl(Old, F_DUPFD, New)
```

The **dup** and **dup2** subroutines differ from the **fcntl** subroutine in the following ways:

- If the file descriptor specified by the *New* parameter is greater than or equal to **OPEN_MAX**, the **dup2** subroutine returns a –1 and sets the **errno** variable to **EBADF**.
- If the file descriptor specified by the *Old* parameter is valid and equal to the file descriptor specified by the *New* parameter, the **dup2** subroutine will return the file descriptor specified by the *New* parameter, without closing it.
- If the file descriptor specified by the *Old* parameter is not valid, the **dup2** subroutine will be unsuccessful and will not close the file descriptor specified by the *New* parameter.
- The value returned by the **dup** and **dup2** subroutines is equal to the *New* parameter upon successful completion; otherwise, the return value is –1.

Return Values

Upon successful completion, the value returned depends on the value of the *Command* parameter, as follows:

Command	Return Value
F_DUPFD	A new file descriptor
F_GETFD	The value of the flag (only the **FD_CLOEXEC** bit is defined)
F_SETFD	A value other than –1
F_GETFL	The value of file flags
F_SETFL	A value other than –1
F_GETOWN	The value of descriptor owner
F_SETOWN	A value other than –1
F_GETLK	A value other than –1
F_SETLK	A value other than –1
F_SETLKW	A value other than –1
F_CLOSEM	A value other than –1.

If the **fcntl** subroutine fails, a value of –1 is returned and the **errno** global variable is set to indicate the error.

Error Codes

The **fcntl** subroutine is unsuccessful if one or more of the following are true:

EACCES	The *Command* argument is **F_SETLK**; the type of lock is a shared or exclusive lock and the segment of a file to be locked is already exclusively–locked by another process, or the type is an exclusive lock and some portion of the segment of a file to be locked is already shared–locked or exclusive–locked by another process.
EBADF	The *FileDescriptor* parameter is not a valid open file descriptor.
EDEADLK	The *Command* argument is **F_SETLKW**; the lock is blocked by some lock from another process and putting the calling process to sleep, waiting for that lock to become free would cause a deadlock.

EMFILE	The *Command* parameter is **F_DUPFD**, and the maximum number of file descriptors are currently open (**OPEN_MAX**).
EINVAL	The *Command* parameter is **F_DUPFD**, and the *Argument* parameter is negative or greater than or equal to **OPEN_MAX**.
EINVAL	An illegal value was provided for the *Command* parameter.
EINVAL	An attempt was made to lock a fifo or pipe.
ESRCH	The value of the *Command* parameter is **F_SETOWN**, and the process ID specified as the *Argument* parameter is not in use.
EINTR	The *Command* parameter was **F_SETLKW** and the process received a signal while waiting to acquire the lock.
EOVERFLOW	The *Command* parameter was **F_GETLK** and the block lock could not be represented in the **flock** structure.

The **dup** and **dup2** subroutines fail if one or both of the following are true:

EBADF	The *Old* parameter specifies an invalid open file descriptor or the *New* parameter specifies a file descriptor that is out of range.
EMFILE	The number of file descriptors exceeds the **OPEN_MAX** value or there is no file descriptor above the value of the *New* parameter.

If NFS is installed on the system, the **fcntl** subroutine can fail if the following is true:

ETIMEDOUT	The connection timed out.

Implementation Specifics

These subroutines are part of Base Operating System (BOS) Runtime.

If *FileDescriptor* refers to a terminal device or socket, then asynchronous I/O facilities can be used. These facilities are normally enabled by using the **ioctl** subroutine with the **FIOASYNC**, **FIOSETOWN**, and **FIOGETOWN** commands. However, a BSD–compatible mechanism is also available if the application is linked with the **libbsd.a** library.

When using the **libbsd.a** library, asynchronous I/O is enabled by using the **F_SETFL** command with the **FASYNC** flag set in the *Argument* parameter. The **F_GETOWN** and **F_SETOWN** commands get the current asynchronous I/O owner and set the asynchronous I/O owner.

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

Related Information

The **close** subroutine, **execl**, **excecv**, **execle**, **execve**, **execlp**, **execvp**, or **exect** subroutines, **fork** subroutine, **ioctl** or **ioctlx** subroutine, **lockf** subroutine, **open**, **openx**, or **creat** subroutines, **read** subroutine, **write** subroutine.

Files, Directories, and File Systems for Programmers in *AIX General Programming Concepts : Writing and Debugging Programs*.

fdetach Subroutine

Purpose

Detaches STREAMS–based file from the file to which it was attached.

Library

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

Syntax

```
#include <stropts.h>
int fdetach(const char *path);
```

Parameters

path	Pathname of a file previous associated with a STREAMS–based object using the **fattach** subroutine.

Description

The **fdetach** subroutine detaches a STREAMS–based file from the file to which it was attached by a previous call to **fattach** subroutine. The *path* argument points to the pathname of the attached STREAMS file. The process must have appropriate privileges or be the owner of the file. A successful call to **fdetach** subroutine causes all pathnames that named the attached STREAMS file to again name the file to which the STREAMS file was attached. All subsequent operations on *path* will operate on the underlying file and not on the STREAMS file.

All open file descriptors established while the STREAMS file was attached to the file referenced by *path* will still refer to the STREAMS file after the **fdetach** subroutine has taken effect.

If there are no open file descriptors or other references to the STREAMS file, then a successful call to **fdetach** subroutine has the same effect as performing the last **close** subroutine on the attached file.

The **umount** command may be used to detach a file name if an | application exits before performing **fdetach** subroutine.

Return Value

0	Successful completion.
–1	Not successful and **errno** set to one of the following.

Errno Value

EACCES	Search permission is denied on a component of the path prefix.
EPERM	The effective user ID is not the owner of *path* and the process does not have appropriate privileges.
ENOTDIR	A component of the path prefix is not a directory.
ENOENT	A component of *path* parameter does not name an existing file or *path* is an empty string.
EINVAL	The *path* parameter names a file that is not currently attached.
ENAMETOOLONG	The size of *path* parameter exceeds {**PATH_MAX**}, or a component of *path* is longer than {**NAME_MAX**}.

ELOOP	Too many symbolic links were encountered in resolving the *path* parameter.
ENOMEM	Insufficient storage space is available.

Implementation Specifics

These subroutines are part of Base Operating System (BOS) Runtime.

Related Information

The **fattach** subroutine, **isastream** subroutine.

feof, ferror, clearerr, or fileno Macro

Purpose

Checks the status of a stream.

Library

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

Syntax

```
#include <stdio.h>

int feof (Stream)
FILE *Stream;

int ferror (Stream)
FILE *Stream;

void clearerr (Stream)
FILE *Stream;

int fileno (Stream)
FILE *Stream;
```

Description

The **feof** macro inquires about the end–of–file character (EOF). If EOF has previously been detected reading the input stream specified by the *Stream* parameter, a nonzero value is returned. Otherwise, a value of 0 is returned.

The **ferror** macro inquires about input or output errors. If an I/O error has previously occurred when reading from or writing to the stream specified by the *Stream* parameter, a nonzero value is returned. Otherwise, a value of 0 is returned.

The **clearerr** macro inquires about the status of a stream. The **clearerr** macro resets the error indicator and the EOF indicator to a value of 0 for the stream specified by the *Stream* parameter.

The **fileno** macro inquires about the status of a stream. The **fileno** macro returns the integer file descriptor associated with the stream pointed to by the *Stream* parameter. Otherwise a value of –1 is returned.

Parameters

Stream	Specifies the input or output stream.

Implementation Specifics

These subroutines are part of Base Operating System (BOS) Runtime.

Related Information

The **fopen**, **freopen**, or **fdopen** subroutine, **open** subroutine.

Input and Output Handling Programmer's Overview in *AIX General Programming Concepts : Writing and Debugging Programs*.

fetch_and_add Subroutine

Purpose

Updates a single word variable atomically.

Library

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

Syntax

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

int fetch_and_add (word_addr, value)
atomic_p word_addr;
int value;
```

Description

The **fetch_and_add** subroutine increments one word in a single atomic operation. This operation is useful when a counter variable is shared between several threads or processes. When updating such a counter variable, it is important to make sure that the fetch, update, and store operations occur atomically (are not interruptible). For example, consider the sequence of events which could occur if the operations were interruptible:

1. A process fetches the counter value and adds one to it.
2. A second process fetches the counter value, adds one, and stores it.
3. The first process stores its value.

The result of this is that the update made by the second process is lost.

Traditionally, atomic access to a shared variable would be controlled by a mechanism such as semaphores. Compared to such mechanisms, the **fetch_and_add** subroutine requires very little overhead, and provided that the counter variable fits in a single machine word, this subroutine provides a highly efficient way of performing this operation.

Note: The word containing the counter variable must be aligned on a full word boundary.

Parameters

word_addr	Specifies the address of the word variable to be incremented.
value	Specifies the value to be added to the word variable.

Return Values

This subroutine returns the original value of the word.

Implementation Specifics

This subroutine is part of the Base Operating System (BOS) Runtime.

Related Information

The **fetch_and_and** subroutine, **fetch_and_or** subroutine, **compare_and_swap** subroutine.

fetch_and_and or fetch_and_or Subroutine

Purpose

Sets or clears bits in a single word variable atomically.

Library

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

Syntax

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

uint fetch_and_and (word_addr, mask)
atomic_p word_addr;
int mask;

uint fetch_and_or (word_addr, mask)
atomic_p word_addr;
int mask;
```

Description

The **fetch_and_and** and **fetch_and_or** subroutines respectively clear and set bits in one word, according to a bit mask, in a single atomic operation. The **fetch_and_and** subroutine clears bits in the word which correspond to clear bits in the bit mask, and the **fetch_and_or** subroutine sets bits in the word which correspond to set bits in the bit mask.

These operations are useful when a variable containing bit flags is shared between several threads or processes. When updating such a variable, it is important that the fetch, bit clear or set, and store operations occur atomically (are not interruptible). For example, consider the sequence of events which could occur if the operations were interruptible:

1. A process fetches the flags variable and sets a bit in it.
2. A second process fetches the flags variable, sets a different bit, and stores it.
3. The first process stores its value.

The result is that the update made by the second process is lost.

Traditionally, atomic access to a shared variable would be controlled by a mechanism such as semaphores. Compared to such mechanisms, the **fetch_and_and** and **fetch_and_or** subroutines require very little overhead, and provided that the flags variable fits in a single machine word, they provide a highly efficient way of performing this operation.

Note: The word containing the flag bits must be aligned on a full word boundary.

Parameters

word_addr	Specifies the address of the single word variable whose bits are to be cleared or set.
mask	Specifies the bit mask which is to be applied to the single word variable.

Return Values

These subroutines return the original value of the word.

Implementation Specifics

These subroutines are part of the Base Operating System (BOS) Runtime.

Related Information

The **fetch_and_add** subroutine, **compare_and_swap** subroutine.

finfo or ffinfo Subroutine

Purpose

Returns file information.

Library

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

Syntax

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

int finfo(Path1, cmd, buffer, length)
const char *Path1;
int cmd;
void  *buffer;
int length;

int ffinfo (fd, cmd, buffer, length)
int fd;
int cmd;
void  *buffer;
int length;
```

Description

The **finfo** and **ffinfo** subroutines return specific file information for the specified file.

Parameters

Path1	Path name of a file system object to query.
fd	File descriptor for an open file to query.
cmd	Specifies the type of file information to be returned.
buffer	User supplied buffer which contains the file information upon successful return. /usr/include/sys/finfo.h describes the buffer.
length	Length of the query buffer.

Commands

F_PATHCONF
When the **F_PATHCONF** command is specified, a file's implementation information is returned.

Note: AIX provides another subroutine which retrieves file implementation characteristics, **pathconf** command. While the **finfo** and **ffinfo** subroutines can be used to retrieve file information, it is preferred that programs use the **pathconf** interface.

F_DIOCAP
When the **F_DIOCAP** command is specified, the file's direct 1/0 capability information is returned. The buffer supplied by the application is of type **struct diocapbuf ***.

Return Values

Upon successful completion, the finfo and ffinfo subroutines return a value of 0 and the user supplied buffer is correctly filled in with the file information requested. If the finfo or ffinfo subroutines were unsuccessful, a value of **–1** is returned and the global **errno** variable is set to indicate the error.

Error Codes

EACCES	Search permission is denied for a component of the path prefix.
EINVAL	If the length specified for the user buffer is greater than **MAX_FINFO_BUF**. If the command argument is not supported. If **F_DIOCAP** command is specified and the file object does not support Direct I/O.
ENAMETOOLONG	The length of the Path parameter string exceeds the **PATH_MAX** value.
ENOENT	The named file does not exist or the Path parameter points to an empty string.
ENOTDIR	A component of the path prefix is not a directory.
EBADF	File descriptor provided is not valid.

Implementation Specifics

These subroutines are part of Base Operating System (BOS) Runtime.

Related Information

The **pathconf** subroutine.

Subroutines Overview in AIX Version 4 General Programming Concepts: Writing and Debugging Programs.

flockfile, ftrylockfile, funlockfile Subroutine

Purpose

Provides for explicit application–level locking of stdio (FILE*) objects.

Library

Standard Library (**libc.a**)

Syntax

```
#include <stdio.h>
void flockfile (FILE * file)
int ftrylockfile (FILE * file)
void funlockfile (FILE * file)
```

Description

The **flockfile**, **ftrylockfile** and **funlockfile** functions provide for explicit application–level locking of stdio (**FILE***) objects. These functions can be used by a thread to delineate a sequence of I/O statements that are to be executed as a unit.

The **flockfile** function is used by a thread to acquire ownership of a (**FILE***) object.

The **ftrylockfile** function is used by a thread to acquire ownership of a (**FILE***) object if the object is available; **ftrylockfile** is a non–blocking version of **flockfile**. The **funlockfile** function is used to relinquish the ownership granted to the thread. The behavior is undefined if a thread other than the current owner calls the **funlockfile** function.

Logically, there is a lock count associated with each (**FILE***) object. This count is implicitly initialised to zero when the (**FILE***) object is created. The (**FILE***) object is unlocked when the count is zero. When the count is positive, a single thread owns the (**FILE***) object. When the **flockfile** function is called, if the count is zero or if the count is positive and the caller owns the (**FILE***) object, the count is incremented. Otherwise, the calling thread is suspended, waiting for the count to return to zero. Each call to **funlockfile** decrements the count. This allows matching calls to **flockfile** (or successful calls to **ftrylockfile**) and **funlockfile** to be nested.

All functions that reference (**FILE***) objects behave as if they use **flockfile** and **funlockfile** internally to obtain ownership of these (**FILE***) objects.

Return Values

None for **flockfile** and **funlockfile**. The function ftrylock returns zero for success and non–zero to indicate that the lock cannot be acquired.

Implementation Specifics

Realtime applications may encounter priority inversion when using FILE locks. The problem occurs when a high priority thread ”locks” a FILE that is about to be ”unlocked” by a low priority thread, but the low priority thread is preempted by a medium priority thread. This scenario leads to priority inversion; a high priority thread is blocked by lower priority threads for an unlimited period of time. During system design, realtime programmers must take into account the possibility of this kind of priority inversion. They can deal with it in a number of 7434 ways, such as by having critical sections that are guarded by FILE locks execute at a high priority, so that a thread cannot be preempted while executing in its critical section.

Future Directions

These subroutines are part of Base Operating System (BOS) suroutines.

Related Information

The **getc_unlocked** subroutine.

The **getchar_unlocked** subroutine.

The **putc_unlocked** subroutine.

The **putchar_unlocked** subroutine.

The **stdio.h** file.

floor, floorl, ceil, ceill, nearest, trunc, rint, itrunc, uitrunc, fmod, fmodl, fabs, or fabsl Subroutine

Purpose

The**floor** subroutine, **floorl** subroutine, **ceil** subroutine, **ceill** subroutine, **nearest** subroutine, **trunc** subroutine, and **rint** subroutine round floating–point numbers to floating–point integer values.

The **itrunc** subroutine and **uitrunc** subroutine round floating–point numbers to signed and unsigned integers, respectively.

The **fmod** subroutine and **fmodl** subroutine compute the modulo remainder. The **fabs** subroutine and **fabsl** subroutine compute the floating–point absolute value.

Libraries

IEEE Math Library (**libm.a**)
or System V Math Library (**libmsaa.a**)
Standard C Library (**libc.a**) (separate syntax follows)

Syntax

```
#include <math.h>

double floor (x)
double x;

long double floorl (x)
long double x;

double ceil (x)
double x;

long double ceill (x)
long double x;

double nearest (x)
double x;

double trunc (x)
double x;

double fmod (x,y)
double x, y;

long double fmodl (x)
long double x, y;

double fabs (x)
double x;

long double fabsl (x)
long double x;
```

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

```
#include <stdlib.h>
#include <limits.h>

double rint (x)
double x;

int itrunc (x)
double x;

unsigned int uitrunc (x)
double x;
```

Description

The **floor** subroutine and **floorl** subroutines return the largest floating–point integer value not greater than the *x* parameter.

The **ceil** subroutine and **ceill** subroutine return the smallest floating–point integer value not less than the *x* parameter.

The **nearest** subroutine returns the nearest floating–point integer value to the *x* parameter. If *x* lies exactly halfway between the two nearest floating–point integer values, an even floating–point integer is returned.

The **trunc** subroutine returns the nearest floating–point integer value to the *x* parameter in the direction of 0. This is equivalent to truncating off the fraction bits of the *x* parameter.

The **rint** subroutine returns one of the two nearest floating–point integer values to the *x* parameter. To determine which integer is returned, use the current floating–point rounding mode as described in the *IEEE Standard for Binary Floating–Point Arithmetic*.

If the current rounding mode is *round toward –INF*, **rint**(*x*) is identical to **floor**(*x*).

If the current rounding mode is *round toward +INF*, **rint**(*x*) is identical to **ceil**(*x*).

If the current rounding mode is *round to nearest*, **rint**(*x*) is identical to **nearest**(*x*).

If the current rounding mode is *round toward zero*, **rint**(*x*) is identical to **trunc**(*x*).

Note: The default floating–point rounding mode is *round to nearest.* All C main programs begin with the rounding mode set to *round to nearest.*

The **itrunc** subroutine returns the nearest signed integer to the *x* parameter in the direction of 0. This is equivalent to truncating the fraction bits from the *x* parameter and then converting *x* to a signed integer.

The **uitrunc** subroutine returns the nearest unsigned integer to the *x* parameter in the direction of 0. This action is equivalent to truncating off the fraction bits of the *x* parameter and then converting *x* to an unsigned integer.

The **fmod** subroutine and **fmodl** subroutine compute the modulo floating–point remainder of *x/y*. The **fmod** and **fmodl** subroutines return the value *x–iy* for a *i* such that if *y* is nonzero, the result has the same sign as *x* and magnitude less than the magnitude of *y.*

The **fabs** and **fabsl** subroutines return the absolute value of *x,* |*x*|.

Note: Compile any routine that uses subroutines from the **libm.a** library with the **–la** flag. To compile the `floor.c` file, for example, enter:

```
cc floor.c –lm
```

Parameters

x	Specifies a double–precision floating–point value. For the **floorl**, **ceill**, **fmodl**, and **fabsl** subroutines, specifies a long double–precision floating–point value.
y	Specifies a double–precision floating–point value. For the **floorl**, **ceill**, **fmodl**, and **fabsl** subroutines, specifies some long double–precision floating–point value.

Error Codes

The **itrunc** and **uitrunc** subroutines return the **INT_MAX** value if *x* is greater than or equal to the **INT_MAX** value and the **INT_MIN** value if *x* is equal to or less than the **INT_MIN** value. The **itrunc** subroutine returns the **INT_MIN** value if *x* is a Quiet NaN(not–a–number) or Silent NaN. The **uitrunc** subroutine returns 0 if *x* is a Quiet NaN or Silent NaN. (The **INT_MAX** and **INT_MIN** values are defined in the **limits.h** file.) The **uitrunc** subroutine **INT_MAX** if *x* is greater than **INT_MAX** and 0 if *x* is less than or equal 0.0

The **fmod** and **fmodl** subroutines for (x/0) return a Quiet NaN and set the **errno** global variable to a **EDOM** value.

Implementation Specifics

These subroutines are part of Base Operating System (BOS) Runtime.

The **itrunc**, **uitrunc**, **trunc**, **nearest**, and **rint** subroutines are not part of the ANSI C Library.

Files

float.h	Contains the ANSI C **FLT_ROUNDS** macro.

Related Information

The **fp_read_rnd** on **fp_swap_rnd** subroutine.

Subroutines Overview in *AIX General Programming Concepts : Writing and Debugging Programs*.

128–Bit long double Floating–Point Format in *AIX General Programming Concepts : Writing and Debugging Programs*.

IEEE Standard for Binary Floating–Point Arithmetic (ANSI/IEEE Standards 754–1985 and 854–1987).

fmtmsg Subroutine

Purpose

Display a message in the specified format on standard error, the console, or both.

Library

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

Syntax

```
#include <fmtmsg.h>

int fmtmsg (long Classification,
const char *Label,
int Severity,
cont char *Text;
cont char *Action,
cont char *Tag)
```

Description

The **fmtmsg** subroutine can be used to display messages in a specified format instead of the traditional **printf** subroutine interface.

Base on a message's classification component, the **fmtmsg** subroutine either writes a formatted message to standard error, the console, or both.

A formatted message consists of up to five parameters. The *Classification* parameter is not part of a message displayed to the user, but defines the source of the message and directs the display of the formatted message.

Parameters

Classification
Contains identifiers from the following groups of major classifications and subclassifications. Any one identifier from a subclass may be used in combination with a single identifier from a different subclass. Two or more identifiers from the same subclass should not be used together, with the exception of identifiers from the display subclass. (Both display subclass identifiers may be used so that messages can be displayed to both standard error and system console).

major classifications
Identifies the source of the condition. Identifiers are: **MM_HARD** (hardware), **MM_SOFT** (software), and **MM_FIRM** (firmware).

message source subclassifications
Identifies the type of software in which the problem is detected. Identifiers are: **MM_APPL** (application), **MM_UTIL** (utility), and **MM_OPSYS** (operating system).

display subclassification
Indicates where the message is to be displayed. Identifiers are: **MM_PRINT** to display the message on the standard error stream, **MM_CONSOLE** to display the message on the system console. One or both identifiers may be used.

status subclassifications

Indicates whether the application will recover from the condition. Identifiers are:**MM_RECOVER** (recoverable) and **MM_RECOV** (non–recoverable).

An additional identifier, **MM_NULLMC**, identifies that no classification component is supplied for the message.

Label
: Identifies the source to the message. The format is two fields separated by a colon. The first field is up to 10 bytes, the second field is up to 14 bytes.

Severity

Text
: Describes the error condition that produced the message. The character string is not limited to a specific size. If the character string is null then a message will be issued stating that no text has been provided.

Action
: Describes the first step to be taken in the error–recovery process. The **fmtmsg** subroutine precedes the action string with the prefix: `TO FIX:`. The *Action* string is not limited to a specific size.

Tag
: An identifier which references online documentation for the message. Suggested usage is that *tag* includes the *Label* and a unique identifying number. A sample *tag* is `UX:cat:146`.

Environment Variables

The **MSGVERB** (message verbosity) environment variable controls the behavior of the **fmtmsg** subroutine.

MSGVERB tells the **fmtmsg** subroutine which message components it is to select when writing messages to standard error. The value of **MSGVERB** is a colon–separated list of optional keywords. **MSGVERB** can be set as follows:

```
MSGVERB=[keyword[:keyword[:...]]]
export  MSGVERB
```

Valid keywords are: *Label, Severity, Text, Action,* and *Tag*. If **MSGVERB** contains a keyword for a component and the component's value is not the component's null value, **fmtmsg** subroutine includes that component in the message when writing the message to standard error. If **MSGVERB** does not include a keyword for a message component, that component is not included in the display of the message. The keywords may appear in any order. If **MSGVERB** is not defined, if its value is the null string, if its value is not of the correct format, of if it contains keywords other than the valid ones listed previously, the **fmtmsg** subroutine selects all components.

MSGVERB affects only which components are selected for display to standard error. All message components are included in console messages.

Application Usage

One or more message components may be systematically omitted from messages generated by an application by using the null value of the parameter for that component. The table below indicates the null values and identifiers for **fmtmsg** subroutine parameters.

Parameter	**Type**	**Null–Value**	**Identifier**
label	char*	(char*)0	MM_NULLLBL
severity	int	0	MM_NULLSEV
class	long	0L	MM_NULLMC
text	char*	(char*)0	MM_NULLTXT

action	char*	(char*)0	MM_NULLACT
tag	char*	(char*)0	MM_NULLTAG

Another means of systematically omitting a component is by omitting the component keywords when defining the MSGVERB environment variable.

Return Values

The exit codes for the **fmtmsg** subroutine are the following:

MM_OK	The function succeeded.
MM_NOTOK	The function failed completely.
MM_MOMSG	The function was unable to generate a message on standard error.
MM_NOCON	The function was unable to generate a console message.

Examples

1. The following example of the **fmtmsg** subroutine:

   ```
   fmtmsg(MM_PRINT, ”UX:cat”, MM_ERROR, ”illegal option”,
   ”refer tp cat in user’s reference manual”, ”UX:cat:001”)
   ```

   produces a complete message in the specified message format:

   ```
   UX:cat ERROR: illegal option
   TO FIX: refer to cat in user’s reference manual UX:cat:001
   ```

2. When the environment variable MSGVERB is set as follows:

   ```
   MSGVERB=severity:text:action
   ```

   and the Example 1 is used, the **fmtmsg** subroutine produces:

   ```
   ERROR: illegal option
   TO FIX: refer to cat in user’s reference manual UX:cat:001
   ```

Implementation Specifics

This subroutine is part of Base Operating System (BOS) Runtime.

Related Information

The **printf** routine.

fnmatch Subroutine

Purpose

Matches file name patterns.

Library

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

Syntax

```
#include <fnmatch.h>

int fnmatch (Pattern, String, Flags);
int Flags;
const char *Pattern, *String;
```

Description

The **fnmatch** subroutine checks the string specified by the *String* parameter to see if it matches the pattern specified by the *Pattern* parameter.

The **fnmatch** subroutine can be used by an application or command that needs to read a dictionary and apply a pattern against each entry; the **find** command is an example of this. It can also be used by the **pax** command to process its *Pattern* variables, or by applications that need to match strings in a similar manner.

Parameters

Pattern
: Contains the pattern to which the *String* parameter is to be compared. The *Pattern* parameter can include the following special characters:

  ***** (asterisk)
  : Matches zero, one, or more characters.

  ? (question mark)
  : Matches any single character, but will not match 0 (zero) characters.

  [ ] (brackets)
  : Matches any one of the characters enclosed within the brackets. If a pair of characters separated by a dash are contained within the brackets, the pattern matches any character that lexically falls between the two characters in the current locale.

String
: Contains the string to be compared against the *Pattern* parameter.

Flags
: Contains a bit flag specifying the configurable attributes of the comparison to be performed by the **fnmatch** subroutine.

  The *Flags* parameter modifies the interpretation of the *Pattern* and *String* parameters. It is the bitwise inclusive OR of zero or more of the following flags (defined in the **fnmatch.h** file):

  FNM_PATHNAME
  : Indicates the / (slash) in the *String* parameter matches a / in the *Pattern* parameter.

  FNM_PERIOD
  : Indicates a leading period in the *String* parameter matches a period in the *Pattern* parameter.

  FNM_NOESCAPE
  : Enables quoting of special characters using the \ (backslash).

If the **FNM_ PATHNAME** flag is set in the *Flags* parameter, a / (slash) in the *String* parameter is explicitly matched by a / in the *Pattern* parameter. It is not matched by either the * (asterisk) or ? (question–mark) special characters, nor by a bracket expression. If the **FNM_PATHNAME** flag is not set, the / is treated as an ordinary character.

If the **FNM_PERIOD** flag is set in the *Flags* parameter, then a leading period in the *String* parameter only matches a period in the *Pattern* parameter; it is not matched by either the asterisk or question–mark special characters, nor by a bracket expression. The setting of the **FNM_PATHNAME** flag determines a period to be leading, according to the following rules:

- If the **FNM_PATHNAME** flag is set, a . (period) is leading only if it is the first character in the *String* parameter or if it immediately follows a /.
- If the **FNM_PATHNAME** flag is not set, a . (period) is leading only if it is the first character of the *String* parameter. If **FNM_PERIOD** is not set, no special restrictions are placed on matching a period.

If the **FNM_NOESCAPE** flag is not set in the *Flags* parameter, a \ (backslash) character in the *Pattern* parameter, followed by any other character, will match that second character in the *String* parameter. For example, \\ will match a backslash in the *String* parameter. If the **FNM_NOESCAPE** flag is set, a \ (backslash) will be treated as an ordinary character.

Return Values

If the value in the *String* parameter matches the pattern specified by the *Pattern* parameter, the **fnmatch** subroutine returns 0. If there is no match, the **fnmatch** subroutine returns the **FNM_NOMATCH** constant, which is defined in the **fnmatch.h** file. If an error occurs, the **fnmatch** subroutine returns a nonzero value.

Implementation Specifics

These subroutines are part of Base Operating System (BOS) Runtime.

Files

/usr/include/fnmatch.h	Contains system–defined flags and constants.

Related Information

The **glob** subroutine, **globfree** subroutine, **regcomp** subroutine, **regfree** subroutine, **regerror** subroutine, **regexec** subroutine.

The **find** command, **pax** command.

Files, Directories, and File Systems for Programmers and Understanding Internationalized Regular Expression Subroutines Ln *AIX General Programming Concepts : Writing and Debugging Programs*

fopen, fopen64, freopen, freopen64 or fdopen Subroutine

Purpose

Opens a stream.

Library

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

Syntax

```
#include <stdio.h>
FILE  *fopen (Path, Type)
const char *Path, *Type;

FILE  *fopen64 (Path, Type)
char  *Path, *Type;

FILE  *freopen (Path, Type, Stream)
const char *Path, *Type;
FILE  *Stream;

FILE  *freopen64 (Path, Type, Stream)
char  *Path, *Type;
FILE  *Stream;

FILE  *fdopen (FileDescriptor, Type)
int   FileDescriptor;
const char *Type;
```

Description

The **fopen** and **fopen64** subroutines open the file named by the *Path* parameter and associate a stream with it and return a pointer to the **FILE** structure of this stream.

When you open a file for update, you can perform both input and output operations on the resulting stream. However, an output operation cannot be directly followed by an input operation without an intervening **fflush** subroutine call or a file positioning operation (**fseek**, **fseeko**, **fseeko64**, **fsetpos**, **fsetpos64** or **rewind** subroutine). Also, an input operation cannot be directly followed by an output operation without an intervening flush or file positioning operation, unless the input operation encounters the end of the file.

When you open a file for appending (that is, when the *Type* parameter is set to **a**), it is impossible to overwrite information already in the file.

If two separate processes open the same file for append, each process can write freely to the file without destroying the output being written by the other. The output from the two processes is intermixed in the order in which it is written to the file.

Note: If the data is buffered, it is not actually written until it is flushed.

The **freopen** and **freopen64** subroutines first attempt to flush the stream and close any file descriptor associated with the *Stream* parameter. Failure to flush the stream or close the file descriptor is ignored.

The **freopen** and **freopen64** subroutines substitute the named file in place of the open stream. The original stream is closed regardless of whether the subsequent open succeeds. The **freopen** and **freopen64** subroutines returns a pointer to the **FILE** structure associated with the *Stream* parameter. The **freopen** and **freopen64** subroutines is typically used to attach the pre–opened streams associated with standard input (**stdin**), standard output (**stdout**), and standard error (**stderr**) streams to other files.

The **fdopen** subroutine associates a stream with a file descriptor obtained from an **openx** subroutine, **dup** subroutine, **creat** subroutine, or **pipe** subroutine. These subroutines open files but do not return pointers to **FILE** structures. Many of the standard I/O package subroutines require pointers to **FILE** structures.

The *Type* parameter for the **fdopen** subroutine specifies the mode of the stream, such as **r** to open a file for reading, or **a** to open a file for appending (writing at the end of the file). The mode value of the *Type* parameter specified with the **fdopen** subroutine must agree with the mode of the file specified when the file was originally opened or created.

The largest value that can be represented correctly in an object of type off_t will be established as the offset maximum in the open file description.

Parameters

Path	Points to a character string that contains the name of the file to be opened.
Type	Points to a character string that has one of the following values:

r	Opens a text file for reading.
w	Creates a new text file for writing, or opens and truncates a file to 0 length.
a	Appends (opens a text file for writing at the end of the file, or creates a file for writing).
rb	Opens a binary file for reading.
wb	Creates a binary file for writing, or opens and truncates a file to 0.
ab	Appends (opens a binary file for writing at the end of the file, or creates a file for writing).
r+	Opens a file for update (reading and writing).
w+	Truncates or creates a file for update.
a+	Appends (opens a text file for writing at end of file, or creates a file for writing).
r+b , **rb+**	Opens a binary file for update (reading and writing).
w+b , **wb+**	Creates a binary file for update, or opens and truncates a file to 0 length.
a+b , **ab+**	Appends (opens a binary file for update, writing at the end of the file, or creates a file for writing).

Note: The operating system does not distinguish between text and binary files. The **b** value in the *Type* parameter value is ignored.

Stream	Specifies the input stream.
FileDescriptor	Specifies a valid open file descriptor.

Return Values

If the **fdopen**, **fopen**, **fopen64**, **freopen** or **freopen64** subroutine is unsuccessful, a null pointer is returned and the **errno** global variable is set to indicate the error.

Error Codes

The **fopen**, **fopen64**, **freopen** and **freopen64** subroutines are unsuccessful if the following is true:

EACCES	Search permission is denied on a component of the path prefix, the file exists and the permissions specified by the mode are denied, or the file does not exist and write permission is denied for the parent directory of the file to be created.
ELOOP	Too many symbolic links were encountered in resolving path.
EINTR	A signal was received during the process.
EISDIR	The named file is a directory and the process does not have write access to it.
ENAMETOOLONG	The length of the filename exceeds **PATH_MAX** or a pathname component is longer than **NAME_MAX**.
ENFILE	The maximum number of files allowed are currently open.
ENOENT	The named file does not exist or the *File Descriptor* parameter points to an empty string.
ENOSPC	The file is not yet created and the directory or file system to contain the new file cannot be expanded.
ENOTDIR	A component of the path prefix is not a directory.
ENXIO	The named file is a character– or block–special file, and the device associated with this special file does not exist.
EOVERFLOW	The named file is a regular file and the size of the file cannot be represented correctly in an object of type off_t.
EROFS	The named file resides on a read–only file system and does not have write access.
ETXTBSY	The file is a pure–procedure (shared–text) file that is being executed and the process does not have write access.

The **fdopen**, **fopen**, **fopen64**, **freopen** and **freopen64** subroutines are unsuccessful if the following is true:

EINVAL	The value of the *Type* argument is not valid.
EINVAL	The value of the *mode* argument is not valid.
EMFILE	**FOPEN_MAX** streams are currently open in the calling process.
EMFILE	**STREAM_MAX** streams are currently open in the calling process.
ENAMETOOLONG	Pathname resolution of a symbolic link produced an intermediate result whose length exceeds **PATH_MAX**.
ENOMEM	Insufficient storage space is available.

The **freopen** and **fopen** subroutines are unsuccessful if the following is true:

EOVERFLOW	The named file is a size larger than 2 Gigabytes.

The **fdopen** subroutine is unsuccessful if the following is true:

EBADF	The value of the *File Descriptor* parameter is not valid.

Implementation Specifics

These subroutines are part of Base Operating System (BOS) Runtime.

POSIX

w	Truncates to 0 length or creates text file for writing.
w+	Truncates to 0 length or creates text file for update.

a	Opens or creates text file for writing at end of file.
a+	Opens or creates text file for update, writing at end of file.

SAA

At least eight streams, including three standard text streams, can open simultaneously. Both binary and text modes are supported.

Related Information

The **fclose** or **fflush** subroutine, **fseek**, **fseeko**, **fseeko64**, **rewind**, **ftell**, **ftello**, **ftello64**, **fgetpos**, **fgetpos64** or **fsetpos** subroutine, **open**, **open64**, **openx**, or **creat** subroutine, **setbuf**, **setvbuf**, **setbuffer**, or **setlinebuf** subroutine.

The Input and Output Handling Programmer's Overview in *AIX General Programming Concepts : Writing and Debugging Programs*.

fork or vfork Subroutine

fork, f_fork, or vfork Subroutine

Purpose

Creates a new process.

Libraries

fork and **vfork**: Standard C Library (**libc.a**)

fork, **f_fork**, and **vfork**: Standard C Library (**libc.a**)

Syntax

```
#include <unistd.h>

pid_t fork(void)

pid_t f_fork(void)

int vfork(void)
```

Description

The **fork** subroutine creates a new process. The new process (child process) is an almost exact copy of the calling process (parent process). The child process inherits the following attributes from the parent process:

- Environment
- Close–on–exec flags (described in the **exec** subroutine)
- Signal handling settings (such as the **SIG_DFL** value, the **SIG_IGN** value, and the *Function Address* parameter)
- Set user ID mode bit
- Set group ID mode bit
- Profiling on and off status
- Nice value
- All attached shared libraries
- Process group ID
- **tty** group ID (described in the **exit**, **atexit**, or **_exit** subroutine, **signal** subroutine, and **raise** subroutine)
- Current directory
- Root directory
- File–mode creation mask (described in the **umask** subroutine)
- File size limit (described in the **ulimit** subroutine)
- Attached shared memory segments (described in the **shmat** subroutine)
- Attached mapped file segments (described in the **shmat** subroutine)
- Debugger process ID and multiprocess flag if the parent process has multiprocess debugging enabled (described in the **ptrace** subroutine).

The child process differs from the parent process in the following ways:

- The child process has only one user thread; it is the one that called the **fork** subroutine.

- The child process has a unique process ID.
- The child process ID does not match any active process group ID.
- The child process has a different parent process ID.
- The child process has its own copy of the file descriptors for the parent process. However, each file descriptor of the child process shares a common file pointer with the corresponding file descriptor of the parent process.
- All **semadj** values are cleared. For information about **semadj** values, see the **semop** subroutine.
- Process locks, text locks, and data locks are not inherited by the child process. For information about locks, see the **plock** subroutine.
- If multiprocess debugging is turned on, the **trace** flags are inherited from the parent; otherwise, the **trace** flags are reset. For information about request 0, see the **ptrace** subroutine.
- The child process **utime**, **stime**, **cutime**, and **cstime** subroutines are set to 0. (For more information, see the **getrusage**, **times**, and **vtimes** subroutines.)
- Any pending alarms are cleared in the child process. (For more information, see the **incinterval**, **setitimer**, and **alarm** subroutines.)
- The set of signals pending for the child process is initialized to the empty set.
- The child process can have its own copy of the message catalogue for the parent process.
- The set of signals pending for the child process is initialized as an empty set.

  Attention: If you are using the **fork** or **vfork** subroutines with an Enhanced X-Windows, X Toolkit, or Motif application, open a separate display connection (socket) for the forked process. If the child process uses the same display connection as the parent, the X Server will not be able to interpret the resulting data. See the Implementation Specifics section for more information.

The **f_fork** subroutine is similar to **fork**, except for:

- It is required that the child process calls one of the **exec** functions immediately after it is created. Since the **fork** handlers are never called, the application data, mutexes and the locks are all undefined in the child process.

Return Values

Upon successful completion, the **fork** subroutine returns a value of 0 to the child process and returns the process ID of the child process to the parent process. Otherwise, a value of –1 is returned to the parent process, no child process is created, and the **errno** global variable is set to indicate the error.

Error Codes

The **fork** subroutine is unsuccessful if one or more of the following are true:

EAGAIN	Exceeds the limit on the total number of processes running either systemwide or by a single user, or the system does not have the resources necessary to create another process.
ENOMEM	Not enough space exists for this process.

Implementation Specifics

These subroutines are part of Base Operating System (BOS) Runtime.

The **vfork** subroutine is supported as a compatibility interface for older Berkeley Software Distribution (BSD) system programs and can be used by compiling with the Berkeley Compatibility Library (**libbsd.a**).

In the Version 4 of the operating system, the parent process does not have to wait until the child either exits or executes, as it does in BSD systems. The child process is given a new address space, as in the **fork** subroutine. The child process does not share any parent address space.

> **Attention:** When using the **fork** or **vfork** subroutines with an Enhanced X-Windows, X Toolkit, or Motif application, a separate display connection (socket) should be opened for the forked process. Use the **XOpenDisplay** or the **XtOpenDisplay** subroutines to open the separate connection. The child process should never use the same display connection as the parent. Display connections are embodied with sockets, and sockets are inherited by the child process. Any attempt to have multiple processes writing to the same display connection results in the random interleaving of X protocol packets at the word level. The resulting data written to the socket will not be valid or undefined X protocol packets, and the X Server will not be able to interpret it.

> **Attention:** Although the **fork** and **vfork** subroutine may be used with Graphics Library applications, the child process must not make any additional Graphics Library subroutine calls. The child application inherits some, but not all of the graphics hardware resources of the parent. Drawing by the child process may hang the graphics adapter, the Enhanced X Server, or may cause unpredictable results and place the system into an unpredictable state.

Note: Some Graphics Library subroutines, such as the **winopen** subroutine, implicitly create an X display connection. This connection may be obtained with the **getXdpy** subroutine.

For additional information, see the **/usr/lpp/GL/README** file.

Related Information

The **alarm** subroutine, **bindprocessor** subroutine, **exec** subroutine, **exit**, **atexit**, or **_exit** subroutine, **getrusage** or **times** subroutine, **getXdpy** subroutine, **incinterval** subroutine, **nice** subroutine, **plock** subroutine, **pthread_atfork** subroutine, **ptrace** subroutine, **raise** subroutine, **semop** subroutine, **setitimer** subroutine, **shmat** subroutine, **setpriority** or **getpriority** subroutine, **sigaction**, **sigvec**, or **signal** subroutine, **ulimit** subroutine, **umask** subroutine, **wait**, **waitpid**, or **wait3** subroutine, **winopen** subroutine, **XOpenDisplay** subroutine, **XtOpenDisplay** subroutine.

Subroutines Overview in *AIX General Programming Concepts : Writing and Debugging Programs*.

Process Duplication and Termination in *AIX General Programming Concepts : Writing and Debugging Programs*LK provides more information about forking a multi–threaded process.

fp_any_enable, fp_is_enabled, fp_enable_all, fp_enable, fp_disable_all, or fp_disable Subroutine

Purpose

These subroutines allow operations on the floating–point trap control.

Library

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

Syntax

```
#include <fptrap.h>

int fp_any_enable()
int fp_is_enabled(Mask)
fptrap_t Mask;

void fp_enable_all()
void fp_enable(Mask)
fptrap_t Mask;

void fp_disable_all()
void fp_disable(Mask)
fptrap_t Mask;
```

Description

Floating point traps must be enabled before traps can be generated. These subroutines aid in manipulating floating–point traps and identifying the trap state and type.

In order to take traps on floating point exceptions, the **fp_trap** subroutine must first be called to put the process in serialized state, and the **fp_enable** subroutine or **fp_enable_all** subroutine must be called to enable the appropriate traps.

The header file **fptrap.h** defines the following names for the individual bits in the floating–point trap control:

TRP_INVALID	Invalid Operation Summary
TRP_DIV_BY_ZERO	Divide by Zero
TRP_OVERFLOW	Overflow
TRP_UNDERFLOW	Underflow
TRP_INEXACT	Inexact Result

Parameters

Mask	A 32–bit pattern that identifies floating–point traps.

Return Values

The **fp_any_enable** subroutine returns 1 if any floating–point traps are enabled. Otherwise, 0 is returned.

The **fp_is_enabled** subroutine returns 1 if the floating–point traps specified by the *Mask* parameter are enabled. Otherwise, 0 is returned.

The **fp_enable_all** subroutine enables all floating–point traps.

The **fp_enable** subroutine enables all floating–point traps specified by the *Mask* parameter.

The **fp_disable_all** subroutine disables all floating–point traps.

The **fp_disable** subroutine disables all floating–point traps specified by the *Mask* parameter.

Implementation Specifics

These subroutines are part of Base Operating System (BOS) Runtime.

Related Information

The **fp_clr_flag**, **fp_set_flag**, **fp_read_flag**, **fp_swap_flag** subroutine, **fp_invalid_op**, **fp_divbyzero**, **fp_overflow**, **fp_underflow**, **fp_inexact**, **fp_any_xcp** subroutines, **fp_iop_snan**, **fp_iop_infsinf**, **fp_iop_infdinf**, **fp_iop_zrdzr**, **fp_iop_infmzr**, **fp_iop_invcmp** subroutines, **fp_read_rnd**, and **fp_swap_rnd** subroutines, **fp_trap** subroutine.

Floating–Point Processor Overview in *Hardware Technical Information-General Architectures*.

The *IEEE Standard for Binary Floating–Point Arithmetic* (ANSI/IEEE Standards 754–1985 and 854–1987).

Subroutines Overview in *AIX General Programming Concepts : Writing and Debugging Programs*.

fp_clr_flag, fp_set_flag, fp_read_flag, or fp_swap_flag Subroutine

Purpose

Allows operations on the floating–point exception flags.

Library

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

Syntax

```
#include <float.h>
#include <fpxcp.h>

void fp_clr_flag(Mask)
fpflag_t Mask;

void fp_set_flag(Mask)
fpflag_t Mask;

fpflag_t fp_read_flag( )

fpflag_t fp_swap_flag(Mask)
fpflag_t Mask;
```

Description

These subroutines aid in determining both when an exception has occurred and the exception type. These subroutines can be called explicitly around blocks of code that may cause a floating–point exception.

According to the *IEEE Standard for Binary Floating–Point Arithmetic*, the following types of floating–point operations must be signaled when detected in a floating–point operation:

- Invalid operation
- Division by zero
- Overflow
- Underflow
- Inexact

An invalid operation occurs when the result cannot be represented (for example, a **sqrt** operation on a number less than 0).

The *IEEE Standard for Binary Floating–Point Arithmetic* states: "For each type of exception, the implementation shall provide a status flag that shall be set on any occurrence of the corresponding exception when no corresponding trap occurs. It shall be reset only at the user's request. The user shall be able to test and to alter the status flags individually, and should further be able to save and restore all five at one time."

Floating–point operations can set flags in the floating–point exception status but cannot clear them. Users can clear a flag in the floating–point exception status using an explicit software action such as the **fp_swap_flag (**0**)** subroutine.

The **fpxcp.h** file defines the following names for the flags indicating floating–point exception status:

FP_INVALID	Invalid operation summary
FP_OVERFLOW	Overflow
FP_UNDERFLOW	Underflow
FP_DIV_BY_ZERO	Division by 0
FP_INEXACT	Inexact result

In addition to these flags, the operating system supports additional information about the cause of an invalid operation exception. The following flags also indicate floating–point exception status and defined in the **fpxcp.h** file. The flag number for each exception type varies, but the mnemonics are the same for all ports. The following invalid operation detail flags are not required for conformance to the IEEE floating–point exceptions standard:

FP_INV_SNAN	Signaling NaN
FP_INV_ISI	INF – INF
FP_INV_IDI	INF / INF
FP_INV_ZDZ	0 / 0
FP_INV_IMZ	INF x 0
FP_INV_CMP	Unordered compare
FP_INV_SQRT	Square root of a negative number
FP_INV_CVI	Conversion to integer error
FP_INV_VXSOFT	Software request

Parameters

Mask	A 32–bit pattern that identifies floating–point exception flags.

Return Values

The **fp_clr_flag** subroutine resets the exception status flags defined by the *Mask* parameter to 0 (false). The remaining flags in the exception status are unchanged.

The **fp_set_flag** subroutine sets the exception status flags defined by the *Mask* parameter to 1 (true). The remaining flags in the exception status are unchanged.

The **fp_read_flag** subroutine returns the current floating–point exception status. The flags in the returned exception status can be tested using the flag definitions above. You can test individual flags or sets of flags.

The **fp_swap_flag** subroutine writes the *Mask* parameter into the floating–point status and returns the floating–point exception status from before the write.

Users set or reset multiple exception flags using **fp_set_flag** and **fp_clr_flag** by ANDing or ORing definitions for individual flags. For example, the following resets both the overflow and inexact flags:

```
fp_clr_flag (FP_OVERFLOW | FP_INEXACT)
```

Implementation Specifics

These subroutines are part of Base Operating System (BOS) Runtime.

Related Information

The **fp_any_enable**, **fp_is_enabled**, **fp_enable_all**, **fp_enable**, **fp_disable**, or **fp_disable_all** subroutine, **fp_any_xcp**, **fp_divbyzero**, **fp_inexact**, **fp_invalid_op**, **fp_overflow**, **fp_underflow** subroutines, **fp_iop_infdinf**, **fp_iop_infmzr**, **fp_iop_infsinf**, **fp_iop_invcmp**, **fp_iop_snan**, or **fp_iop_zrdzr** subroutines, **fp_read_rnd** or **fp_swap_rnd** subroutine.

IEEE Standard for Binary Floating–Point Arithmetic (ANSI/IEEE Standards 754–1985 and 854–1987) describes the IEEE floating–point exceptions.

Floating–Point Exceptions Overview and Subroutines Overview in *AIX General Programming Concepts : Writing and Debugging Programs*.

fp_cpusync Subroutine

Purpose

Queries or changes the floating–point exception enable (FE) bit in the Machine Status register (MSR).

Note: This subroutine has been replaced by the **fp_trapstate** subroutine. The **fp_cpusync** subroutine is supported for compatibility, but the **fp_trapstate** subroutine should be used for development.

Library

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

Syntax

```
#include <fptrap.h>

int fp_cpusync (Flag);
int Flag;
```

Description

The **fp_cpusync** subroutine is a service routine used to query, set, or reset the Machine Status Register (MSR) floating–point exception enable (FE) bit. The MSR FE bit determines whether a processor runs in pipeline or serial mode. Floating–point traps can only be generated by the hardware when the processor is in synchronous mode.

The **fp_cpusync** subroutine changes only the MSR FE bit. It is a service routine for use in developing custom floating–point exception–handling software. If you are using the **fp_enable** or **fp_enable_all** subroutine or the **fp_sh_trap_info** or **fp_sh_set_stat** subroutine, you must use the **fp_trap** subroutine to place the process in serial mode.

Parameters

Flag Specifies to query or modify the MSR FE bit:

FP_SYNC_OFF	Sets the FE bit in the MSR to Off, which disables floating–point exception processing immediately.
FP_SYNC_ON	Sets the FE bit in the MSR to On, which enables floating–exception processing for the next floating–point operation.
FP_SYNC_QUERY	Returns the current state of the process (either **FP_SYNC_ON** or **FP_SYNC_OFF**) without modifying it.

If called with any other value, the **fp_cpusync** subroutine returns **FP_SYNC_ERROR**.

Return Values

If called with the **FP_SYNC_OFF** or **FP_SYNC_ON** flag, the **fp_cpusync** subroutine returns a value indicating which flag was in the previous state of the process.

If called with the **FP_SYNC _QUERY** flag, the **fp_cpusync** subroutine returns a value indicating the current state of the process, either the **FP_SYNC_OFF** or **FP_SYNC_ON** flag.

Error Codes

If the **fp_cpusync** subroutine is called with an invalid parameter, the subroutine returns **FP_SYNC_ERROR**. No other errors are reported.

Related Information

The **fp_any_enable**, **fp_is_enabled**, **fp_enable_all**, **fp_enable**, **fp_disable_all**, or **fp_disable** subroutine, **fp_clr_flag**, **fpset_flag**, **fp_read_flag**, or **fp_swap_flag** subroutine, **sigaction**, **sigvec**, or **signal** subroutine.

Floating–Point Processor Overview in *Hardware Technical Information-General Architectures*.

Floating–Point Exceptions Overview in *AIX General Programming Concepts : Writing and Debugging Programs*.

fp_flush_imprecise Subroutine

Purpose

Forces imprecise signal delivery.

Library

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

Syntax

```
#include <fptrap.h>

void fp_flush_imprecise ()
```

Description

The **fp_flush_imprecise** subroutine forces any imprecise interrupts to be reported. To ensure that no signals are lost when a program voluntarily exits, use this subroutine in combination with the **atexit** subroutine.

Example

The following example illustrates using the **atexit** subroutine to run the **fp_flush_imprecise** subroutine before a program exits:

```
#include <fptrap.h>
#include <stdlib.h>
#include <stdio.h>
  if (0!=atexit(fp_flush_imprecise))
           puts ("Failure in atexit(fp_flush_imprecise) ");
```

Implementation Specifics

This subroutine is part of Base Operating System (BOS) Runtime.

Related Information

The **atexit** subroutine, **fp_any_enable**, **fp_is_enabled**, **fp_enable_all**, **fp_enable**, **fp_disable_all**, or **fp_disable** subroutine, **fp_clr_flag**, **fp_read_flag**, **fp_swap_flag**, or **fpset_flag** subroutine, **fp_cpusync** subroutine, **fp_trap** subroutine **sigaction** subroutine.

Floating–Point Exceptions Overview in *AIX General Programming Concepts : Writing and Debugging Programs*.

fp_invalid_op, fp_divbyzero, fp_overflow, fp_underflow, fp_inexact, fp_any_xcp Subroutine

Purpose

Tests to see if a floating–point exception has occurred.

Library

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

Syntax

```
#include <float.h>
#include <fpxcp.h>

int
fp_invalid_op()
int fp_divbyzero()

int fp_overflow()

int fp_underflow()

int
fp_inexact()
int fp_any_xcp()
```

Description

These subroutines aid in determining when an exception has occurred and the exception type. These subroutines can be called explicitly after blocks of code that may cause a floating–point exception.

Return Values

The **fp_invalid_op** subroutine returns a value of 1 if a floating–point invalid–operation exception status flag is set. Otherwise, a value of 0 is returned.

The **fp_divbyzero** subroutine returns a value of 1 if a floating–point divide–by–zero exception status flag is set. Otherwise, a value of 0 is returned.

The **fp_overflow** subroutine returns a value of 1 if a floating–point overflow exception status flag is set. Otherwise, a value of 0 is returned.

The **fp_underflow** subroutine returns a value of 1 if a floating–point underflow exception status flag is set. Otherwise, a value of 0 is returned.

The **fp_inexact** subroutine returns a value of 1 if a floating–point inexact exception status flag is set. Otherwise, a value of 0 is returned.

The **fp_any_xcp** subroutine returns a value of 1 if a floating–point invalid operation, divide–by–zero, overflow, underflow, or inexact exception status flag is set. Otherwise, a value of 0 is returned.

Implementation Specifics

These subroutines are part of Base Operating System (BOS) Runtime.

Related Information

The **fp_any_enable**, **fp_is_enabled**, **fp_enable_all**, **fp_enable fp_disable_all**, or **fp_disable** subroutine, **fp_clr_flag**, **fp_read_flag**, **fp_set_flag**, or **fp_swap_flag** subroutine, **fp_read_rnd** or **fp_swap_rnd** subroutine.

Floating–Point Processor Overview in *Hardware Technical Information-General Architectures*.

Floating–Poin t Exceptions Overview and Subroutines Overview in *AIX General Programming Concepts : Writing and Debugging Programs*.

fp_iop_snan, fp_iop_infsinf, fp_iop_infdinf, fp_iop_zrdzr, fp_iop_infmzr, fp_iop_invcmp, fp_iop_sqrt, fp_iop_convert, or fp_iop_vxsoft Subroutines

Purpose

Tests to see if a floating–point exception has occurred.

Library

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

Syntax

```
#include <float.h>
#include <fpxcp.h>

int fp_iop_snan()
int fp_iop_infsinf()

int
fp_iop_infdinf()
int fp_iop_zrdzr()

int
fp_iop_infmzr()
int fp_iop_invcmp()

int
fp_iop_sqrt()
int fp_iop_convert()

int
fp_iop_vxsoft ();
```

Description

These subroutines aid in determining when an exception has occurred and the exception type. These subroutines can be called explicitly after blocks of code that may cause a floating–point exception.

Return Values

The **fp_iop_snan** subroutine returns a value of 1 if a floating–point invalid–operation exception status flag is set due to a signaling NaN (NaNS) flag. Otherwise, a value of 0 is returned.

The **fp_iop_infsinf** subroutine returns a value of 1 if a floating–point invalid–operation exception status flag is set due to an INF–INF flag. Otherwise, a value of 0 is returned.

The **fp_iop_infdinf** subroutine returns a value of 1 if a floating–point invalid–operation exception status flag is set due to an INF/INF flag. Otherwise, a value of 0 is returned.

The **fp_iop_zrdzr** subroutine returns a value of 1 if a floating–point invalid–operation exception status flag is set due to a 0.0/0.0 flag. Otherwise, a value of 0 is returned.

The **fp_iop_infmzr** subroutine returns a value of 1 if a floating–point invalid–operation exception status flag is set due to an INF*0.0 flag. Otherwise, a value of 0 is returned.

The **fp_iop_invcmp** subroutine returns a value of 1 if a floating–point invalid–operation exception status flag is set due to a compare involving a NaN. Otherwise, a value of 0 is returned.

The **fp_iop_sqrt** subroutine returns a value of 1 if a floating–point invalid–operation exception status flag is set due to the calculation of a square root of a negative number. Otherwise, a value of 0 is returned.

The **fp_iop_convert** subroutine returns a value of 1 if a floating–point invalid–operation exception status flag is set due to the conversion of a floating–point number to an integer, where the floating–point number was a NaN, an INF, or was outside the range of the integer. Otherwise, a value of 0 is returned.

The **fp_iop_vxsoft** subroutine returns a value of 1 if the VXSOFT detail bit is on. Otherwise, a value of 0 is returned.

fp_raise_xcp Subroutine

Purpose

Generates a floating–point exception.

Library

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

Syntax

```
#include <fpxcp.h>

int fp_raise_xcp(
mask)
fpflag_t mask;
```

Description

The **fp_raise_xcp** subroutine causes any floating–point exceptions defined by the *mask* parameter to be raised immediately. If the exceptions defined by the *mask* parameter are enabled and the program is running in serial mode, the signal for floating–point exceptions, **SIGFPE**, is raised.

If more than one exception is included in the *mask* variable, the exceptions are raised in the following order:

1. Invalid
2. Dividebyzero
3. Underflow
4. Overflow
5. Inexact

Thus, if the user exception handler does not disable further exceptions, one call to the **fp_raise_xcp** subroutine can cause the exception handler to be entered many times.

Parameters

mask	Specifies a 32–bit pattern that identifies floating–point traps.

Return Values

The **fp_raise_xcp** subroutine returns 0 for normal completion and returns a nonzero value if an error occurs.

Related Information

The **fp_any_enable**, **fp_clr_flag**, **fp_read_flag**, **fp_swap_flag**, or **fpset_flag** subroutine, **fp_cpusync** subroutine, **fp_is_enabled**, **fp_enable_all**, **fp_enable**, **fp_disable_all**, or **fp_disable** subroutine, **fp_trap** subroutine, **sigaction** subroutine.

fp_read_rnd or fp_swap_rnd Subroutine

Purpose

Read and set the IEEE floating–point rounding mode.

Library

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

Syntax

```
#include <float.h>

fprnd_t fp_read_rnd()
fprnd_t fp_swap_rnd(RoundMode)
fprnd_t RoundMode;
```

Description

The **fp_read_rnd** subroutine returns the current rounding mode. The **fp_swap_rnd** subroutine changes the rounding mode to the *RoundMode* parameter and returns the value of the rounding mode before the change.

Floating–point rounding occurs when the infinitely precise result of a floating–point operation cannot be represented exactly in the destination floating–point format (such as double–precision format).

The *IEEE Standard for Binary Floating–Point Arithmetic* allows floating–point numbers to be rounded in four different ways: round toward zero, round to nearest, round toward +INF, and round toward –INF. Once a rounding mode is selected it affects all subsequent floating–point operations until another rounding mode is selected.

Note: The default floating–point rounding mode is round to nearest. All C main programs begin with the rounding mode set to round to nearest.

The encodings of the rounding modes are those defined in the *ANSI C Standard*. The **float.h** file contains definitions for the rounding modes. Below is the **float.h** definition, the *ANSI C Standard* value, and a description of each rounding mode.

float.h Definition	ANSI Value	Description
FP_RND_RZ	0	Round toward 0
FP_RND_RN	1	Round to nearest
FP_RND_RP	2	Round toward +INF
FP_RND_RM	3	Round toward –INF

The **fp_swap_rnd** subroutine can be used to swap rounding modes by saving the return value from **fp_swap_rnd**(*RoundMode*). This can be useful in functions that need to force a specific rounding mode for use during the function but wish to restore the caller's rounding mode on exit. Below is a code fragment that accomplishes this action:

```
save_mode = fp_swap_rnd (new_mode);
....desired code using new_mode
(void) fp_swap_rnd(save_mode); /*restore caller's mode*/
```

Parameters

RoundMode — Specifies one of the following modes: **FP_RND_RZ**, **FP_RND_RN**, **FP_RND_RP**, or **FP_RND_RM**.

Implementation Specifics

These subroutines are part of Base Operating System (BOS) Runtime.

Related Information

The **floor**, **ceil**, **nearest**, **trunc**, **rint**, **itrunc**, **uitrunc**, **fmod**, or **fabs** subroutine, **fp_any_enable**, **fp_is_enabled**, **fp_enable_all**, **fp_enable**,**fp_disable_all**, or **fp_disable** subroutine, **fp_clr_flag**, **fp_read_flag**, **fp_set_flag**, or **fp_swap_flag** subroutine.

Subroutines Overview in *AIX General Programming Concepts : Writing and Debugging Programs*.

fp_sh_info, fp_sh_trap_info, or fp_sh_set_stat Subroutine

Purpose

From within a floating–point signal handler, determines any floating–point exception that caused the trap in the process and changes the state of the Floating–Point Status and Control register (FPSCR) in the user process.

Library

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

Syntax

```
#include <fpxcp.h>
#include <fptrap.h>
#include <signal.h>

void fp_sh_info(scp, fcp, struct_size)
struct sigcontext *scp;
struct fp_sh_info *fcp;
size_t struct_size;

void fp_sh_trap_info(scp, fcp)
struct sigcontext *scp;
struct fp_ctx *fcp;

void fp_sh_set_stat(scp, fpscr)
struct sigcontext *scp;
fpstat_t fpscr;
```

Description

These subroutines are for use within a user–written signal handler. They return information about the process that was running at the time the signal occurred, and they update the Floating–Point Status and Control register for the process.

Note: The **fp_sh_trap_info** subroutine is maintained for compatibility only. It has been replaced by the **fp_sh_info** subroutine, which should be used for development.

These subroutines operate only on the state of the user process that was running at the time the signal was delivered. They read and write the **sigcontext** structure. They do not change the state of the signal handler process itself.

The state of the signal handler process can be modified by the **fp_any_enable**, **fp_is_enabled**, **fp_enable_all**, **fp_enable**, **fp_disable_all**, or **fp_disable** subroutine.

fp_sh_info

The **fp_sh_info** subroutine returns information about the process that caused the trap by means of a floating–point context (**fp_sh_info**) structure. This structure contains the following information:

```
typedef struct fp_sh_info {
fpstat_t       fpscr;
fpflag_t       trap;
short          trap_mode;
char           flags;
char           extra;
} fp_sh_info_t;
```

The fields are:

fpscr — The Floating–Point Status and Control register (FPSCR) in the user process at the time the interrupt occurred.

trap — A mask indicating the trap or traps that caused the signal handler to be entered. This mask is the logical OR operator of the enabled floating–point exceptions that occurred to cause the trap. This mask can have up to two exceptions; if there are two, the **INEXACT** signal must be one of them. If the mask is 0, the **SIGFPE** signal was raised not by a floating–point operation, but by the **kill** or **raise** subroutine or the **kill** command.

trap_mode — The trap mode in effect in the process at the time the signal handler was entered. The values returned in the **fp_sh_info.trap_mode** file use the following argument definitions:

FP_TRAP_OFF	Trapping off
FP_TRAP_SYNC	Precise trapping on
FP_TRAP_IMP_REC	Recoverable imprecise trapping on
FP_TRAP_IMP	Non–recoverable imprecise trapping on

flags — This field is interpreted as an array of bits and should be accessed with masks. The following mask is defined:

FP_IAR_STAT	If the value of the bit at this mask is 1, the exception was precise and the IAR points to the instruction that caused the exception. If the value bit at this mask is 0, the exception was imprecise.

fp_sh_trap_info

The **fp_sh_trap_info** subroutine is maintained for compatibility only. The **fp_sh_trap_info** subroutine returns information about the process that caused the trap by means of a floating–point context (**fp_ctx**) structure. This structure contains the following information:

```
fpstat_t fpscr;
fpflag_t trap;
```

The fields are:

fpscr — The Floating–Point Status and Control register (FPSCR) in the user process at the time the interrupt occurred.

trap — A mask indicating the trap or traps that caused the signal handler to be entered. This mask is the logical OR operator of the enabled floating–point exceptions that occurred to cause the trap. This mask can have up to two exceptions; if there are two, the **INEXACT** signal must be one of them. If the mask is 0, the **SIGFPE** signal was raised not by a floating–point operation, but by the **kill** or **raise** subroutine or the **kill** command.

fp_sh_set_stat

The **fp_sh_set_stat** subroutine updates the Floating–Point Status and Control register (FPSCR) in the user process with the value in the `fpscr` field.

The signal handler must either clear the exception bit that caused the trap to occur or disable the trap to prevent a recurrence. If the instruction generated more than one exception, and the signal handler clears only one of these exceptions, a signal is raised for the remaining exception when the next floating–point instruction is executed in the user process.

Parameters

fcp	Specifies a floating–point context structure.
scp	Specifies a **sigcontext** structure for the interrupt.
struct_size	Specifies the size of the **fp_sh_info** structure.
fpscr	Specifies which Floating–Point Status and Control register to update.

Related Information

The **fp_any_enable**, **fp_disable_all**, **fp_disable**, **fp_enable_all**, **fp_enable**, or **fp_is_enabled** subroutine, **fp_clr_flag**, **fp_read_flag**, **fp_set_flag**, or **fp_swap_flag** subroutine, **fp_trap** subroutine.

Floating–Point Exceptions Overview in *AIX General Programming Concepts : Writing and Debugging Programs*.

fp_trap Subroutine

Purpose

Queries or changes the mode of the user process to allow floating–point exceptions to generate traps.

Library

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

Syntax

```
#include <fptrap.h>

int fp_trap(flag)
int flag;
```

Description

The **fp_trap** subroutine queries and changes the mode of the user process to allow or disallow floating–point exception trapping. Floating–point traps can only be generated when a process is executing in a traps–enabled mode.

The default state is to execute in pipelined mode and not to generate floating–point traps.

Note: The **fp_trap** routines only change the execution state of the process. To generate floating–point traps, you must also enable traps. Use the **fp_enable** and **fp_enable_all** subroutines to enable traps.

Before calling the **fp_trap(FP_TRAP_SYNC)** routine, previous floating–point operations can set to True certain exception bits in the Floating–Point Status and Control register (FPSCR). Enabling these exceptions and calling the **fp_trap(FP_TRAP_SYNC)** routine does not cause an immediate trap to occur. That is, the operation of these traps is edge–sensitive, not level–sensitive.

The **fp_trap** subroutine does not clear the exception history. You can query this history by using any of the following subroutines:

- **fp_any_xcp**
- **fp_divbyzero**
- **fp_iop_convert**
- **fp_iop_infdinf**
- **fp_iop_infmzr**
- **fp_iop_infsinf**
- **fp_iop_invcmp**
- **fp_iop_snan**
- **fp_iop_sqrt**
- **fp_iop_vxsoft**
- **fp_iop_zrdzr**
- **fp_inexact**
- **fp_invalid_op**
- **fp_overflow**
- **fp_underflow**

Parameters

flag Specifies a query of or change in the mode of the user process:

FP_TRAP_OFF	Puts the user process into trapping–off mode and returns the previous mode of the process, either **FP_TRAP_SYNC**, **FP_TRAP_IMP**, **FP_TRAP_IMP_REC**, or **FP_TRAP_OFF**.
FP_TRAP_QUERY	Returns the current mode of the user process.
FP_TRAP_SYNC	Puts the user process into precise trapping mode and returns the previous mode of the process.
FP_TRAP_IMP	Puts the user process into non–recoverable imprecise trapping mode and returns the previous mode.
FP_TRAP_IMP_REC	Puts the user process into recoverable imprecise trapping mode and returns the previous mode.
FP_TRAP_FASTMODE	Puts the user process into the fastest trapping mode available on the hardware platform.

Note: Some hardware models do not support all modes. If an unsupported mode is requested, the **fp_trap** subroutine returns **FP_TRAP_UNIMPL**.

Return Values

If called with the **FP_TRAP_OFF**, **FP_TRAP_IMP**, **FP_TRAP_IMP_REC**, or **FP_TRAP_SYNC** flag, the **fp_trap** subroutine returns a value indicating which flag was in the previous mode of the process if the hardware supports the requested mode. If the hardware does not support the requested mode, the **fp_trap** subroutine returns **FP_TRAP_UNIMPL**.

If called with the **FP_TRAP_QUERY** flag, the **fp_trap** subroutine returns a value indicating the current mode of the process, either the **FP_TRAP_OFF**, **FP_TRAP_IMP**, **FP_TRAP_IMP_REC**, or **FP_TRAP_SYNC** flag.

If called with **FP_TRAP_FASTMODE**, the **fp_trap** subroutine sets the fastest mode available and returns the mode selected.

Error Codes

If the **fp_trap** subroutine is called with an invalid parameter, the subroutine returns **FP_TRAP_ERROR**.

If the requested mode is not supported on the hardware platform, the subroutine returns **FP_TRAP_UNIMPL**.

fp_trapstate Subroutine

Purpose

Queries or changes the trapping mode in the Machine Status register (MSR).

Note: This subroutine replaces the **fp_cpusync** subroutine. The **fp_cpusync** subroutine is supported for compatibility, but the **fp_trapstate** subroutine should be used for development.

Library

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

Syntax

```
#include <fptrap.h>

int fp_trapstate (int)
```

Description

The **fp_trapstate** subroutine is a service routine used to query or set the trapping mode. The trapping mode determines whether floating–point exceptions can generate traps, and can affect execution speed. See Floating–Point Exceptions Overview in *AIX General Programming Concepts : Writing and Debugging Programs* for a description of precise and imprecise trapping modes. Floating–point traps can be generated by the hardware only when the processor is in a traps–enabled mode.

The **fp_trapstate** subroutine changes only the trapping mode. It is a service routine for use in developing custom floating–point exception–handling software. If you are using the **fp_enable** or **fp_enable_all** subroutine or the **fp_sh_info** or **fp_sh_set_stat** subroutine, you must use the **fp_trap** subroutine to change the process' trapping mode.

Parameters

flag Specifies a query of, or change in, the trap mode:

FP_TRAPSTATE_OFF Sets the trapping mode to Off and returns the previous mode.

FP_TRAPSTATE_QUERY Returns the current trapping mode without modifying it.

FP_TRAPSTATE_IMP Puts the process in non–recoverable imprecise trapping mode and returns the previous state.

FP_TRAPSTATE_IMP_REC
Puts the process in recoverable imprecise trapping mode and returns the previous state.

FP_TRAPSTATE_PRECISE
Puts the process in precise trapping mode and returns the previous state.

FP_TRAPSTATE_FASTMODE
Puts the process in the fastest trap–generating mode available on the hardware platform and returns the state selected.

Note: Some hardware models do not support all modes. If an unsupported mode is requested, the **fp_trapstate** subroutine returns **FP_TRAP_UNIMPL** and the trapping mode is not changed.

Return Values

If called with the **FP_TRAPSTATE_OFF**, **FP_TRAPSTATE_IMP**, **FP_TRAPSTATE_IMP_REC**, or **FP_TRAPSTATE_PRECISE** flag, the **fp_trapstate** subroutine returns a value indicating the previous mode of the process. The value may be **FP_TRAPSTATE_OFF**, **FP_TRAPSTATE_IMP**, **FP_TRAPSTATE_IMP_REC**, or **FP_TRAPSTATE_PRECISE**. If the hardware does not support the requested mode, the **fp_trapstate** subroutine returns **FP_TRAP_UNIMPL**.

If called with the **FP_TRAP_QUERY** flag, the **fp_trapstate** subroutine returns a value indicating the current mode of the process. The value may be **FP_TRAPSTATE_OFF**, **FP_TRAPSTATE_IMP**, **FP_TRAPSTATE_IMP_REC**, or **FP_TRAPSTATE_PRECISE**.

If called with the **FP_TRAPSTATE_FASTMODE** flag, the **fp_trapstate** subroutine returns a value indicating which mode was selected. The value may be **FP_TRAPSTATE_OFF**, **FP_TRAPSTATE_IMP**, **FP_TRAPSTATE_IMP_REC**, or **FP_TRAPSTATE_PRECISE**.

Implementation Specifics

This subroutine is part of Base Operating System (BOS) Runtime.

Related Information

The **fp_any_enable**, **fp_disable_all**, **fp_disable**, **fp_enable_all**, **fp_enable**, or **fp_is_enabled** subroutine, **fp_clr_flag**, **fp_read_flag**, **fpset_flag**, or **fp_swap_flag** subroutine, **sigaction**, **signal**, or **sigvec** subroutine.

The Floating–Point Processor Overview in *Hardware Technical Information-General Architectures*.

Floating–Point Exceptions Overview in *AIX General Programming Concepts : Writing and Debugging Programs*.

fread or fwrite Subroutine

Purpose

Reads and writes binary files.

Library

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

Syntax

```
#include <stdio.h>
size_t fread ( (void *)
Pointer, Size, NumberOfItems, Stream)
size_t Size, NumberOfItems;
FILE *Stream;
size_t fwrite (Pointer, Size, NumberOfItems, Stream)
const void *Pointer;
size_t Size, NumberOfItems;
FILE *Stream;
```

Description

The **fread** subroutine copies the number of data items specified by the *NumberOfItems* parameter from the input stream into an array beginning at the location pointed to by the *Pointer* parameter. Each data item has the form **Pointer*.

The **fread** subroutine stops copying bytes if an end–of–file (EOF) or error condition is encountered while reading from the input specified by the *Stream* parameter, or when the number of data items specified by the *NumberOfItems* parameter have been copied. This subroutine leaves the file pointer of the *Stream* parameter, if defined, pointing to the byte following the last byte read. The **fread** subroutine does not change the contents of the *Stream* parameter.

The `st_atime` field will be marked for update by the first successful run of the **fgetc**, **fgets**, **fgetwc**, **fgetws**, **fread**, **fscanf**, **getc**, **getchar**, **gets**, or **scanf** subroutine using a stream that returns data not supplied by a prior call to the **ungetc** or **ungetwc** subroutine.

Note: The **fread** subroutine is a buffered **read** subroutine library call. It reads data in 4KB blocks. For tape block sizes greater than 4KB, use the **open** subroutine and **read** subroutine.

The **fwrite** subroutine writes items from the array pointed to by the *Pointer* parameter to the stream pointed to by the *Stream* parameter. Each item's size is specified by the *Size* parameter. The **fwrite** subroutine writes the number of items specified by the *NumberOfItems* parameter. The file–position indicator for the stream is advanced by the number of bytes successfully written. If an error occurs, the resulting value of the file–position indicator for the stream is indeterminate.

The **fwrite** subroutine appends items to the output stream from the array pointed to by the *Pointer* parameter. The **fwrite** subroutine appends as many items as specified in the *NumberOfItems* parameter.

The **fwrite** subroutine stops writing bytes if an error condition is encountered on the stream, or when the number of items of data specified by the *NumberOfItems* parameter have been written. The **fwrite** subroutine does not change the contents of the array pointed to by the *Pointer* parameter.

The `st_ctime` and `st_mtime` fields will be marked for update between the successful run of the **fwrite** subroutine and the next completion of a call to the **fflush** or **fclose** subroutine on the same stream, the next call to the **exit** subroutine, or the next call to the **abort** subroutine.

Parameters

Pointer	Points to an array.
Size	Specifies the size of the variable type of the array pointed to by the *Pointer* parameter. The *Size* parameter can be considered the same as a call to **sizeof** subroutine.
NumberOfItems	Specifies the number of items of data.
Stream	Specifies the input or output stream.

Return Values

The **fread** and **fwrite** subroutines return the number of items actually transferred. If the *NumberOfItems* parameter contains a 0, no characters are transferred, and a value of 0 is returned. If the *NumberOfItems* parameter contains a negative number, it is translated to a positive number, since the *NumberOfItems* parameter is of the unsigned type.

Error Codes

If the **fread** subroutine is unsuccessful because the I/O stream is unbuffered or data needs to be read into the I/O stream's buffer, it returns one or more of the following error codes:

EAGAIN	Indicates that the **O_NONBLOCK** flag is set for the file descriptor specified by the *Stream* parameter, and the process would be delayed in the **fread** operation.
EBADF	Indicates that the file descriptor specified by the *Stream* parameter is not a valid file descriptor open for reading.
EINTR	Indicates that the read operation was terminated due to receipt of a signal, and no data was transferred.

Note: Depending upon which library routine the application binds to, this subroutine may return **EINTR**. Refer to the **signal** subroutine regarding **sa_restart**.

EIO	Indicates that the process is a member of a background process group attempting to perform a read from its controlling terminal, and either the process is ignoring or blocking the **SIGTTIN** signal or the process group has no parent process.
ENOMEM	Indicates that insufficient storage space is available.
ENXIO	Indicates that a request was made of a nonexistent device.

If the **fwrite** subroutine is unsuccessful because the I/O stream is unbuffered or the I/O stream's buffer needs to be flushed, it returns one or more of the following error codes:

EAGAIN	Indicates that the **O_NONBLOCK** flag is set for the file descriptor specified by the *Stream* parameter, and the process is delayed in the write operation.
EBADF	Indicates that the file descriptor specified by the *Stream* parameter is not a valid file descriptor open for writing.
EFBIG	Indicates that an attempt was made to write a file that exceeds the file size of the process limit or the systemwide maximum file size.
EINTR	Indicates that the write operation was terminated due to the receipt of a signal, and no data was transferred.
EIO	Indicates that the process is a member of a background process group attempting to perform a write to its controlling terminal, the **TOSTOP** signal is set, the process is neither ignoring nor blocking the **SIGTTOU** signal, and the process group of the process is orphaned.

ENOSPC Indicates that there was no free space remaining on the device containing the file.

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

The **fwrite** subroutine is also unsuccessful due to the following error conditions:

ENOMEM Indicates that insufficient storage space is available.

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

Implementation Specifics

These subroutines are part of Base Operating System (BOS) Runtime.

Related Information

The **abort** subroutine, **exit** subroutine, **fflush** or **fclose** subroutine, **fopen**, **freopen**, or **fdopen** subroutine, **getc**, **getchar**, **fgetc**, or **getw** subroutine, **getwc**, **fgetwc**, or **getwchar** subroutine, **gets** or **fgets** subroutine, **getws** or **fgetws** subroutine, **open** subroutine, **print**, **fprintf**, or **sprintf** subroutine, **putc**, **putchar**, **fputc**, or **putw** subroutine, **putwc**, **putwchar**, or **fputwc** subroutine, **puts** or **fputs** subroutine, **putws** or **fputws** subroutine, **read** subroutine, **scanf**, **fscanf**, **sscanf**, or **wsscanf** subroutine, **ungetc** or **ungetwc** subroutine, **write** subroutine.

The Input and Output Handling Programmer's Overview in *AIX General Programming Concepts : Writing and Debugging Programs*.

freeaddrinfoSubroutine

Purpose

To free memory allocated by **getaddrinfo**. This includes the addrinfo structures, the socket address structures, and canonical host name strings pointed to by the addrinfo structures.

Library

Library (**libc.a**)

Syntax

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

Description

This function frees any dynamic storage pointed to by elements of ai, as well as the space for ai itself. Also, it will descend the linked list, repeating this process for all nodes in the list until a NULL ai_next pointer is encountered.

Related Information

The **getaddrinfo** subroutine, **gai_strerror**, and **getnameinfo** subroutine.

frevoke Subroutine

Purpose

Revokes access to a file by other processes.

Library

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

Syntax

```
int frevoke (FileDescriptor)
int FileDescriptor;
```

Description

The **frevoke** subroutine revokes access to a file by other processes.

All accesses to the file are revoked, except through the file descriptor specified by the *FileDescriptor* parameter to the **frevoke** subroutine. Subsequent attempts to access the file, using another file descriptor established before the **frevoke** subroutine was called, fail and cause the process to receive a return value of –1, and the **errno** global variable is set to **EBADF**.

A process can revoke access to a file only if its effective user ID is the same as the file owner ID or if the invoker has root user authority.

Note: The **frevoke** subroutine has no affect on subsequent attempts to open the file. To ensure exclusive access to the file, the caller should change the mode of the file before issuing the **frevoke** subroutine. Currently the **frevoke** subroutine works only on terminal devices.

Parameters

FileDescriptor	A file descriptor returned by a successful **open** subroutine.

Return Values

Upon successful completion, the **frevoke** subroutine returns a value of 0.

If the **frevoke** subroutine fails, it returns a value of –1 and the **errno** global variable is set to indicate the error.

Error Codes

The **frevoke** subroutine fails if the following is true:

EBADF	The *FileDescriptor* value is not the valid file descriptor of a terminal.
EPERM	The effective user ID of the calling process is not the same as the file owner ID.
EINVAL	Revocation of access rights is not implemented for this file.

frexp, frexpl, ldexp, ldexpl, modf, or modfl Subroutine

Purpose

Manipulates floating–point numbers.

Library

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

Syntax

```
#include <math.h>

double frexp (Value, Exponent)
double Value;
int *Exponent;

long double frexpl (Value, Exponent)
long double Value;
int Exponent;

double ldexp (Mantissa, Exponent)
double Mantissa;
int Exponent ;

long double ldexpl (Mantissa, Exponent)
long double Mantissa;
int Exponent;

double modf (Value, IntegerPointer)
double Value, *IntegerPointer;

long double modfl (Value, IntegerPointer)
long double Value, *IntegerPointer;
```

Description

Every nonzero number can be written uniquely as *x* * 2***n*, where the mantissa (fractional part) *x* is in the range 0.5 <= |*x*| < 1.0, and the exponent *n* is an integer.

The **frexp** subroutine breaks a floating–point number into a normalized fraction and an integral power of 2. It stores the integer in the object pointed to by the *Exponent* parameter and returns the fraction part. The **frexpl** subroutine performs the same function for numbers in the long double data type.

The **ldexp** subroutine multiplies a floating–point number by an integral power of 2. The **ldexpl** subroutine performs the same function for numbers in the long double data type.

The **modf** subroutine breaks the *Value* parameter into an integral and fractional part, each of which has the same sign as the value. It stores the integral part in a **double** variable at the location pointed to by the *IntegerPointer* parameter. The **modfl** subroutine performs the same function for numbers in the long double data type.

Parameters

Value	Specifies a double–precision floating–point value.
Exponent	For the **frexp** subroutine, specifies an integer pointer to store the exponent; for the **ldexp** subroutine, specifies an integer value.
Mantissa	Specifies a double–precision floating–point value.
IntegerPointer	Specifies a pointer to the **double** variable in which to store the signed integral part.

Return Values

The **frexp** and **frexpl** subroutines return a value *x* such that *x* is in the range 0.5 <= |*x*| < 1.0 or is 0, and the *Value* parameter equals *x* * 2**(**Exponent*). If the *Value* parameter is 0, the object pointed to by the **Exponent* parameter and *x* are also 0. If the *Value* parameter is a NaN (not–a–number), *x* is a NaNQ, and the object pointed to by the **Exponent* parameter is set to **LONG_MIN**. If the *Value* parameter is +INF, then +INF is returned and the object pointed to by the **Exponent* parameter is set to **INT_MAX**. If the *Value* parameter is –INF, then –INF is returned and the object pointed to by the **Exponent* parameter is set to **INT_MIN**.

The **ldexp** and **ldexpl** subroutines return the value *x* * 2**(*Exponent*).

The **modf** and **modfl** subroutines return the signed fractional part of the *Value* parameter and stores the signed integral part in the object pointed to by the *IntegerPointer* parameter. If the *Value* parameter is a NaN value, then a NaNQ value is returned, and a NaNQ is stored in the object pointed to by the *IntegerPointer* parameter. If the *Value* parameter is +/–INF, then +/– 0.0 is returned, and +/–INF is stored in the object pointed to by the *IntegerPointer* parameter.

Error Codes

If the result of the **ldexp** or **ldexpl** subroutine overflows, then +/– **HUGE_VAL** is returned, and the global variable **errno** is set to **ERANGE**.

If the result of the **ldexp** or **ldexpl** subroutine underflows, 0 is returned, and the **errno** global variable is set to a **ERANGE** value.

Implementation Specifics

These subroutines are part of Base Operating System (BOS) Runtime.

Related Information

The **scanf**, **fscanf**, or **sscanf** subroutine, **sgetl** or **sputl** subroutine.

Subroutines Overview in *AIX General Programming Concepts : Writing and Debugging Programs*.

128–Bit long Double Floating–Point Format in *AIX General Programming Concepts : Writing and Debugging Programs*.

fscntl Subroutine

Purpose

Controls file system control operations.

Library

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

Syntax

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

int fscntl (vfs_id, Command, Argument,  ArgumentSize)
int vfs_id;
int Command;
char *Argument;
int ArgumentSize;
```

Description

The **fscntl** subroutine performs a variety of file system–specific functions. These functions typically require root user authority.

At present, only one file system, the Journaled File System, supports any commands via the **fscntl** subroutine.

Note: Application programs should not call this function, which is reserved for system management commands such as the **chfs** command.

Parameters

vfs_id	Identifies the file system to be acted upon. This information is returned by the **stat** subroutine in the `st_vfs` field of the **stat.h** file.
Command	Identifies the operation to be performed.
Argument	Specifies a pointer to a block of file system specific information that defines how the operation is to be performed.
ArgumentSize	Defines the size of the buffer pointed to by the *Argument* parameter.

Return Values

Upon successful completion, the **fscntl** subroutine returns a value of 0. Otherwise, a value of –1 is returned and the **errno** global variable is set to indicate the error.

Error Codes

The **fscntl** subroutine fails if one or both of the following are true:

EINVAL	The *vfs_id* parameter does not identify a valid file system.
EINVAL	The *Command* parameter is not recognized by the file system.

Implementation Specifics

This subroutine is part of Base Operating System (BOS) Runtime.

Related Information

The **chfs** command.

The **stat.h** file.

Understanding File–System Helpers in *AIX General Programming Concepts : Writing and Debugging Programs* explains file system helpers and examines file system–helper execution syntax.

fseek, fseeko, fseeko64, rewind, ftell, ftello, ftello64, fgetpos, fgetpos64, fsetpos, or fsetpos64 Subroutine

Purpose

Repositions the file pointer of a stream.

Library

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

Syntax

```
#include <stdio.h>

int fseek (Stream, Offset, Whence)
FILE *Stream;
long int Offset;
int Whence;

void rewind (Stream)
FILE *Stream;

long int ftell (Stream)
FILE *Stream;

int fgetpos (Stream, Position)
FILE *Stream;
fpos_t *Position;

int fsetpos (Stream, Position)
FILE *Stream;
const fpos_t *Position;
```

Note: The **fseeko**, **fseeko64**, **ftello**, **ftello64**, **fgetpos64**, and **fsetpot64** subroutines apply to Version 4.2 and later releases.

```
int fseeko (Stream, Offset, Whence)
FILE *Stream;
off_t Offset;
int Whence;

int fseeko64 (Stream, Offset, Whence)
FILE *Stream;
off64_t Offset;
int Whence;

off_t int ftello (Stream)
FILE *Stream;

off64_t int ftello64 (Stream)
FILE *Stream;

int fgetpos64 (Stream, Position)
FILE *Stream;
fpos64_t *Position;

int fsetpos64 (Stream, Position)
FILE *Stream;
const fpos64_t *Position;
```

Description

Note: The **fseeko**, **fseeko64**, **ftello**, **ftello64**, **fgetpos64**, and **fsetpot64** subroutines apply to Version 4.2 and later releases.

The **fseek**, **fseeko** and **fseeko64** subroutines set the position of the next input or output operation on the I/O stream specified by the Stream parameter. The position if the next operation is determined by the *Offset* parameter, which can be either positive or negative.

The **fseek**, **fseeko** and **fseeko64** subroutines set the file pointer associated with the specified *Stream* as follows:

- If the *Whence* parameter is set to the **SEEK_SET** value, the pointer is set to the value of the *Offset* parameter.
- If the *Whence* parameter is set to the **SEEK_CUR** value, the pointer is set to its current location plus the value of the *Offset* parameter.
- If the *Whence* parameter is set to the **SEEK_END** value, the pointer is set to the size of the file plus the value of the *Offset* parameter.

The **fseek**, **fseeko**, and **fseeko64** subroutine are unsuccessful if attempted on a file that has not been opened using the **fopen** subroutine. In particular, the **fseek** subroutine cannot be used on a terminal or on a file opened with the **popen** subroutine. The **fseek** and **fseeko** subroutines will also fail when the resulting offset is larger than can be properly returned.

The **rewind** subroutine is equivalent to calling the **fseek** subroutine using parameter values of （*Stream*,**SEEK_SET,SEEK_SET**）, except that the **rewind** subroutine does not return a value.

The **fseek**, **fseeko**, **fseeko64** and **rewind** subroutines undo any effects of the **ungetc** and **ungetwc** subroutines and clear the end–of–file (EOF) indicator on the same stream.

The **fseek**, **fseeko**, and **fseeko64** function allows the file–position indicator to be set beyond the end of existing data in the file. If data is written later at this point, subsequent reads of data in the gap will return bytes of the value 0 until data is actually written into the gap.

A successful calls to the **fsetpos** or **fsetpos64** subroutines clear the **EOF** indicator and undoes any effects of the **ungetc** and **ungetwc** subroutines.

After an **fseek**, **fseeko**, **fseeko64** or a **rewind** subroutine, the next operation on a file opened for update can be either input or output.

ftell, **ftello** and **ftello64** subroutines return the position current value of the file–position indicator for the stream pointed to by the *Stream* parameter. **ftell** and **ftello** will fail if the resulting offset is larger than can be properly returned.

The **fgetpos** and **fgetpos64** subroutines store the current value of the file–position indicator for the stream pointed to by the *Stream* parameter in the object pointed to by the *Position* parameter. The **fsetpos** and **fsetpos64** set the file–position indicator for *Stream* according to the value of the *Position* parameter, which must be the result of a prior call to **fgetpos** or **fgetpos64** subroutine. **fgetpos** and **fsetpos** will fail if the resulting offset is larger than can be properly returned.

Parameters

Stream	Specifies the input/output (I/O) stream.
Offset	Determines the position of the next operation.
Whence	Determines the value for the file pointer associated with the *Stream* parameter.
Position	Specifies the value of the file–position indicator.

Return Values

Upon successful completion, the **fseek**, **fseeko** and **fseeko64** subroutine return a value of 0. Otherwise, it returns a value of –1.

Upon successful completion, the **ftell**, **ftello** and **ftello64** subroutine return the offset of the current byte relative to the beginning of the file associated with the named stream. Otherwise, a **long int** value of –1 is returned and the **errno** global variable is set.

Upon successful completion, the **fgetpos**, **fgetpos64**, **fsetpos** and **fsetpos64** subroutines return a value of 0. Otherwise, a nonzero value is returned and the **errno** global variable is set to the specific error.

The **errno** global variable is used to determine if an error occurred during a **rewind** subroutine call.

Error Codes

If the **fseek**, **fseeko**, **fseeko64**, **ftell**, **ftello**, **ftello64** or **rewind** subroutine are unsuccessful because the stream is unbuffered or the stream buffer needs to be flushed and the call to the subroutine causes an underlying **lseek** or **write** subroutine to be invoked, it returns one or more of the following error codes:

EAGAIN	Indicates that the **O_NONBLOCK** flag is set for the file descriptor, delaying the process in the write operation.
EBADF	Indicates that the file descriptor underlying the *Stream* parameter is not open for writing.
EFBIG	Indicates that an attempt has been made to write to a file that exceeds the file–size limit of the process or the maximum file size.
EFBIG	Indicates that the file is a regular file and that an attempt was made to write at or beyond the offset maximum associated with the corresponding stream.
EINTR	Indicates that the write operation has been terminated because the process has received a signal, and either no data was transferred, or the implementation does not report partial transfers for this file.
EIO	Indicates that the process is a member of a background process group attempting to perform a **write** subroutine to its controlling terminal, the **TOSTOP** flag is set, the process is not ignoring or blocking the **SIGTTOU** signal, and the process group of the process is orphaned. This error may also be returned under implementation–dependent conditions.
ENOSPC	Indicates that no remaining free space exists on the device containing the file.
EPIPE	Indicates that an attempt has been made to write to a pipe or FIFO that is not open for reading by any process. A **SIGPIPE** signal will also be sent to the process.
EINVAL	Indicates that the *Whence* parameter is not valid. The resulting file–position indicator will be set to a negative value. The **EINVAL** error code does not apply to the **ftell** and **rewind** subroutines.
ESPIPE	Indicates that the file descriptor underlying the *Stream* parameter is associated with a pipe or FIFO.
EOVERFLOW	Indicates that for *fseek*, the resulting file offset would be a value that cannot be represented correctly in an object of type *long*.
EOVERFLOW	Indicates that for *fseeko*, the resulting file offset would be a value that cannot be represented correctly in an object of type *off_t*.
ENXIO	Indicates that a request was made of a non–existent device, or the request was outside the capabilities of the device.

The **fgetpos** and **fsetpos** subroutines are unsuccessful due to the following conditions:

EINVAL	Indicates that either the *Stream* or the *Position* parameter is not valid. The **EINVAL** error code does not apply to the **fgetpos** subroutine.
EBADF	Indicates that the file descriptor underlying the *Stream* parameter is not open for writing.
ESPIPE	Indicates that the file descriptor underlying the *Stream* parameter is associated with a pipe or FIFO.

The **fseek**, **fseeko**, **ftell**, **ftello**, **fgetpos**, and **fsetpos** subroutines are unsucessful under the following condition:

EOVERFLOW	The resulting could not be returned properly.

Implementation Specifics

These subroutines are part of Base Operating system (BOS) Runtime.

Related Information

The **closedir** subroutine, **fopen**, **fopen64**, **freopen**, **freopen64** or **fdopen** subroutine, **lseek** or **lseek64** subroutine, **opendir**, **readdir**, **rewinddir**, **seekdir**, or **telldir** subroutine, **popen** subroutine, **ungetc** or **ungetwc** subroutine, **write**, **writex**, **writev**, or **writevx** subroutine.

Input and Output Handling Programmer's Overview in *AIX General Programming Concepts : Writing and Debugging Programs*.

fsync Subroutine

Purpose

Writes changes in a file to permanent storage.

Library

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

Syntax

```
#include <unistd.h>

int fsync (FileDescriptor)
int FileDescriptor;
```

Description

The **fsync** subroutine causes all modified data in the open file specified by the *FileDescriptor* parameter to be saved to permanent storage. On return from the **fsync** subroutine, all updates have been saved on permanent storage.

Data written to a file that a process has opened for deferred update (with the **O_DEFER** flag) is not written to permanent storage until another process issues an **fsync** subroutine against this file or runs a synchronous **write** subroutine (with the **O_SYNC** flag) on this file. See the **fcntl.h** file and the **open** subroutine for descriptions of the **O_DEFER** and **O_SYNC** flags respectively.

Note: The file identified by the *FileDescriptor* parameter must be open for writing when the **fsync** subroutine is issued or the call is unsuccessful. This restriction was not enforced in BSD systems.

Parameters

FileDescriptor	A valid, open file descriptor.

Return Values

Upon successful completion, the **fsync** subroutine returns a value of 0. Otherwise, a value of –1 is returned and the **errno** global variable is set to indicate the error.

Error Codes

The **fsync** subroutine is unsuccessful if one or more of the following are true:

EIO	An I/O error occurred while reading from or writing to the file system.
EBADF	The *FileDescriptor* parameter is not a valid file descriptor open for writing.
EINVAL	The file is not a regular file.
EINTR	The **fsync** subroutine was interrupted by a signal.

Implementation Specifics

This subroutine is part of Base Operating System (BOS) Runtime.

Related Information

The **open**, **openx**, or **creat** subroutine, **sync** subroutine, **write**, **writex**, **writev**, or **writevx** subroutine.

The **fcntl.h** file.

Files, Directories, and File Systems Overview for Programmers in *AIX General Programming Concepts : Writing and Debugging Programs* contains information about i–nodes, file descriptors, file–space allocation, and more.

ftok Subroutine

Purpose

Generates a standard interprocess communication key.

Library

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

Syntax

```
#include <sys/types.h>
#include <sys/ipc.h>

key_t ftok (Path, ID)
char *Path;
int ID;
```

Description

Attention: If the *Path* parameter of the **ftok** subroutine names a file that has been removed while keys still refer to it, the **ftok** subroutine returns an error. If that file is then re–created, the **ftok** subroutine will probably return a key different from the original one.

Attention: Each installation should define standards for forming keys. If standards are not adhered to, unrelated processes may interfere with each other's operation.

The **ftok** subroutine returns a key, based on the *Path* and *ID* parameters, to be used to obtain interprocess communication identifiers. The **ftok** subroutine returns the same key for linked files if called with the same *ID* parameter. Different keys are returned for the same file if different *ID* parameters are used.

All interprocess communication facilities require you to supply a key to the **msgget**, **semget**, and **shmget** subroutines in order to obtain interprocess communication identifiers. The **ftok** subroutine provides one method for creating keys, but other methods are possible. For example, you can use the project ID as the most significant byte of the key, and use the remaining portion as a sequence number.

Parameters

Path	Specifies the path name of an existing file that is accessible to the process.
ID	Specifies a character that uniquely identifies a project.

Return Values

When successful, the **ftok** subroutine returns a key that can be passed to the **msgget**, **semget**, or **shmget** subroutine.

Error Codes

The **ftok** subroutine returns the value **(key_t)–1** if one or more of the following are true:

- The file named by the *Path* parameter does not exist.
- The file named by the *Path* parameter is not accessible to the process.
- The *ID* parameter has a value of 0.

Implementation Specifics

This subroutine is part of Base Operating System (BOS) Runtime.

Related Information

The **msgget** subroutine, **semget** subroutine, **shmget** subroutine.

Subroutines Overview and Understanding Memory Mapping in *AIX General Programming Concepts : Writing and Debugging Programs*.

ftw or ftw64 Subroutine

Purpose

Walks a file tree.

Library

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

Syntax

```
#include <ftw.h>

int ftw (Path, Function, Depth)
char *Path;
int (*Function(const char*, const struct stat*, int);
int Depth;

int ftw64 (Path, Function, Depth)
char *Path;
int (*Function(const char*, const struct stat64*, int);
int Depth;
```

Description

The **ftw** and **ftw64** subroutines recursively searches the directory hierarchy that descends from the directory specified by the *Path* parameter.

For each file in the hierarchy, the **ftw** and **ftw64** subroutines call the function specified by the *Function* parameter. **ftw** passes it a pointer to a null–terminated character string containing the name of the file, a pointer to a stat structure containing information about the file, and an integer. **ftw64** passes it a pointer to a null–terminated character string containing the name of the file, a pointer to a **stat64** structure containing information about the file, and an integer.

The integer passed to the *Function* parameter identifies the file type with one of the following values:

FTW_F	Regular file
FTW_D	Directory
FTW_DNR	Directory that cannot be read
FTW_SL	Symbolic Link
FTW_NS	File for which the **stat** structure could not be executed successfully

If the integer is **FTW–DNR**, the files and subdirectories contained in that directory are not processed.

If the integer is **FTW–NS**, the **stat** structure contents are meaningless. An example of a file that causes **FTW–NS** to be passed to the *Function* parameter is a file in a directory for which you have read permission but not execute (search) permission.

The **ftw** and **ftw64** subroutines finish processing a directory before processing any of its files or subdirectories.

The **ftw** and **ftw64** subroutines continue the search until the directory hierarchy specified by the *Path* parameter is completed, an invocation of the function specified by the *Function* parameter returns a nonzero value, or an error is detected within the **ftw** and **ftw64** subroutines, such as an I/O error.

The **ftw** and **ftw64** subroutines traverse symbolic links encountered in the resolution of the *Path* parameter, including the final component. Symbolic links encountered while walking the directory tree rooted at the *Path* parameter are not traversed.

The **ftw** and **ftw64** subroutines use one file descriptor for each level in the tree. The *Depth* parameter specifies the maximum number of file descriptors to be used. In general, the **ftw** and **ftw64** subroutines runs faster if the value of the *Depth* parameter is at least as large as the number of levels in the tree. However, the value of the *Depth* parameter must not be greater than the number of file descriptors currently available for use. If the value of the *Depth* parameter is 0 or a negative number, the effect is the same as if it were 1.

Because the **ftw** and **ftw64** subroutines are recursive, it is possible for it to terminate with a memory fault due to stack overflow when applied to very deep file structures.

The **ftw** and **ftw64** subroutines use the **malloc** subroutine to allocate dynamic storage during its operation. If the **ftw** and **ftw64** subroutined is terminated prior to its completion, such as by the **longjmp** subroutine being executed by the function specified by the *Function* parameter or by an interrupt routine, the **ftw** and **ftw64** subroutines cannot free that storage. The storage remains allocated. A safe way to handle interrupts is to store the fact that an interrupt has occurred, and arrange to have the function specified by the *Function* parameter return a nonzero value the next time it is called.

Parameters

Path	Specifies the directory hierarchy to be searched.
Function	Specifies the file type.
Depth	Specifies the maximum number of file descriptors to be used. *Depth* cannot be greater than OPEN_MAX which is described in the sys/limits.h header file.

Return Values

If the tree is exhausted, the **ftw** and **ftw64** subroutines returns a value of 0. If the subroutine pointed to by **fn** returns a nonzero value, **ftw** and **ftw64** subroutines stops its tree traversal and returns whatever value was returned by the subroutine pointed to by **fn**. If the **ftw** and **ftw64** subroutines detects an error, it returns a –1 and sets the **errno** global variable to indicate the error.

Error Codes

If the **ftw** or **ftw64** subroutines detect an error, a value of –1 is returned and the **errno** global variable is set to indicate the error.

The **ftw** and **ftw64** subroutine are unsuccessful if:

EACCES	Search permission is denied for any component of the *Path* parameter or read permission is denied for *Path*.
ENAMETOOLONG	The length of the path exceeds **PATH_MAX** while **_POSIX_NO_TRUNC** is in effect.
ENOENT	The *Path* parameter points to the name of a file that does not exist or points to an empty string.
ENOTDIR	A component of the *Path* parameter is not a directory.

The **ftw** subroutine is unsuccessful if:

EOVERFLOW	A file in *Path* is of a size larger than 2 Gigabytes.

Implementation Specifics

This subroutines is part of Base Operating System (BOS) Runtime.

Related Information

The **malloc**, **free**, **realloc**, **calloc**, **mallopt**, **mallinfo**, or **alloca** subroutine, **setjmp** or **longjmp** subroutine, **signal** subroutine, **stat** subroutine.

Searching and Sorting Example Program and Subroutines Overview in *AIX General Programming Concepts : Writing and Debugging Programs*.

fwide Subroutine

Purpose

Set stream orientation.

Library

Standard Library (**libc.a**)

Syntax

```
#include <stdio.h>
#include <wchar.h>

int fwid (FILE * stream, int mode),
```

Description

The **fwide** function determines the orientation of the stream pointed to by stream. If mode is greater than zero, the function first attempts to make the stream wide–orientated. If mode is less than zero, the function first attempts to make the stream byte–orientated. Otherwise, mode is zero and the function does not alter the orientation of the stream.

If the orientation of the stream has already been determined, **fwide** does not change it.

Because no return value is reserved to indicate an error, an application wishing to check for error situations should set errno to 0, then call **fwide**, then check errno and if it is non–zero, assume an error has occurred.

Return Values

The **fwide** function returns a value greater than zero if, after the call, the stream has wide–orientation, a value less than zero if the stream has byte–orientation, or zero if the stream has no orientation.

Errors

The **fwide** function may fail if:

EBADF	The stream argument is not a valid stream.

Implementation Specifics

A call to **fwide** with mode set to zero can be used to determine the current orientation of a stream.

Related Information

The **wchar.h** file

fwprintf, wprintf, swprintf Subroutines

Purpose

Print formatted wide–character output.

Library

Standard Library (**libc.a**)

Syntax

```
#include <stdio.h>
#include <wchar.h>

int fwprintf ( FILE * stream, const wchar_t * format,...)
int wprintf (const wchar_t * format,..)
int swprintf (wchar_t *s, size_t n, const wchar_t * format,...)
```

Description

The **fwprintf** function places output on the named output stream. The **wprintf** function places output on the standard output stream **stdout**. The **swprintf** function places output followed by the null wide–character in consecutive wide–characters starting at ***s**; no more than **n** wide–characters are written, including a terminating null wide–character, which is always added (unless **n** is zero).

Each of these functions converts, formats and prints its arguments under control of the **format** wide–character string. The **format** is composed of zero or more directives: **ordinary wide–characters**, which are simply copied to the output stream and **conversion specifications** , each of which results in the fetching of zero or more arguments. The results are undefined if there are insufficient arguments for the **format.** If the **format** is exhausted while arguments remain, the excess arguments are evaluated but are otherwise ignored.

EX Conversions can be applied to the **nth** argument after the **format** in the argument list, rather than to the next unused argument. In this case, the conversion wide–character % (see below) is replaced by the sequence **%n$**, where n is a decimal integer in the range [1, {NL_ARGMAX}], giving the position of the argument in the argument list. This feature provides for the definition of format wide–character strings that select arguments in an order appropriate to specific languages (see the EXAMPLES section).

In format wide–character strings containing the **%n$** form of conversion specifications, numbered arguments in the argument list can be referenced from the format wide–character string as many times as required.

In format wide–character strings containing the % form of conversion specifications, each argument in the argument list is used exactly once.

All forms of the **fwprintf** functions allow for the insertion of a language–dependent radix character in the output string, output as a wide–character value. The radix character is defined in the program's locale (category LC_NUMERIC). In the POSIX locale, or in a locale where the radix character is not defined, the radix character defaults to a period (.).

EX Each conversion specification is introduced by the % wide–character or by the wide–character sequence %n$,after which the following appear in sequence:

- Zero or more **flags** (in any order), which modify the meaning of the conversion specification.
- An optional minimum **field width**. If the converted value has fewer wide–characters than the field width, it will be padded with spaces by default on the left; it will be padded on the right, if the left–adjustment flag (–), described below, is given to the field width. The field width takes the form of an asterisk (*), described below, or a decimal integer.

- An optional **precision** that gives the minimum number of digits to appear for the d, i, o, u, **x** and **X** conversions; the number of digits to appear after the radix character for the e, E and f conversions; the maximum number of significant digits for the g and G conversions; or the maximum number of wide–characters to be printed from a string in s conversions. The precision takes the form of a period (.) followed either by an asterisk (*), described below, or an optional decimal digit string, where a null digit string is treated as 0. If a precision appears with any other conversion wide–character, the behaviour is undefined.
- An optional l (ell) specifying that a following **c** conversion wide–character applies to a **wint_t** argument; an optional l specifying that a following s conversion wide–character applies to a **wchar_t** argument; an optional h specifying that a following d, i, o, u, **x** or **X** conversion wide–character applies to a type **short int** or type **unsigned short int** argument (the argument will have been promoted according to the integral promotions, and its value will be converted to type **short int** or **unsigned short int** before printing); an optional h specifying that a following n conversion wide–character applies to a pointer to a type **short int** argument; an optional l (ell) specifying that a following d, i, o, u, **x** or **X** conversion wide–character applies to a type **long int** or **unsigned long int** argument; an optional l (ell) specifying that a following n conversion wide–character applies to a pointer to a type **long int** argument; or an optional L specifying that a following e, E, f, g or G conversion wide–character applies to a type **long double** argument. If an h, l or L appears with any other conversion wide–character, the behavior is undefined.
- A **conversion wide–character** that indicates the type of conversion to be applied.

A field width, or precision, or both, may be indicated by an asterisk (*). In this case an argument of type int supplies the field width or precision. Arguments specifying field width, or precision, or both must appear in that order before the argument, if any, to be converted. A negative field width is taken as a – flag followed by a positive field width. A negative precision is taken as if EX the precision were omitted. In format wide–character strings containing the %n$ form of a conversion specification, a field width or precision may be indicated by the sequence *m$, where m is a decimal integer in the range [1, {NL_ARGMAX}] giving the position in the argument list (after the format argument) of an integer argument containing the field width or precision, for example:

```
wprintf(L"%1$d:%2$.*3$d:%4$.*3$d\n", hour, min, precision, sec);
```

The **format** can contain either numbered argument specifications (that is, **%n$** and ***m$),** or unnumbered argument specifications (that is, % and *), but normally not both. The only exception to this is that %% can be mixed with the **%n$** form. The results of mixing numbered and unnumbered argument specifications in a **format** wide–character string are undefined. When numbered argument specifications are used, specifying the Nth argument requires that all the leading arguments, from the first to the (N–1)th, are specified in the format wide–character string.

The flag wide–characters and their meanings are:

'	The integer portion of the result of a decimal conversion (%i, %d, %u, %f, %g or %G) will be formatted with thousands' grouping wide–characters. For other conversions the behaviour is undefined. The non–monetary grouping wide–character is used.
–	The result of the conversion will be left–justified within the field. The conversion will be right–justified if this flag is not specified.
+	The result of a signed conversion will always begin with a sign (+ or –). The conversion will begin with a sign only when a negative value is converted if this flag is not specified.
space	If the first wide–character of a signed conversion is not a sign or if a signed conversion results in no wide–characters, a space will be prefixed to the result. This means that if the space and + flags both appear, the space flag will be ignored.

This flag specifies that the value is to be converted to an alternative form. For o conversion, it increases the precision (if necessary) to force the first digit of the result to be 0. For x or X conversions, a non–zero result will have 0x (or 0X) prefixed to it. For e, E, f, g or G conversions, the result will always contain a radix character, even if no digits follow it. Without this flag, a radix character appears in the result of these conversions only if a digit follows it. For g and G conversions, trailing zeros will **not** be removed from the result as they normally are. For other conversions, the behavior is undefined.

0 For d, i, o, u, x, X, e, E, f, g and G conversions, leading zeros (following any indication of sign or base) are used to pad to the field width; no space padding is performed. If the 0 and – flags both appear, the 0 flag will be ignored. For d, i, o, u, x and X conversions, if a precision is specified, the 0 flag will be ignored. If the 0 and ' flags both appear, the grouping wide–characters are inserted before zero padding. For other conversions, the behavior is undefined.

The conversion wide–characters and their meanings are:

d,i The **int** argument is converted to a signed decimal in the style [–] **dddd**. The precision specifies the minimum number of digits to appear; if the value being converted can be represented in fewer digits, it will be expanded with leading zeros. The default precision is 1. The result of converting 0 with an explicit precision of 0 is no wide–characters.

o The **unsigned int** argument is converted to unsigned octal format in the style **dddd**. The precision specifies the minimum number of digits to appear; if the value being converted can be represented in fewer digits, it will be expanded with leading zeros. The default precision is 1. The result of converting 0 with an explicit precision of 0 is no wide–characters.

u The **unsigned int** argument is converted to unsigned decimal format in the style **dddd**. The precision specifies the minimum number of digits to appear; if the value being converted can be represented in fewer digits, it will be expanded with leading zeros. The default precision is 1. The result of converting 0 with an explicit precision of 0 is no wide–characters.

x The **unsigned int** argument is converted to unsigned hexadecimal format in the style **dddd**; the letters abcdef are used. The precision specifies the minimum number of digits to appear; if the value being converted can be represented in fewer digits, it will be expanded with leading zeros. The default precision is 1. The result of converting 0 with an explicit precision of 0 is no wide–characters.

X Behaves the same as the x conversion wide–character except that letters ABCDEF are used instead of abcdef.

f The **double** argument is converted to decimal notation in the style [–] **ddd.ddd**, where the number of digits after the radix character is equal to the precision specification. If the precision is missing, it is taken as 6; if the precision is explicitly 0 and no # flag is present, no radix character appears. If a radix character appears, at least one digit appears before it. The value is rounded to the appropriate number of digits.

The **fwprintf** family of functions may make available wide–character string representations for infinity and NaN.

e, E The **double** argument is converted in the style [–] **d.ddde +/– dd**, where there is one digit before the radix character (which is non–zero if the argument is non–zero) and the number of digits after it is equal to the precision; if the precision is missing, it is taken as 6; if the precision is 0 and no # flag is present, no radix character appears. The value is rounded to the appropriate number of digits. The E conversion wide–character will produce a number with E instead of e introducing the exponent. The exponent always contains at least two digits. If the value is 0, the exponent is 0.

The **fwprintf** family of functions may make available wide–character string representations for infinity and NaN.

g, G The **double** argument is converted in the style f or e (or in the style E in the case of a G conversion wide–character), with the precision specifying the number of significant digits. If an explicit precision is 0, it is taken as 1. The style used depends on the value converted; style e (or E) will be used only if the exponent resulting from such a conversion is less than –4 or greater than or equal to the precision. Trailing zeros are removed from the fractional portion of the result; a radix character appears only if it is followed by a digit.

The **fwprintf** family of functions may make available wide–character string representations for infinity and NaN.

c If no l (ell) qualifier is present, the **int** argument is converted to a wide–character as if by calling the **btowc** function and the resulting wide–character is written. Otherwise the **wint_t** argument is converted to **wchar_t,** and written.

s If no l (ell) qualifier is present, the argument must be a pointer to a character array containing a character sequence beginning in the initial shift state. Characters from the array are converted as if by repeated calls to the **mbrtowc** function, with the conversion state described by an **mbstate_t** object initialised to zero before the first character is converted, and written up to (but not including) the terminating null wide–character. If the precision is specified, no more than that many wide–characters are written. If the precision is not specified or is greater than the size of the array, the array must contain a null wide–character.

If an l (ell) qualifier is present, the argument must be a pointer to an array of type **wchar_t.** Wide characters from the array are written up to (but not including) a terminating null wide–character. If no precision is specified or is greater than the size of the array, the array must contain a null wide–character. If a precision is specified, no more than that many wide–characters are written.

p The argument must be a pointer to void. The value of the pointer is converted to a sequence of printable wide–characters, in an implementation–dependent manner. The argument must be a pointer to an integer into which is written the number of wide–characters written to the output so far by this call to one of the **fwprintf** functions. No argument is converted.

C Same as lc.

S Same as ls.

% Output a % wide–character; no argument is converted. The entire conversion specification must be %%.

If a conversion specification does not match one of the above forms, the behavior is undefined.

In no case does a non–existent or small field width cause truncation of a field; if the result of a conversion is wider than the field width, the field is simply expanded to contain the conversion result. Characters generated by **fwprintf** and **wprintf** are printed as if **fputwc** had been called.

The **st_ctime** and **st_mtime** fields of the file will be marked for update between the call to a successful execution of **fwprintf** or **wprintf** and the next successful completion of a call to **fflush** or **fclose** on the same stream or a call to exit or abort.

Return Values

Upon successful completion, these functions return the number of wide–characters transmitted excluding the terminating null wide–character in the case of **swprintf** or a negative value if an output error was encountered.

Error Codes

For the conditions under which **fwprintf** and **wprintf** will fail and may fail, refer to **fputwc** .In addition, all forms of **fwprintf** may fail if:

EILSEQ	A wide–character code that does not correspond to a valid character has been detected
EINVAL	There are insufficient arguments.
	In addition, **wprintf** and **fwprintf** may fail if:
ENOMEM	Insufficient storage space is available.

Examples

To print the language–independent date and time format, the following statement could be used:

```
wprintf (format, weekday, month, day, hour, min);
```

For American usage, format could be a pointer to the wide–character string:

```
L"%s, %s %d, %d:%.2d\n"
```

producing the message:

```
Sunday, July 3, 10:02
```

whereas for German usage, format could be a pointer to the wide–character string:

```
L"%1$s, %3$d. %2$s, %4$d:%5$.2d\n"
```

producing the message:

```
Sonntag, 3. July, 10:02
```

Implementation Specifics

These subroutines are part of Base Operating System (BOS) subroutines.

Related Information

The **btowc** subroutine.

The **fputwc** subroutine.

The **fwscanf** subroutine.

The **setlocale** subroutine.

The **mbrtowc** subroutine.

The **stdio.h** file.

The **wchar.h** file.

The **XBD** specification, *Chapter 5, Locale*.

fwscanf, wscanf, swscanf Subroutines

Purpose

Convert formatted wide–character input

Library

Standard Library (**libc.a**)

Syntax

```
#include <stdio.h>
#include <wchar.h>

int fwscanf (FILE * stream, const wchar_t * format, ...);
int wscanf (const wchar_t * format, ...);
int swscanf (const wchar_t * s, const wchar_t * format, ...);
```

Description

The **fwscanf** function reads from the named input stream. The **wscanf** function reads from the standard input stream stdin. The **swscanf** function reads from the wide–character string s. Each function reads wide–characters, interprets them according to a format, and stores the results in its arguments. Each expects, as arguments, a control wide–character string format described below, and a set of pointer arguments indicating where the converted input should be stored. The result is undefined if there are insufficient arguments for the format. If the format is exhausted while arguments remain, the excess arguments are evaluated but are otherwise ignored.

Conversions can be applied to the **nth** argument after the **format** in the argument list, rather than to the next unused argument. In this case, the conversion wide–character % (see below) is replaced by the sequence **%n$,** where **n** is a decimal integer in the range [1, {NL_ARGMAX}]. This feature provides for the definition of format wide–character strings that select arguments in an order appropriate to specific languages. In format wide–character strings containing the **%n$** form of conversion specifications, it is unspecified whether numbered arguments in the argument list can be referenced from the format wide–character string more than once.

The format can contain either form of a conversion specification, that is, % or **%n$,** but the two forms cannot normally be mixed within a single format wide–character string. The only exception to this is that %% or %* can be mixed with the **%n$** form.

The **fwscanf** function in all its forms allows for detection of a language–dependent radix character in the input string, encoded as a wide–character value. The radix character is defined in the program's locale (category LC_NUMERIC). In the POSIX locale, or in a locale where the radix character is not defined, the radix character defaults to a period (.).

The format is a wide–character string composed of zero or more directives. Each directive is composed of one of the following: one or more white–space wide–characters (space, tab, newline, vertical–tab or form–feed characters); an ordinary wide–character (neither % nor a white–space character); or a conversion specification. Each conversion specification is introduced by a % or the sequence **%n$** after which the following appear in sequence:

- An optional assignment–suppressing character *.
- An optional non–zero decimal integer that specifies the maximum field width.
- An optional size modifier h, l (ell) or L indicating the size of the receiving object. The conversion wide–characters c, s and [must be preceded by l (ell) if the corresponding argument is a pointer to **wchar_t** rather than a pointer to a character type. The conversion wide–characters d, i and n must be preceded by h if the corresponding argument is a pointer to **short int** rather than a pointer to **int,** or by l (ell) if it is a pointer to **long int**. Similarly, the conversion wide–characters o, u and x must be preceded by h if

the corresponding argument is a pointer to **unsigned short int** rather than a pointer to **unsigned int**, or by l (ell) if it is a pointer to **unsigned long int**. The conversion wide–characters e, f and g must be preceded by l (ell) if the corresponding argument is a pointer to **double** rather than a pointer to **float,or** by L if it is a pointer to long double. If an h, l (ell) or L appears with any other conversion wide–character, the behavior is undefined.

- A conversion wide–character that specifies the type of conversion to be applied. The valid conversion wide–characters are described below.

The **fwscanf** functions execute each directive of the format in turn. If a directive fails, as detailed below, the function returns. Failures are described as input failures (due to the unavailability of input bytes) or matching failures (due to inappropriate input).

A directive composed of one or more white–space wide–characters is executed by reading input until no more valid input can be read, or up to the first wide–character which is not a white–space wide–character, which remains unread.

A directive that is an ordinary wide–character is executed as follows. The next wide–character is read from the input and compared with the wide–character that comprises the directive; if the comparison shows that they are not equivalent, the directive fails, and the differing and subsequent wide–characters remain unread.

A directive that is a conversion specification defines a set of matching input sequences, as described below for each conversion wide–character. A conversion specification is executed in the following steps:

Input white–space wide–characters (as specified by **iswspace**) are skipped, unless the conversion specification includes a [, c or n conversion character.

An item is read from the input, unless the conversion specification includes an n conversion wide–character. An input item is defined as the longest sequence of input wide–characters, not exceeding any specified field width, which is an initial subsequence of a matching sequence. The first wide–character, if any, after the input item remains unread. If the length of the input item is 0, the execution of the conversion specification fails; this condition is a matching failure, unless end–of–file, an encoding error, or a read error prevented input from the stream, in which case it is an input failure.

Except in the case of a % conversion wide–character, the input item (or, in the case of a %n conversion specification, the count of input wide–characters) is converted to a type appropriate to the conversion wide–character. If the input item is not a matching sequence, the execution of the conversion specification fails; this condition is a matching failure. Unless assignment suppression was indicated by a *, the result of the conversion is placed in the object pointed to by the first argument following the **format** argument that has not already received a conversion result if the conversion specification is introduced by %, or in the nth argument if introduced by the wide–character sequence %n$. If this object does not have an appropriate type, or if the result of the conversion cannot be represented in the space provided, the behavior is undefined. The following conversion wide–characters are valid:

d Matches an optionally signed decimal integer, whose format is the same as expected for the subject sequence of **wcstol** with the value 10 for the base argument. In the absence of a size modifier, the corresponding argument must be a pointer to **int**.

i Matches an optionally signed integer, whose format is the same as expected for the subject sequence of **wcstol** with 0 for the base argument. In the absence of a size modifier, the corresponding argument must be a pointer to **int**.

o Matches an optionally signed octal integer, whose format is the same as expected for the subject sequence of **wcstoul** with the value 8 for the base argument. In the absence of a size modifier, the corresponding argument must be a pointer to **unsigned int**.

u Matches an optionally signed decimal integer, whose format is the same as expected for the subject sequence of **wcstoul** with the value 10 for the base argument. In the absence of a size modifier, the corresponding argument must be a pointer to **unsigned int**.

x Matches an optionally signed hexadecimal integer, whose format is the same as expected for the subject sequence of **wcstoul** with the value 16 for the base argument. In the absence of a size modifier, the corresponding argument must be a pointer to **unsigned int**.

e, f, g Matches an optionally signed floating–point number, whose format is the same as expected for the subject sequence of **wcstod** . In the absence of a size modifier, the corresponding argument must be a pointer to float.

If the **fwprintf** family of functions generates character string representations for infinity and NaN (a 7858 symbolic entity encoded in floating–point format) to support the ANSI/IEEE Std 754:1985 standard, the **fwscanf**5 family of functions will recognise them as input.

s Matches a sequence of non white–space wide–characters. If no l (ell) qualifier is present, characters from the input field are converted as if by repeated calls to the **wcrtomb** function, with the conversion state described by an **mbstate_t** object initialised to zero before the first wide–character is converted. The corresponding argument must be a pointer to a character array large enough to accept the sequence and the terminating null character, which will be added automatically.

Otherwise, the corresponding argument must be a pointer to an array of **wchar_t** large enough to accept the sequence and the terminating null wide–character, which will be added automatically.

[Matches a non–empty sequence of wide–characters from a set of expected wide–characters (the **scanset**). If no l (ell) qualifier is present, wide–characters from the input field are converted as if by repeated calls to the **wcrtomb** function, with the conversion state described by an **mbstate_t** object initialised to zero before the first wide–character is converted. The corresponding argument must be a pointer to a character array large enough to accept the sequence and the terminating null character, which will be added automatically.

If an l (ell) qualifier is present, the corresponding argument must be a pointer to an array of **wchar_t** large enough to accept the sequence and the terminating null wide–character, which will be added automatically

The conversion specification includes all subsequent widw characters in the **format** string up to and including the matching right square bracket (]). The wide–characters between the square brackets (the **scanlist**) comprise the scanset, unless the wide–character after the left square bracket is a circumflex (^), in which case the scanset contains all wide–characters that do not appear in the scanlist between the circumflex and the right square bracket. If the conversion specification begins with [ ] or [^], the right square bracket is included in the scanlist and the next right square bracket is the matching right square bracket that ends the conversion specification; otherwise the first right square bracket is the one that ends the conversion specification. If a – is in the scanlist and is not the first wide–character, nor the second where the first wide–character is a ^;, nor the last wide–character, the behavior is implementation–dependent.

c Matches a sequence of wide–characters of the number specified by the field width (1 if no field width is present in the conversion specification). If no l (ell) qualifier is present, wide–characters from the input field are converted as if by repeated calls to the **wcrtomb** function, with the conversion state described by an **mbstate_t** object initialised to zero before the first wide–character is converted. The corresponding argument must be a pointer to a character array large enough to accept the sequence. No null character is added.

Otherwise, the corresponding argument must be a pointer to an array of **wchar_t** large enough to accept the sequence. No null wide–character is added.

p Matches an implementation–dependent set of sequences, which must be the same as the set of sequences that is produced by the %p conversion of the corresponding **fwprintf** functions. The corresponding argument must be a pointer to a pointer to void. The interpretation of the input item is implementation–dependent. If the input item is a value converted earlier during the same program execution, the pointer that results will compare equal to that value; otherwise the behavior of the %p conversion is undefined.

n No input is consumed. The corresponding argument must be a pointer to the integer into which is to be written the number of wide–characters read from the input so far by this call to the **fwscanf** functions. Execution of a %n conversion specification does not increment the assignment count returned at the completion of execution of the function.

C Same as lc.

S Same as ls.

% Matches a single %; no conversion or assignment occurs. The complete conversion specification must be %%.

If a conversion specification is invalid, the behavior is undefined.

The conversion characters E, G and X are also valid and behave the same as, respectively, e, g and x.

If end–of–file is encountered during input, conversion is terminated. If end–of–file occurs before any wide–characters matching the current conversion specification (except for %n) have been read (other than leading white–space, where permitted), execution of the current conversion specification terminates with an input failure. Otherwise, unless execution of the current conversion specification is terminated with a matching failure, execution of the following conversion specification (if any) is terminated with an input failure.

Reaching the end of the string in **swscanf** is equivalent to encountering end–of–file for **fwscanf**.

If conversion terminates on a conflicting input, the offending input is left unread in the input. Any trailing white space (including newline) is left unread unless matched by a conversion specification. The success of literal matches and suppressed assignments is only directly determinable via the %n conversion specification.

The **fwscanf** and **wscanf** functions may mark the **st_atime** field of the file associated with stream for update. The **st_atime** field will be marked for update by the first successful execution of **fgetc**, **fgetwc**, **fgets**, **fgetws**, **fread**, **getc**, **getwc**, **getchar**, **getwchar**, **gets**, **fscanf** or **fwscanf** using stream that returns data not supplied by a prior call to **ungetc**.

Return Values

Upon successful completion, these functions return the number of successfully matched and assigned input items; this number can be 0 in the event of an early matching failure. If the input ends before the first matching failure or conversion, EOF is returned. If a read error occurs the error indicator for the stream is set, EOF is returned, and errno is set to indicate the error.

Error Codes

For the conditions under which the **fwscanf** functions will fail and may fail, refer to **fgetwc**. In addition, **fwscanf** may fail if:

EILSEQ Input byte sequence does not form a valid character.

EINVAL There are insufficient arguments.

Examples

The call:

```
int i, n; float x; char name[50];
n = wscanf(L"%d%f%s", &i, &x, name);
```

with the input line:

```
25 54.32E-1 Hamster
```

will assign to n the value 3, to i the value 25, to x the value 5.432, and name will contain the string Hamster.

The call:

```
int i; float x; char name[50];
(void) wscanf(L"%2d%f%*d %[0123456789]", &i, &x, name);
```

with input:

```
56789 0123 56a72
```

will assign 56 to **i**, 789.0 to x, skip 0123, and place the string 56\0 in **name**. The next call to **getchar** will return the character a.

Implementation Specifics

In format strings containing the % form of conversion specifications, each argument in the argument list is used exactly once.

Related Information

The **getwc** subroutine.

The **fwprintf** subroutine.

The **setlocale** subroutine.

The **wcstod** subroutine.

The **wcstol** subroutine.

The **wcstoul** subroutine.

The **wcrtomb** subroutine.

The **langinfo.h** file.

The **stdio.h** file.

The **wchar.h** file.

The **XBD** specification, *Chapter 5, Locale*.

gai_strerror Subroutine

Purpose

Facilitates consistent error information from EAI_* values returned by **getaddrinfo**.

Library

Library (**libc.a**)

Syntax

```
#include <sys/socket.h>
#include <netdb.h>
char *
gai_strerror (ecode)
int ecode;
int
gai_strerror_r (ecode, buf, buflen)
int ecode;
char *buf;
int buflen;
```

Description

Facilitates consistent error information from EAI_* values returned by **getaddrinfo**.

For multithreaded environments, the second version should be used. In **gai_strerror_r**, *buf* is a pointer to a data area to be filled in. *buflen* is the length (in bytes) available in *buf*.

It is the caller's responsibility to insure that *buf* is sufficiently large to store the requested information, including a trailing null character. It is the responsibility of the function to insure that no more than *buflen* bytes are written into *buf*.

Return Values

If successful, a pointer to a string containing an error message appropriate for the EAI_* errors is returned. If ecode is not one of the EAI_* values, a pointer to a string indicating an unknown error is returned.

Related Information

The **getaddrinfo** subroutine, **freeaddrinfo** subroutine, and **getnameinfo** subroutine.

Subroutines Overview in *AIX Version 4 General Programming Concepts: Writing and Debugging Programs*.

get_speed, set_speed, or reset_speed Subroutines

Purpose

Set and get the terminal baud rate.

Library

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

Syntax

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

int get_speed (FileDescriptor)
int FileDescriptor;

int set_speed (FileDescriptor, Speed)
int FileDescriptor;
int Speed;

int reset_speed (FileDescriptor)
int FileDescriptor;
```

Description

The baud rate functions **set_speed** subroutine and **get_speed** subroutine are provided top allow the user applications to program any value of the baud rate that is supported by the asynchronous adapter, but that cannot be expressed using the termios subroutines **cfsetospeed**, **cfsetispeed**, **cfgetospeed**, and **cfsgetispeed**. Those subroutines are indeed limited to the set values {BO, B50, ..., B38400} described in <**termios.h**>.

Interaction with the termios Baud flags:

If the terminal's device driver supports these subroutines, it has two interfaces for baud rate manipulation.

Operation for Baud Rate:

normal mode: This is the default mode, in which a termios supported speed is in use.

speed–extended mode: This mode is entered either by calling **set_speed** subroutine a non–termios supported speed at the configuration of the line.

In this mode, all the calls to **tcgetattr** subroutine or **TCGETS ioctl** subroutine will have B50 in the returned termios structure.

If **tcsetatt** subroutine or **TCSETS**, **TCSETAF**, or **TCSETAW ioctl** subroutines is called and attempt to set B50, the actual baud rate is not changed. If is attempts to set any other termios–supported speed, the driver will switch back to the normal mode and the requested baud rate is set. Calling **reset_speed** subroutine is another way to switch back to the normal mode.

Parameters

FileDescriptor	Specifies an open file descriptor.
Speed	The integer value of the requested speed.

Return Values

Upon successful completion, **set_speed** and **reset_speed** return a value of 0, and **get_speed** returns a positive integer specifying the current speed of the line. Otherwise, a value of –1 is returned and the **errno** global variable is set to indicate the error.

Error Codes

EINVAL The *FileDescriptor* parameter does not specify a valid file descriptor for a **tty** the recognizes the **set_speed**, **get_speed** and **reset_speed** subroutines, or the *Speed* parameter of **set_speed** is not supported by the terminal.

Plus all the **errno** codes that may be set in case of failure in an **ioctl** subroutine issued to a streams based **tty.**

Related Information

cfgetospeed, **cfsetospeed**, **cfgetispeed**, or **cfsetispeed** subroutines.

getaddrinfo Subroutine

Purpose Protocol–independent hostname–to–address translation.

Note: Hostname–to–address translation is done in a protocol–independent fashion using this function.

Attention: This specification is taken from IEEE POSIX 1003.1g (Protocol Independent Interfaces) DRAFT 6.3. This function may be modified to match that specification as it develops. Should there be any discrepancies between this description and the POSIX description, the POSIX description takes precedence.

Library

Library (**libinet.*x***)

Syntax

```
#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 first two arguments describe the hostname and/or service name to be referenced. 0 (zero) or 1 (one) of these arguments may be NULL. A non–NULLhostname 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 third argument specifies hints concerning the desired return information. To be valid, the hints structure must contain zero (or NULL) values for all members, with the exceptions of: ai_flags, ai_family, ai_socktype, and ai_protocol. These members may be set to a specific value to indicate desired results (ai_family may be set to PF_INET6 to indicate only IPv6 sockets), or to zero (or the appropriate unspecified value (PF_UNSPEC for ai_family)) to indicate that any type will be accepted.

The addrinfo structure is defined as:

```
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;   /* canoncial 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 fourth argument. 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	hostname nor 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

Implementation Specifics

The hostname and servname arguments are pointers to null–terminated strings or NULL. One or both of these two arguments must be a non–NULL pointer. In the normal client scenario, both the hostname and servname are specified. In the normal server scenario, only the servname is specified. A non–NULL hostname string can be either a host name or a numeric host address string (i.e., a dotted–decimal IPv4 address or an IPv6 hex address) 2E 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 third argument, 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 0 for ai_socktype means the caller will accept any socket type. A value of 0 for ai_protocol means the caller will accept any protocol. For example, if the caller handles only TCP and not UDP, then the ai_socktype member of the hints structure should be set to SOCK_STREAM when getaddrinfo() is called. If the caller handles only IPv4 and not IPv6, then the ai_family member of the hints structure should be set to PF_INET when getaddrinfo() is called. If the third argument to getad drinfo() is a NULL pointer, this is the same as if the caller had filled 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 final argument. 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() function. 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, then the caller plans to use the returned socket address structure in a call to bind(). In this case, if the hostname argument is a NULL pointer, then 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, then the returned socket address structure will be ready for a call to connect() (for a connection–oriented protocol) or either connect(), sendto(), or sendmsg() (for a connectionless protocol). In this case, if the hostname argument is a NULL pointer, then 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, then 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.

All of the information returned by getaddrinfo() 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 the function freeaddrinfo is called.

Related Information

The **freeaddrinfo** subroutine.

getargs Subroutine

Purpose

Gets arguments of a process.

Library

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

Syntax

```
#include <procinfo.h>
#include <sys/types.h>

int getargs (processBuffer, bufferLen, argsBuffer, argsLen)
struct procsinfo *processBuffer;
int bufferLen;
char *argsBuffer;
int argsLen;
```

Description

The **getargs** subroutine returns a list of parameters that were passed to a command when it was started. Only one process can be examined per call to **getargs**.

The **getargs** subroutine uses the pi_pid field of *processBuffer* to determine which process to look for. *bufferLen* should be set to size of **struct procsinfo**. Parameters are returned in *argsBuffer*, which should be allocated by the caller. The size of this array must be given in *argsLen*.

On return, *argsBuffer* consists of a succession of strings, each terminated with a null character (ascii '\0'). Hence, two consecutive `NULL`s indicate the end of the list.

Note: The arguments may be changed asynchronously by the process, but results are not guaranteed to be consistent.

Parameters

processBuffer	Specifies the address of a **procsinfo** structure, whose pi_pid field should contain the pid of the process that is to be looked for.
bufferLen	Specifies the size of a single **procsinfo** structure,
argsBuffer	Specifies the address of an array of characters to be filled with a series of strings representing the parameters that are needed. An extra `NULL` character marks the end of the list. This array must be allocated by the caller.
argsLen	Specifies the size of the *argsBuffer* array. No more than *argsLen* characters are returned.

Return Values

If successful, the **getargs** subroutine returns zero. Otherwise, a value of –1 is returned and the **errno** global variable is set to indicate the error.

Error Codes

The **getargs** subroutine does not succeed if the following are true:

EBADF The specified process does not exist.

EFAULT The copy operation to the buffer was not successful or the *processBuffer* or *argsBuffer* parameters are invalid.

EINVAL The *bufferLen* parameter does not contain the size of a single **procsinfo** structure.

Implementation Specifics

This subroutine is part of the Base Operating System (BOS) Runtime.

Related Information

The **getevars**, **getpid**, **getpgrp**, **getppid**, or **getthrds** subroutines.

The **ps** command.

getaudithostattr, IDtohost, hosttoID, nexthost or putaudithostattr Subroutine

Purpose

Accesses the host information in the audit host database.

Library

Security Library (**libc.a**)

Syntax

```
#include <usersec.h>

int  getaudithostattr (Hostname, Attribute, Value, Type)
char *Hostname;
char *Attribute;
void *Value;
int Type;

char *IDtohost (ID);
char *ID;

char *hosttoID (Hostname, Count);
char *Hostname;
int Count;

char *nexthost (void);

int putaudithostattr (Hostname, Attribute, Value, Type);
char *Hostname;
char *Attribute;
void *Value;
int Type;
```

Description

These subroutines access the audit host information.

The **getaudithostattr** subroutine reads a specified attribute from the host database. If the database is not already open, this subroutine does an implicit open for reading.

Similarly the **putaudithostattr** subroutine writes a specified attribute into the host database. If the database is not already open, this subroutine does an implicit open for reading and writing. Data changed by the **putaudithostattr** must be explicitly committed by calling the **putaudithostattr** subroutine with a Type of **SEC_COMMIT**. Until all the data is committed, only these subroutines within the process return written data.

New entries in the host database must first be created by invoking **putaudithostattr** with the **SEC_NEW** type.

The **IDtohost** subroutine converts an 8 byte host identifier into a hostname.

The **hosttoID** subroutine converts a hostname to a pointer to an array of valid 8 byte host identifiers. A pointer to the array of identifiers is returned on success. A **NULL** pointer is returned on failure. The number of known host identifiers is returned in ***Count**.

The **nexthost** subroutine returns a pointer to the name of the next host in the audit host database.

Parameters

Attribute
: Specifies which attribute is read. The following possible attributes are defined in the **usersec.h** file:

  S_AUD_CPUID Host identifier list. The attribute type is **SEC_LIST**.

Count
: Specifies the number of 8 byte host identifier entries that are available in the *IDarray* parameter or that have been returned in the *IDarray* parameter.

Hostname
: Specifies the name of the host for the operation.

ID
: An 8 byte host identifier.

IDarray
: Specifies a pointer to an array of 1 or more 8 byte host identifiers.

Type
: Specifies the type of attribute expected. Valid types are defined in **usersec.h**. The only valid Type value is **SEC_LIST**.

Value
: The return value for read operations and the new value for write operations.

Return Values

On successful completion, the **getaudithostattr**, **IDtohost**, **hosttoID**, **nexthost**, or **putaudithostattr** subroutine returns 0. If unsuccessful, the subroutine returns non–zero.

Error Codes

The **getaudithostattr**, **IDtohost**, **hosttoID**, **nexthost**, or **putaudithostattr** subroutine fails if the following is true:

EINVAL
: If invalid attribute *Name* or if *Count* is equal to zero for the **hosttoID** subroutine.

ENOENT
: If there is no matching *Hostname* entry in the database.

Related Information

The **auditmerge** command, **auditpr** command, **auditselect** command, **auditstream** command.

The **auditread** subroutine, **setaudithostdb** or **endaudithostdb** subroutine.

getc, getchar, fgetc, or getw Subroutine

Purpose

Gets a character or word from an input stream.

Library

Standard I/O Package (**libc.a**)

Syntax

```
#include <stdio.h>

int getc (Stream)
FILE *Stream;

int fgetc (Stream)
FILE *Stream;

int getchar (void)

int getw (Stream)
FILE *Stream;
```

Description

The **getc** macro returns the next byte as an **unsigned char** data type converted to an **int** data type from the input specified by the *Stream* parameter and moves the file pointer, if defined, ahead one byte in the *Stream* parameter. The **getc** macro cannot be used where a subroutine is necessary; for example, a subroutine pointer cannot point to it.

Because it is implemented as a macro, the **getc** macro does not work correctly with a *Stream* parameter value that has side effects. In particular, the following does not work:

```
getc(*f++)
```

In such cases, use the **fgetc** subroutine.

The **fgetc** subroutine performs the same function as the **getc** macro, but **fgetc** is a true subroutine, not a macro. The **fgetc** subroutine runs more slowly than **getc** but takes less disk space.

The **getchar** macro returns the next byte from **stdin** (the standard input stream). The **getchar** macro is equivalent to **getc(stdin)**.

The first successful run of the **fgetc**, **fgets**, **fgetwc**, **fgetws**, **fread**, **fscanf**, **getc**, **getchar**, **gets** or **scanf** subroutine using a stream that returns data not supplied by a prior call to the **ungetc** or **ungetwc** subroutine marks the `st_atime` field for update.

The **getc** and **getchar** macros have also been implemented as subroutines for ANSI compatibility. To access the subroutines instead of the macros, insert **#undef getc** or **#undef getchar** at the beginning of the source file.

The **getw** subroutine returns the next word (**int**) from the input specified by the *Stream* parameter and increments the associated file pointer, if defined, to point to the next word. The size of a word varies from one machine architecture to another. The **getw** subroutine returns the constant **EOF** at the end of the file or when an error occurs. Since **EOF** is a valid integer value, the **feof** and **ferror** subroutines should be used to check the success of **getw**. The **getw** subroutine assumes no special alignment in the file.

Because of additional differences in word length and byte ordering from one machine architecture to another, files written using the **putw** subroutine are machine–dependent and may not be readable using the **getw** macro on a different type of processor.

Parameters

Stream	Points to the file structure of an open file.

Return Values

Upon successful completion, the **getc**, **fgetc**, **getchar**, and **getw** subroutines return the next byte or **int** data type from the input stream pointed by the *Stream* parameter. If the stream is at the end of the file, an end–of–file indicator is set for the stream and the integer constant **EOF** is returned. If a read error occurs, the **errno** global variable is set to reflect the error, and a value of **EOF** is returned. The **ferror** and **feof** subroutines should be used to distinguish between the end of the file and an error condition.

Error Codes

If the stream specified by the *Stream* parameter is unbuffered or data needs to be read into the stream's buffer, the **getc**, **getchar**, **fgetc**, or **getw** subroutine is unsuccessful under the following error conditions:

EAGAIN Indicates that the **O_NONBLOCK** flag is set for the file descriptor underlying the stream specified by the *Stream* parameter. The process would be delayed in the **fgetc** subroutine operation.

EBADF Indicates that the file descriptor underlying the stream specified by the *Stream* parameter is not a valid file descriptor opened for reading.

EFBIG Indicates that an attempt was made to read a file that exceeds the process' file–size limit or the maximum file size. See the **ulimit** subroutine.

EINTR Indicates that the read operation was terminated due to the receipt of a signal, and either no data was transferred, or the implementation does not report partial transfer for this file.

Note: Depending upon which library routine the application binds to, this subroutine may return **EINTR**. Refer to the **signal** subroutine regarding **sa_restart**.

EIO Indicates that a physical error has occurred, or the process is in a background process group attempting to perform a **read** subroutine call from its controlling terminal, and either the process is ignoring (or blocking) the **SIGTTIN** signal or the process group is orphaned.

EPIPE Indicates that an attempt is made to read from a pipe or first–in–first–out (FIFO) that is not open for reading by any process. A **SIGPIPE** signal will also be sent to the process.

EOVERFLOW Indicates that the file is a regular file and an attempt was made to read at or beyond the offset maximum associated with the corresponding stream.

The **getc**, **getchar**, **fgetc**, or **getw** subroutine is also unsuccessful under the following error conditions:

ENOMEM Indicates insufficient storage space is available.

ENXIO Indicates either a request was made of a nonexistent device or the request was outside the capabilities of the device.

Implementation Specifics

These subroutines are part of Base Operating System (BOS) Runtime.

Related Information

The **feof**, **ferror**, **clearerr**, or **fileno** subroutine, **freopen**, **fopen**, or **fdopen** subroutine, **fread** or **fwrite** subroutine, **getwc**, **fgetwc**, or **getwchar** subroutine, **get** or **fgets** subroutine, **putc**, **putchar**, **fputc**, or **putw** subroutine, **scanf**, **sscanf**, **fscanf**, or **wsscanf** subroutine.

List of Character Manipulation Services, Subroutines Overview in *AIX General Programming Concepts : Writing and Debugging Programs*.

getc_unlocked, getchar_unlocked, putc_unlocked, putchar_unlocked Subroutines

Purpose

stdio with explicit client locking.

Library

Standard Library (**libc.a**)

Syntax

```
#include <stdio.h>

int getc_unlocked (FILE * stream);
int getchar_unlocked (void);
int putc_unlocked (int c, FILE * stream);
int putchar_unlocked (int c);
```

Description

Versions of the functions **getc**, **getchar**, **putc**, and **putchar** respectively named **getc_unlocked**, **getchar_unlocked**, **putc_unlocked**, and **putchar_unlocked** are provided which are functionally identical to the original versions with the exception that they are not required to be implemented in a thread–safe manner. They may only safely be used within a scope protected by **flockfile** (or **ftrylockfile**) and **funlockfile.** These functions may safely be used in a multi–threaded program if and only if they are called while the invoking thread owns the (FILE*) object, as is the case after a successful call of the **flockfile** or **ftrylockfile** functions.

Return Values

See **getc**, **getchar**, **putc**, and **putchar**.

Implementation Specifics

These subroutines are part of Base Operating System (BOS) subroutine.

Related Information

The **getc** subroutine.

The **getchar** subroutine.

The **putc** subroutine.

The **putchar** subroutine.

The **stdio.h** file.

getconfattr Subroutine

Purpose

Accesses the user information in the user database.

Library

Security Library (**libc.a**)

Syntax

```
#include <usersec.h>
#include <userconf.h>

int getconfattr (sys, Attribute, Value, Type)
char *sys;
char *Attribute;
void *Value;
int Type;
```

Description

The **getconfattr** subroutine reads a specified attribute from the user database.

Parameters

sys System attribute. The following possible attributes are defined in the **userconf.h** file.

- SC_SYS_LOGIN
- SC_SYS_USER
- SC_SYS_ADMUSER
- SC_SYS_AUDIT SEC_LIST
- SC_SYS_AUSERS SEC_LIST
- SC_SYS_ASYS SEC_LIST
- SC_SYS_ABIN SEC_LIST
- SC_SYS_ASTREAM SEC_LIST

Attribute Specifies which attribute is read. The following possible attributes are defined in the **usersec.h** file:

S_ID User ID. The attribute type is **SEC_INT**.

S_PGRP Principle group name. The attribute type is **SEC_CHAR**.

S_GROUPS Groups to which the user belongs. The attribute type is **SEC_LIST**.

S_ADMGROUPS Groups for which the user is an administrator. The attribute type is **SEC_LIST**.

S_ADMIN Administrative status of a user. The attribute type is **SEC_BOOL**.

S_AUDITCLASSES Audit classes to which the user belongs. The attribute type is **SEC_LIST**.

S_AUTHSYSTEM	Defines the user's authentication method. The attribute type is **SEC_CHAR.**
S_HOME	Home directory. The attribute type is **SEC_CHAR**.
S_SHELL	Initial program run by a user. The attribute type is **SEC_CHAR**.
S_GECOS	Personal information for a user. The attribute type is **SEC_CHAR**.
S_USRENV	User–state environment variables. The attribute type is **SEC_LIST**.
S_SYSENV	Protected–state environment variables. The attribute type is **SEC_LIST**.
S_LOGINCHK	Specifies whether the user account can be used for local logins. The attribute type is **SEC_BOOL**.
S_HISTEXPIRE	Defines the period of time (in weeks) that a user cannot reuse a password. The attribute type is **SEC_INT**.
S_HISTSIZE	Specifies the number of previous passwords that the user cannot reuse. The attribute type is **SEC_INT**.
S_MAXREPEAT	Defines the maximum number of times a user can repeat a character in a new password. The attribute type is **SEC_INT**.
S_MINAGE	Defines the minimum age in weeks that the user's password must exist before the user can change it. The attribute type is **SEC_INT**.
S_PWDCHECKS	Defines the password restriction methods for this account. The attribute type is **SEC_LIST**.
S_MINALPHA	Defines the minimum number of alphabetic characters required in a new user's password. The attribute type is **SEC_INT**.
S_MINDIFF	Defines the minimum number of characters required in a new password that were not in the old password. The attribute type is **SEC_INT**.
S_MINLEN	Defines the minimum length of a user's password. The attribute type is **SEC_INT**.
S_MINOTHER	Defines the minimum number of non–alphabetic characters required in a new user's password. The attribute type is **SEC_INT**.
S_DICTIONLIST	Defines the password dictionaries for this account. The attribute type is **SEC_LIST**.
S_SUCHK	Specifies whether the user account can be accessed with the **su** command. Type **SEC_BOOL**.
S_REGISTRY	Defines the user's authentication registry. The attribute type is **SEC_CHAR**.

S_RLOGINCHK Specifies whether the user account can be used for remote logins using the **telnet** or **rlogin** commands. The attribute type is **SEC_BOOL**.

S_DAEMONCHK Specifies whether the user account can be used for daemon execution of programs and subsystems using the **cron** daemon or **src**. The attribute type is **SEC_BOOL**.

S_TPATH Defines how the account may be used on the trusted path. The attribute type is **SEC_CHAR**. This attribute must be one of the following values:

nosak The secure attention key is not enabled for this account.

notsh The trusted shell cannot be accessed from this account.

always This account may only run trusted programs.

on Normal trusted–path processing applies.

S_TTYS List of ttys that can or cannot be used to access this account. The attribute type is **SEC_LIST**.

S_SUGROUPS Groups that can or cannot access this account. The attribute type is **SEC_LIST**.

S_EXPIRATION Expiration date for this account, in seconds since the epoch. The attribute type is **SEC_CHAR**.

S_AUTH1 Primary authentication methods for this account. The attribute type is **SEC_LIST**.

S_AUTH2 Secondary authentication methods for this account. The attribute type is **SEC_LIST**.

S_UFSIZE Process file size soft limit. The attribute type is **SEC_INT**.

S_UCPU Process CPU time soft limit. The attribute type is **SEC_INT**.

S_UDATA Process data segment size soft limit. The attribute type is **SEC_INT**.

S_USTACK Process stack segment size soft limit. Type: **SEC_INT**.

S_URSS Process real memory size soft limit. Type: **SEC_INT**.

S_UCORE Process core file size soft limit. The attribute type is **SEC_INT**.

S_PWD Specifies the value of the `passwd` field in the **/etc/passwd** file. The attribute type is **SEC_CHAR**.

S_UMASK File creation mask for a user. The attribute type is **SEC_INT**.

S_LOCKED Specifies whether the user's account can be logged into. The attribute type is **SEC_BOOL**.

S_UFSIZE_HARD Process file size hard limit. The attribute type is **SEC_INT**.

S_UCPU_HARD Process CPU time hard limit. The attribute type is **SEC_INT**.

S_UDATA_HARD Process data segment size hard limit. The attribute type is **SEC_INT**.

S_USTACK_HARD
Process stack segment size hard limit. Type: **SEC_INT**.

S_URSS_HARD Process real memory size hard limit. Type: **SEC_INT**.

S_UCORE_HARD Process core file size hard limit. The attribute type is **SEC_INT**.

Note: These values are string constants that should be used by applications both for convenience and to permit optimization in latter implementations.

Type Specifies the type of attribute expected. Valid types are defined in the **usersec.h** file and include:

SEC_INT The format of the attribute is an integer.

For the **getuserattr** subroutine, the user should supply a pointer to a defined integer variable. For the **putuserattr** subroutine, the user should supply an integer.

SEC_CHAR The format of the attribute is a null–terminated character string.

SEC_LIST The format of the attribute is a series of concatenated strings, each null–terminated. The last string in the series is terminated by two successive null characters.

SEC_BOOL The format of the attribute from **getuserattr** is an integer with the value of either 0 (false) or 1 (true). The format of the attribute for **putuserattr** is a null–terminated string containing one of the following strings: true, false, yes, no, always, or never.

SEC_COMMIT For the **putuserattr** subroutine, this value specified by itself indicates that changes to the named user are to be committed to permanent storage. The *Attribute* and *Value* parameters are ignored. If no user is specified, the changes to all modified users are committed to permanent storage.

SEC_DELETE The corresponding attribute is deleted from the database.

SEC_NEW Updates all the user database files with the new user name when using the **putuserattr** subroutine.

Security

Files Accessed:

Mode	File
rw	/etc/security/user
rw	/etc/security/limits
rw	/etc/security/login.cfg

Return Values

If successful, returns 0

If successful, returns –1

Error Codes

ENOENT	The specified User parameter does not exist or the attribute is not defined for this user.

Implementation Specifics

These subroutines are part of Base Operating System (BOS) Runtime.

Files

/etc/passwd	Contains user IDs.

Related Information

The **getuserattr** subroutine.

List of Security and Auditing Subroutines, Subroutines Overview in *AIX General Programming Concepts : Writing and Debugging Programs*.

getcontext or setcontext Subroutine

Purpose

Initializes the structure pointed to by *ucp* to the context of the calling process.

Library

(**libc.a**)

Syntax

#include <ucontext.h>

int getcontext (ucontext_t **ucp***);**

int setcontext (const uncontext_t **ucp*);

Description

The **getcontext** subroutine initalizes the structure pointed to by ucp to the current user context of the calling process. The **ucontext_t** type that ucp points to defines the user context and includes the contents of the calling process' machine registers, the signal mask, and the current execution stack.

The setcontext subroutine restores the user context pointed to by ucp. A successful call to setcontext subroutine does not return; program execution resumes at the point specified by the upc argument passed to setcontext subroutine. The ucp argument should be created either by a prior call to getcontext subroutine, or by being passed as an argument to a signal handler. If the ucp argument was created with getcontext subroutine, program execution continues as if the corresponding call of getcontext subroutine had just returned. If the ucp argument was created with makecontext subroutine, program execution continues with the function passed to makecontext subroutine. When that function returns, the process continues as if after a call to setcontext subroutine with the ucp argument that was input to makecontext subroutine. If the ucp argument was passed to a signal handler, program execution continues with the program instruction following the instruction interrupted by the signal. If the uc_link member of the ucontext_t structure pointed to by the ucp arguement is equal to 0, then this context is the main context, and the process will exit when this context returns.

Parameters

ucp	A pointer to a user stucture.

Return Values

Upon successful completion, setcontext subroutine does not return and getcontext subroutine returns 0. Otherwise, a value –1 is returned.

–1	Not successful and the **errno** global variable is set to one of the following error codes.

Related Information

The **makecontext** subroutine, **setjmp** subroutine, **sigaltstack** subroutine, **sigaction** subroutine, **sigprocmask** subroutine, and **sigsetjmp** subroutine.

getcwd Subroutine

Purpose

Gets the path name of the current directory.

Library

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

Syntax

```
#include <unistd.h>

char *getcwd (Buffer, Size)
char *Buffer;
size_t Size;
```

Description

The **getcwd** subroutine places the absolute path name of the current working directory in the array pointed to by the *Buffer* parameter, and returns that path name. The *size* parameter specifies the size in bytes of the character array pointed to by the *Buffer* parameter.

Parameters

Buffer	Points to string space that will contain the path name. If the *Buffer* parameter value is a null pointer, the **getcwd** subroutine, using the **malloc** subroutine, obtains the number of bytes of free space as specified by the *Size* parameter. In this case, the pointer returned by the **getcwd** subroutine can be used as the parameter in a subsequent call to the **free** subroutine. Starting the **getcwd** subroutine with a null pointer as the *Buffer* parameter value is not recommended.
Size	Specifies the length of the string space. The value of the *Size* parameter must be at least 1 greater than the length of the path name to be returned.

Return Values

If the **getcwd** subroutine is unsuccessful, a null value is returned and the **errno** global variable is set to indicate the error. The **getcwd** subroutine is unsuccessful if the *Size* parameter is not large enough or if an error occurs in a lower–level function.

Error Codes

If the **getcwd** subroutine is unsuccessful, it returns one or more of the following error codes:

EACCES	Indicates that read or search permission was denied for a component of the path name
EINVAL	Indicates that the *Size* parameter is 0 or a negative number.
ENOMEM	Indicates that insufficient storage space is available.
ERANGE	Indicates that the *Size* parameter is greater than 0, but is smaller than the length of the path name plus 1.

Implementation Specifics

This subroutine is part of Base Operating System (BOS) Runtime.

Related Information

The **getwd** subroutine, **malloc** subroutine.

Files, Directories, and File Systems for Programmers in *AIX General Programming Concepts : Writing and Debugging Programs*.

getdate Subroutine

Purpose

Convert user format date and time.

Library

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

Syntax

```
#include <time.h>

struct tm *getdate (const char *string)

extern int getdate_err
```

Description

The **getdate** subroutine converts user definable date and/or time specifications pointed to by *string*, into a **struct tm**. The structure declaration is in the **time.h** header file (see **ctime** subroutine).

User supplied templates are used to parse and interpret the input string. The templates are contained in text files created by the user and identified by the environment variable **DATEMSK**. The **DATEMSK** variable should be set to indicate the full pathname of the file that contains the templates. The first line in the template that matches the input specification is used for interpretation and conversation into the internal time format.

The following field descriptors are supported:

%%	Same as %.
%a	Abbreviated weekday name.
%A	Full weekday name.
%b	Abbreviated month name.
%B	Full month name.
%c	Locale's appropriate date and time representation.
%C	Century number (00–99; leading zeros are permitted but not required)
%d	Day of month (01 – 31: the leading zero is optional.
%e	Same as %d.
%D	Date as %m/%d/%y.
%h	Abbreviated month name.
%H	Hour (00 – 23)
%I	Hour (01 – 12)
%m	Month number (01 – 12)
%M	Minute (00 – 59)
%n	Same as \n.
%p	Locale's equivalent of either AM or PM.
%r	Time as %I:%M:%S %p
%R	Time as %H: %M
%S	Seconds (00 – 61) Leap seconds are allowed but are not predictable through use of algorithms.

%t Same as tab.

%T Time as %H: %M:%S

%w Weekday number (Sunday = 0 – 6)

%x Locale's appropriate date representation.

%X Locale's appropriate time representation.

%y Year within century.

Note: When the environment variable **XPG_TIME_FMT=ON**, **%y** is the year within the century. When a century is not otherwise specified, values in the range 69–99 refer to years in the twentieth century (1969 to 1999, inclusive); values in the range 00–68 refer to 2000 to 2068, inclusive.

%Y Year as ccyy (such as 1986)

%Z Time zone name or no characters if no time zone exists. If the time zone supplied by %Z is not the same as the time zone **getdate** subroutine expects, an invalid input specification error will result. The **getdate** subroutine calculates an expected time zone based on information supplied to the interface (such as hour, day, and month).

The match between the template and input specification performed by the **getdate** subroutine is case sensitive.

The month and weekday names can consist of any combination of upper and lower case letters. The used can request that the input date or time specification be in a specific language by setting the LC_TIME category (See the **setlocale** subroutine).

Leading zero's are not necessary for the descriptors that allow leading zero's. However, at most two digits are allowed for those descriptors, including leading zero's. Extra whitespace in either the template file or in *string* is ignored.

The field descriptors %c, %x, and %X will not be supported if they include unsupported field descriptors.

Example 1 is an example of a template. Example 2 contains valid input specifications for the template. Example 3 shows how local date and time specifications can be defined in the template.

The following rules apply for converting the input specification into the internal format:

- If only the weekday is given, today is assumed if the given month is equal to the current day and next week if it is less.
- If only the month is given, the current month is assumed if the given month is equal to the current month and next year if it is less and no year is given (the first day of month is assumed if no day is given).
- If no hour, minute, and second are given, the current hour, minute and second are assumed.
- If no date is given, today is assumed if the given hour is greater than the current hour and tomorrow is assumed if it is less.

See Example 4 for examples illustrating the use of the above rules.

Return Values

Upon successful completion, the **getdate** subroutine returns a pointer to **struct tm;** otherwise, it returns a null pointer and the external variable **getdate_err** is set to indicate the error.

Error Codes

Upon failure, a null pointer is returned and the variable **getdate_err** is set to indicate the error.

The following is a complete list of the **getdate_err** settings and their corresponding descriptions:

1	The **DATEMSK** environment variable is null or undefined.
2	The template file cannot be opened for reading.
3	Failed to get file status information.
4	The template file is not a regular file.
5	An error is encountered while reading the template file.
6	Memory allocation failed (not enough memory available.
7	There is no line in the template that matches the input.
8	Invalid input specification, Example: February 31 or a time is specified that can not be represented in a time_t (representing the time in seconds since 00:00:00 UTC, January 1, 1970).

Examples

1. The following example shows the possible contents of a template:

```
%m
%A %B %d, %Y, %H:%M:%S
%A
%B
%m/%d/%y %I %p
%d, %m, %Y %H:%M
at %A the %dst of %B in %Y
run job at %I %p, %B %dnd
&A den %d. %B %Y %H.%M Uhr
```

2. The following are examples of valid input specifications for the template in Example 1:

```
getdate (”10/1/87 4 PM”)
getdate (”Friday”)
getdate (”Friday September 18, 1987, 10:30:30”)
getdate (”24,9,1986 10:30”)
getdate (”at monday the 1st of december in 1986”)
getdate (”run job at 3 PM. december 2nd”)
```

If the LC_TIME category is set to a German locale that includes `freitag` as a weekday name and `oktober` as a month name, the following would be valid:

```
getdate (”freitag den 10. oktober 1986 10.30 Uhr”)
```

3. The following examples shows how local date and time specification can be defined in the template.

Invocation	**Line in Template**
getdate (”11/27/86”)	%m/%d/%y
getdate (”27.11.86”0	%d.%m.%y
getdate (”86–11–27”)	%y–%m–%d
getdate (”Friday 12:00:00”)	%A %H:%M:%S

4. The following examples help to illustrate the above rules assuming that the current date Mon Sep 22 12:19:47 EDT 1986 and the LC_TIME category is set to the default ”C” locale.

Input	Line in Template	Date
Mon	%a	Mon Sep 22 12:19:47 EDT 1986
Sun	%a	Sun Sep 28 12:19:47 EDT 1986
Fri	%a	Fri Sep 26 12:19:47 EDT 1986
September	%B	Mon Sep1 12:19:47 EDT 1986
January	%B	Thu Jan 1 12:19:47 EDT 1986
December	%B	Mon Dec 1 12:19:47 EDT 1986
Sep Mon	%b %a	Mon Sep 1 12:19:47 EDT 1986
Jan Fri	%b %a	Fri Jan 2 12:19:47 EDT 1986
Dec Mon	%b %a	Mon Dec 1 12:19:47 EDT 1986
Jan Wed 1989	%b %a %Y	Wed Jan 4 12:19:47 EDT 1986
Fri 9	%a %H	Fri Sep 26 12:19:47 EDT 1986
Feb 10:30	%b %H: %S	Sun Feb 1 12:19:47 EDT 1986
10:30	%H: %M	Tue Sep 23 12:19:47 EDT 1986
13:30	%H: %M	Mon Sep 22 12:19:47 EDT 1986

Implementation Specifics

This subroutine is part of Base Operating System (BOS) Runtime.

Related Information

The **ctime**, **ctype**, **setlocale**, **strftime**, and **times** subroutines.

getdtablesize Subroutine

Purpose

Gets the descriptor table size.

Library

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

Syntax

```
#include <unistd.h>

int getdtablesize (void)
```

Description

The **getdtablesize** subroutine is used to determine the size of the file descriptor table.

The size of the file descriptor table for a process is set by the **ulimit** command or by the **setrlimit** subroutine. The **getdtablesize** subroutine returns the current size of the table as reported by the **getrlimit** subroutine. If **getrlimit** reports that the table size is unlimited, **getdtablesize** instead returns the value of OPEN_MAX, which is the largest possible size of the table.

Note: The **getdtablesize** subroutine returns a runtime value that is specific to the version of AIX on which the application is running. In AIX 4.3.1, **getdtablesize** returns a value that is set in the **limits** file, which can be different from system to system.

Return Values

The **getdtablesize** subroutine returns the size of the descriptor table.

Implementation Specifics

This subroutine is part of Base Operating System (BOS) Runtime.

Related Information

The **close** subroutine, **open** subroutine, **select** subroutine.

getenv Subroutine

Purpose

Returns the value of an environment variable.

Library

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

Syntax

```
#include <stdlib.h>

char *getenv (Name)
const char *Name;
```

Description

The **getenv** subroutine searches the environment list for a string of the form *Name***=***Value*. Environment variables are sometimes called shell variables because they are frequently set with shell commands.

Parameters

Name	Specifies the name of an environment variable. If a string of the proper form is not present in the current environment, the **getenv** subroutine returns a null pointer.

Return Values

The **getenv** subroutine returns a pointer to the value in the current environment, if such a string is present. If such a string is not present, a null pointer is returned. The **getenv** subroutine normally does not modify the returned string. The **putenv** subroutine, however, may overwrite or change the returned string. Do not attempt to free the returned pointer. The **getenv** subroutine returns a pointer to the user's copy of the environment (which is static), until the first invocation of the **putenv** subroutine that adds a new environment variable. The **putenv** subroutine allocates an area of memory large enough to hold both the user's environment and the new variable. The next call to the **getenv** subroutine returns a pointer to this newly allocated space that is not static. Subsequent calls by the **putenv** subroutine use the **realloc** subroutine to make space for new variables. Unsuccessful completion returns a null pointer.

Implementation Specifics

This subroutine is part of Base Operating System (BOS) Runtime.

Related Information

The **putenv** subroutine.

getevars Subroutine

Purpose

Gets environment of a process.

Library

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

Syntax

```
#include <procinfo.h>
#include <sys/types.h>

int getevars (processBuffer, bufferLen, argsBuffer, argsLen)
struct procsinfo *processBuffer;
int bufferLen;
char *argsBuffer;
int argsLen;
```

Description

The **getevars** subroutine returns the environment that was passed to a command when it was started. Only one process can be examined per call to **getevars**.

The **getevars** subroutine uses the pi_pid field of *processBuffer* to determine which process to look for. *bufferLen* should be set to size of **struct procsinfo**. Parameters are returned in *argsBuffer*, which should be allocated by the caller. The size of this array must be given in *argsLen*.

On return, *argsBuffer* consists of a succession of strings, each terminated with a null character (ascii '\0'). Hence, two consecutive `NULL`s indicate the end of the list.

Note: The arguments may be changed asynchronously by the process, but results are not guaranteed to be consistent.

Parameters

processBuffer	Specifies the address of a **procsinfo** structure, whose pi_pid field should contain the pid of the process that is to be looked for.
bufferLen	Specifies the size of a single **procsinfo** structure,
argsBuffer	Specifies the address of an array of characters to be filled with a series of strings representing the parameters that are needed. An extra `NULL` character marks the end of the list. This array must be allocated by the caller.
argsLen	Specifies the size of the *argsBuffer* array. No more than *argsLen* characters are returned.

Return Values

If successful, the **getevars** subroutine returns zero. Otherwise, a value of –1 is returned and the **errno** global variable is set to indicate the error.

Error Codes

The **getevars** subroutine does not succeed if the following are true:

EBADF	The specified process does not exist.
EFAULT	The copy operation to the buffer was not successful or the *processBuffer* or *argsBuffer* parameters are invalid.
EINVAL	The *bufferLen* parameter does not contain the size of a single **procsinfo** structure.

Implementation Specifics

This subroutine is part of the Base Operating System (BOS) Runtime.

Related Information

The **getargs**, **getpid**, **getpgrp**, **getppid**, or **getthrds** subroutines.

The **ps** command.

getfsent, getfsspec, getfsfile, getfstype, setfsent, or endfsent Subroutine

Purpose

Gets information about a file system.

Library

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

Syntax

```
#include <fstab.h>

struct fstab *getfsent( )

struct fstab *getfsspec (Special)
char *Special;

struct fstab *getfsfile(File)
char *File;

struct fstab *getfstype(Type)
char *Type;

void setfsent( )

void endfsent( )
```

Description

The **getfsent** subroutine reads the next line of the **/etc/filesystems** file, opening the file if necessary.

The **setfsent** subroutine opens the **/etc/filesystems** file and positions to the first record.

The **endfsent** subroutine closes the **/etc/filesystems** file.

The **getfsspec** and **getfsfile** subroutines sequentially search from the beginning of the file until a matching special file name or file–system file name is found, or until the end of the file is encountered. The **getfstype** subroutine does likewise, matching on the file–system type field.

Note: All information is contained in a static area, which must be copied to be saved.

Parameters

Special	Specifies the file–system file name.
File	Specifies the file name.
Type	Specifies the file–system type.

Return Values

The **getfsent**, **getfsspec**, **getfstype**, and **getfsfile** subroutines return a pointer to a structure that contains information about a file system. The header file **fstab.h** describes the structure. A null pointer is returned when the end of file (EOF) is reached or if an error occurs.

Implementation Specifics

These subroutines are part of Base Operating System (BOS) Runtime.

Files

/etc/filesystems Centralizes file system characteristics.

Related Information

The **getvfsent**, **getvfsbytype**, **getvfsbyname**, **getvfsbyflag**, **setvfsent**, or **endvfsent** subroutine.

The **filesystems** file.

Files, Directories, and File Systems for Programmers in *AIX General Programming Concepts : Writing and Debugging Programs*.

getfsent_r, getfsspec_r, getfsfile_r, getfstype_r, setfsent_r, or endfsent_r Subroutine

Purpose

Gets information about a file system.

Library

Thread–Safe C Library (**libc_r.a**)

Syntax

```
#include <fstab.h>

int getfsent_r (FSSent, FSFile, PassNo)
struct fstab *FSSent;
AFILE_t *FSFile;
int *PassNo;

int getfsspec_r (Special, FSSent, FSFile, PassNo)
const char *Special;
struct fstab *FSSent;
AFILE_t *FSFile;
int *PassNo;

int getfsfile_r (File, FSSent, FSFile, PassNo)
const char *File;
struct fstab *FSSent;
AFILE_t *FSFile;
int *PassNo;

int getfstype_r (Type, FSSent, FSFile, PassNo)
const char *Type;
struct fstab *FSSent;
AFILE_t *FSFile;
int *PassNo;

int setfsent_r (FSFile, PassNo)
AFILE_t * FSFile;
int *PassNo;

int endfsent_r (FSFile)
AFILE_t *FSFile;
```

Description

The **getfsent_r** subroutine reads the next line of the **/etc/filesystems** file, opening it necessary.

The **setfsent_r** subroutine opens the **filesystems** file and positions to the first record.

The **endfsent_r** subroutine closes the **filesystems** file.

The **getfsspec_r** and **getfsfile_r** subroutines search sequentially from the beginning of the file until a matching special file name or file–system file name is found, or until the end of the file is encountered. The **getfstype_r** subroutine behaves similarly, matching on the file–system type field.

Parameters

FSSent	Points to a structure containing information about the file system. The *FSSent* parameter must be allocated by the caller. It cannot be a null value.
FSFile	Points to an attribute structure. The *FSFile* parameter is used to pass values between subroutines.
PassNo	Points to an integer. The **setfsent_r** subroutine initializes the *PassNo* parameter.
Special	Specifies a special file name to search for in the **filesystems** file.
File	Specifies a file name to search for in the **filesystems** file.
Type	Specifies a type to search for in the **filesystems** file.

Return Values

0	Indicates that the subroutine was successful.
–1	Indicates that the subroutine was not successful.

Implementation Specifics

These subroutines are part of Base Operating System (BOS) Runtime.

Programs using this subroutine must link to the **libpthreads.a** library.

Files

/etc/filesystems	Centralizes file–system characteristics.

Related Information

The **getvfsent**, **getvfsbytype**, **getvfsbyname**, **getvfsbyflag**, **setvfsent**, or **endvfsent** subroutine.

The **filesystems** file in *AIX Files Reference*.

List of Multithread Subroutines in *AIX General Programming Concepts : Writing and Debugging Programs*.

getgid or getegid Subroutine

Purpose

Gets the process group IDs.

Library

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

Syntax

```
#include <unistd.h>
#include <sys/types.h>

gid_t getgid (void);

gid_t getegid (void);
```

Description

The **getgid** subroutine returns the real group ID of the calling process.

The **getegid** subroutine returns the effective group ID of the calling process.

Return Values

The **getgid** and **getegid** subroutines return the requested group ID. The **getgid** and **getegid** subroutines are always successful.

Implementation Specifics

These subroutines are part of Base Operating System (BOS) Runtime.

Related Information

The **getgroups** subroutine, **initgroups** subroutine, **setgid** subroutine, **setgroups** subroutine.

The **groups** command, **setgroups** command.

List of Security and Auditing Subroutines and Subroutines Overview in *AIX General Programming Concepts : Writing and Debugging Programs*.

getgrent, getgrgid, getgrnam, setgrent, or endgrent Subroutine

Purpose

Accesses the basic group information in the user database.

Library

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

Syntax

```
#include <sys/types.h>
#include <grp.h>

struct group *getgrent ( );

struct group *getgrgid (GID)
gid_t GID;

struct group *getgrnam (Name)
const char *Name;

void setgrent ( );

void endgrent ( );
```

Description

Attention: The information returned by the **getgrent**, **getgrnam**, and **getgrgid** subroutines is stored in a static area and is overwritten on subsequent calls. You must copy this information to save it.

Attention: These subroutines should not be used with the **getgroupattr** subroutine. The results are unpredictable.

The **setgrent** subroutine opens the user database if it is not already open. Then, this subroutine sets the cursor to point to the first group entry in the database.

The **getgrent**, **getgrnam**, and **getgrgid** subroutines return information about the requested group. The **getgrent** subroutine returns the next group in the sequential search. The **getgrnam** subroutine returns the first group in the database whose name matches that of the *Name* parameter. The **getgrgid** subroutine returns the first group in the database whose group ID matches the *GID* parameter. The **endgrent** subroutine closes the user database.

Note: An ! (exclamation mark) is written into the `gr_passwd` field. This field is ignored and is present only for compatibility with older versions of UNIX.

The Group Structure

The **group** structure is defined in the **grp.h** file and has the following fields:

`gr_name`	Contains the name of the group.
`gr_passwd`	Contains the password of the group. **Note:** This field is no longer used.
`gr_gid`	Contains the ID of the group.
`gr_mem`	Contains the members of the group.

If the Network Information Service (NIS) is enabled on the system, these subroutines attempt to retrieve the group information from the NIS authentication server.

Parameters

GID	Specifies the group ID.
Name	Specifies the group name.
Group	Specifies the basic group information to enter into the user database.

Return Values

If successful, the **getgrent**, **getgrnam**, and **getgrgid** subroutines return a pointer to a valid group structure. Otherwise, a null pointer is returned.

Error Codes

These subroutines fail if one or more of the following are returned:

EIO	Indicates that an input/output (I/O) error has occurred.
EINTR	Indicates that a signal was caught during the **getgrnam** or **getgrgid** subroutine.
EMFILE	Indicates that the maximum number of file descriptors specified by the **OPEN_MAX** value are currently open in the calling process.
ENFILE	Indicates that the maximum allowable number of files is currently open in the system.

To check an application for error situations, set the **errno** global variable to a value of 0 before calling the **getgrgid** subroutine. If the **errno** global variable is set on return, an error occurred.

Implementation Specifics

These subroutines are part of Base Operating System (BOS) Runtime.

File

/etc/group	Contains basic group attributes.

Related Information

List of Security and Auditing Subroutines, Subroutines Overview in *AIX General Programming Concepts : Writing and Debugging Programs*.

getgrgid_r Subroutine

Purpose

Gets a group database entry for a group ID.

Library

Thread–Safe C Library (**libc_r.a**)

Syntax

```
#include <sys/types.h>
#include <grp.h>
int getgrgid_r(gid_t gid,
struct group *grp,
char *buffer,
size_t bufsize,
struct group **result);
```

Description

The **getgrgid_r** subroutine updates the **group** structure pointed to by *grp* and stores a pointer to that structure at the location pointed to by *result*. The structure contains an entry from the group database with a matching *gid*. Storage referenced by the group structure is allocated from the memory provided with the *buffer* parameter, which is *bufsize* characters in size. The maximum size needed for this buffer can be determined with the {_SC_GETGR_R_SIZE_MAX} *sysconf* parameter. A NULL pointer is returned at the location pointed to by *result* on error or if the requested entry is not found.

Return Values

Upon successful completion, **getgrgid_r** returns a pointer to a **struct group** with the structure defined in **<grp.h>** with a matching entry if one is found. The **getgrgid_r** function returns a null pointer if either the requested entry was not found, or an error occurred. On error, *errno* will be set to indicate the error.

The return value points to a static area that is overwritten by a subsequent call to the **getgrent**, **getgrgid**, or **getgrnam** subroutine.

If successful, the **getgrgid_r** function returns zero. Otherwise, an error number is returned to indicate the error.

Error Codes

The **getgrgid_r** function fails if:

ERANGE	Insufficient storage was supplied via *buffer* and *bufsize* to contain the data to be referenced by the resulting **group** structure.

Applications wishing to check for error situations should set *errno* to 0 before calling **getgrgid_r**. If *errno* is set on return, an error occurred.

Implementation Specifics

The **getgrent, getgrgid, getgrnam, setgrent, endgrent** subroutine.

The **<grp.h>**, **<limits.h>**, and **<sys/types.h>** header files.

getgrnam_r Subroutine

Purpose

Search a group database for a name.

Library

Thread–Safe C Library (**libc_r.a**)

Syntax

```
#include <sys/types.h>
#include <grp.h>
int getgrnam_r (const char **name,
struct group *grp,
char *buffer,
size_t bufsize,
struct group **result);
```

Description

The **getgrnam_r** function updates the **group** structure pointed to by *grp* and stores pointer to that structure at the location pointed to by *result.* The structure contains an entry from the group database with a matching *gid* or *name*. Storage referenced by the group structure is allocated from the memory provided with the *buffer* parameter, which is *bufsize* characters in size. The maximum size needed for this buffer can be determined with the {_SC_GETGR_R_SIZE_MAX} *sysconf* parameter. A NULL pointer is returned at the location pointed to by *result* on error or if the requested entry is not found.

Return Values

The **getgrnam_r** function returns a pointer to a **struct group** with the structure defined in **<grp.h>** with a matching entry if one is found. The **getgrnam_r** function returns a null pointer if either the requested entry was not found, or an error occurred. On error, *errno* will be set to indicate the error.

The return value points to a static area that is overwritten by a subsequent call to the **getgrent**, **getgrgid**, or **getgrnam** subroutine.

If successful, the **getgrnam_r** function returns zero. Otherwise, an error number is returned to indicate the error.

Error Codes

The **getgrnam_r** function fails if:

ERANGE	Insufficient storage was supplied via *buffer* and *bufsize* to contain the data to be referenced by the resulting **group** structure.

Applications wishing to check for error situations should set *errno* to 0 before calling **getgrnam_r**. If *errno* is set on return, an error occurred.

Implementation Specifics

The **getgrent, getgrgid, getgrnam, setgrent, endgrent** subroutine.

The **<grp.h>**, **<limits.h>**, and **<sys/types.h>** header files.

getgroupattr, IDtogroup, nextgroup, or putgroupattr Subroutine

Purpose

Accesses the group information in the user database.

Library

Security Library (**libc.a**)

Syntax

```
#include <usersec.h>

int getgroupattr (Group, Attribute, Value, Type)
char *Group;
char *Attribute;
void *Value;
int Type;

int putgroupattr (Group, Attribute, Value, Type)
char *Group;
char *Attribute;
void *Value;
int Type;

char *IDtogroup (GID)
gid_t GID;

char *nextgroup (Mode, Argument)
int Mode, Argument;
```

Description

> **Attention:** These subroutines and the **setpwent** and **setgrent** subroutines should not be used simultaneously. The results can be unpredictable.

These subroutines access group information. Because of their greater granularity and extensibility, you should use them instead of the **getgrent**, **putgrent**, **getgrnam**, **getgrgid**, **setgrent**, and **endgrent** subroutines.

The **getgroupattr** subroutine reads a specified attribute from the group database. If the database is not already open, the subroutine will do an implicit open for reading.

Similarly, the **putgroupattr** subroutine writes a specified attribute into the group database. If the database is not already open, the subroutine does an implicit open for reading and writing. Data changed by **putgroupattr** must be explicitly committed by calling the **putgroupattr** subroutine with a *Type* parameter specifying the **SEC_COMMIT** value. Until the data is committed, only **get** subroutine calls within the process will return the written data.

New entries in the user and group databases must first be created by invoking **putgroupattr** with the **SEC_NEW** type.

The **IDtogroup** subroutine translates a group ID into a group name.

The **nextgroup** subroutine returns the next group in a linear search of the group database. The consistency of consecutive searches depends upon the underlying storage–access mechanism and is not guaranteed by this subroutine.

The **setuserdb** and **enduserdb** subroutines should be used to open and close the user database.

Parameters

Argument — Presently unused and must be specified as null.

Attribute — Specifies which attribute is read. The following possible values are defined in the **usersec.h** file:

- **S_ID** — Group ID. The attribute type is **SEC_INT**.
- **S_USERS** — Members of the group. The attribute type is **SEC_LIST**.
- **S_ADMS** — Administrators of the group. The attribute type is **SEC_LIST**.
- **S_ADMIN** — Administrative status of a group. Type: **SEC_BOOL**.
- **S_GRPEXPORT** — Specifies if the DCE registry can overwrite the local group information with the DCE group information during a DCE export operation. The attribute type is **SEC_BOOL**.

Additional user–defined attributes may be used and will be stored in the format specified by the *Type* parameter.

GID — Specifies the group ID to be translated into a group name.

Group — Specifies the name of the group for which an attribute is to be read.

Mode — Specifies the search mode. Also can be used to delimit the search to one or more user credential databases. Specifying a non–null *Mode* value implicitly rewinds the search. A null mode continues the search sequentially through the database. This parameter specifies one of the following values as a bit mask (defined in the **usersec.h** file):

- **S_LOCAL** — The local database of groups are included in the search.
- **S_SYSTEM** — All credentials servers for the system are searched.

Type — Specifies the type of attribute expected. Valid values are defined in the **usersec.h** file and include:

- **SEC_INT** — The format of the attribute is an integer. The buffer returned by the **getgroupattr** subroutine and the buffer supplied by the **putgroupattr** subroutine are defined to contain an integer.
- **SEC_CHAR** — The format of the attribute is a null–terminated character string.
- **SEC_LIST** — The format of the attribute is a series of concatenated strings, each null–terminated. The last string in the series is terminated by two successive null characters.
- **SEC_BOOL** — A pointer to an integer (**int ***) that has been cast to a null pointer.
- **SEC_COMMIT** — For the **putgroupattr** subroutine, this value specified by itself indicates that changes to the named group are committed to permanent storage. The *Attribute* and *Value* parameters are ignored. If no group is specified, changes to all modified groups are committed to permanent storage.

SEC_DELETE	The corresponding attribute is deleted from the database.
SEC_NEW	If using the **putgroupattr** subroutine, updates all the group database files with the new group name.

Value Specifies the address of a pointer for the **getgroupattr** subroutine. The **getgroupattr** subroutine will return the address of a buffer in the pointer. For the **putgroupattr** subroutine, the *Value* parameter specifies the address of a buffer in which the attribute is stored. See the *Type* parameter for more details.

Security

Files Accessed:

Mode	File
rw	**/etc/group** (write access for **putgroupattr**)
rw	**/etc/security/group** (write access for **putgroupattr**)

Return Values

The **getgroupattr** and **putgroupattr** subroutines, when successfully completed, return a value of 0. Otherwise, a value of –1 is returned and the **errno** global variable is set to indicate the error.

The **IDtogroup** and **nextgroup** subroutines return a character pointer to a buffer containing the requested group name, if successfully completed. Otherwise, a null pointer is returned and the **errno** global variable is set to indicate the error.

Error Codes

Note: All of these subroutines return errors from other subroutines.

These subroutines fail if the following is true:

EACCES	Access permission is denied for the data request.

The **getgroupattr** and **putgroupattr** subroutines fail if one or more of the following are true:

EINVAL	The *Value* parameter does not point to a valid buffer or to valid data for this type of attribute. Limited testing is possible and all errors may not be detected.
EINVAL	The *Group* parameter is null or contains a pointer to a null string.
EINVAL	The *Type* parameter contains more than one of the **SEC_INT**, **SEC_BOOL**, **SEC_CHAR**, **SEC_LIST**, or **SEC_COMMIT** attributes.
EINVAL	The *Type* parameter specifies that an individual attribute is to be committed, and the *Group* parameter is null.
ENOENT	The specified *Group* parameter does not exist or the attribute is not defined for this group.
EPERM	Operation is not permitted.

The **IDtogroup** subroutine fails if the following is true:

ENOENT	The *GID* parameter could not be translated into a valid group name on the system.

The **nextgroup** subroutine fails if one or more of the following are true:

EINVAL	The *Mode* parameter is not null, and does not specify either **S_LOCAL** or **S_SYSTEM**.
EINVAL	The *Argument* parameter is not null.
ENOENT	The end of the search was reached.

Implementation Specifics

These subroutines are part of Base Operating System (BOS) Runtime.

Related Information

The **getuserattr** subroutine, **getuserpw** subroutine, **setpwdb** subroutine, **setuserdb** subroutine.

List of Security and Auditing Subroutines and Subroutines Overview in *AIX General Programming Concepts : Writing and Debugging Programs*.

getgroups Subroutine

Purpose

Gets the supplementary group ID of the current process.

Library

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

Syntax

```
#include <sys/types.h>
#include <unistd.h>

int getgroups (NGroups, GIDSet)
int NGroups;
gid_t GIDSet [ ];
```

Description

The **getgroups** subroutine gets the supplementary group ID of the process. The list is stored in the array pointed to by the *GIDSet* parameter. The *NGroups* parameter indicates the number of entries that can be stored in this array. The **getgroups** subroutine never returns more than the number of entries specified by the **NGROUPS_MAX** constant. (The **NGROUPS_MAX** constant is defined in the **limits.h** file.) If the value in the *NGroups* parameter is 0, the **getgroups** subroutine returns the number of groups in the supplementary group.

Parameters

GIDSet	Points to the array in which the supplementary group ID of the user's process is stored.
NGroups	Indicates the number of entries that can be stored in the array pointed to by the *GIDSet* parameter.

Return Values

Upon successful completion, the **getgroups** subroutine returns the number of elements stored into the array pointed to by the *GIDSet* parameter. If the **getgroups** subroutine is unsuccessful, a value of –1 is returned and the **errno** global variable is set to indicate the error.

Error Codes

The **getgroups** subroutine is unsuccessful if either of the following error codes is true:

EFAULT	The *NGroups* and *GIDSet* parameters specify an array that is partially or completely outside of the allocated address space of the process.
EINVAL	The *NGroups* parameter is smaller than the number of groups in the supplementary group.

Implementation Specifics

This subroutine is part of Base Operating System (BOS) Runtime.

Related Information

The **getgid** subroutine, **initgroups** subroutine, **setgid** subroutine, **setgroups** subroutine.

The **groups** command, **setgroups** command.

List of Security and Auditing Subroutines and Subroutines Overview in *AIX General Programming Concepts : Writing and Debugging Programs*.

getgrpaclattr, nextgrpacl, or putgrpaclattr Subroutine

Purpose

Accesses the group screen information in the SMIT ACL database.

Library

Security Library (**libc.a**)

Syntax

```
#include <usersec.h>

int getgrpaclattr (Group, Attribute, Value, Type)
char *User;
char *Attribute;
void *Value;
int Type;

char *nextgrpacl(void)

int putgrpaclattr (Group, Attribute, Value, Type)
char *User;
char *Attribute;
void *Value;
int Type;
```

Description

The **getgrpaclattr** subroutine reads a specified group attribute from the SMIT ACL database. If the database is not already open, this subroutine does an implicit open for reading.

Similarly, the **putgrpaclattr** subroutine writes a specified attribute into the user SMIT ACL database. If the database is not already open, this subroutine does an implicit open for reading and writing. Data changed by the **putgrpaclattr** subroutine must be explicitly committed by calling the **putgrpaclattr** subroutine with a *Type* parameter specifying **SEC_COMMIT**. Until all the data is committed, only the **getgrpaclattr** subroutine within the process returns written data.

The **nextgrpacl** subroutine returns the next group in a linear search of the group SMIT ACL database. The consistency of consecutive searches depends upon the underlying storage–access mechanism and is not guaranteed by this subroutine.

The **setacldb** and **endacldb** subroutines should be used to open and close the database.

Parameters

Attribute	Specifies which attribute is read. The following possible attributes are defined in the **usersec.h** file: **S_SCREENS** String of SMIT screens. The attribute type is **SEC_LIST**.
Type	Specifies the type of attribute expected. Valid types are defined in the **usersec.h** file and include: **SEC_LIST** The format of the attribute is a series of concatenated strings, each null–terminated. The last string in the series must be an empty (zero character count) string. For the **getgrpaclattr** subroutine, the user should supply a pointer to a defined character pointer variable. For the **putgrpaclattr** subroutine, the user should supply a character pointer. **SEC_COMMIT** For the **putgrpaclattr** subroutine, this value specified by itself indicates that changes to the named group are to be committed to permanent storage. The *Attribute* and *Value* parameters are ignored. If no group is specified, the changes to all modified groups are committed to permanent storage. **SEC_DELETE** The corresponding attribute is deleted from the group SMIT ACL database. **SEC_NEW** Updates the group SMIT ACL database file with the new group name when using the **putgrpaclattr** subroutine.
Value	Specifies a buffer, a pointer to a buffer, or a pointer to a pointer depending on the *Attribute* and *Type* parameters. See the *Type* parameter for more details.

Return Values

If successful, the **getgrpaclattr** returns 0. Otherwise, a value of –1 is returned and the **errno** global variable is set to indicate the error.

Error Codes

Possible return codes are:

EACCES	Access permission is denied for the data request.
ENOENT	The specified *Group* parameter does not exist or the attribute is not defined for this group.
ENOATTR	The specified user attribute does not exist for this group.
EINVAL	The *Attribute* parameter does not contain one of the defined attributes or null.
EINVAL	The *Value* parameter does not point to a valid buffer or to valid data for this type of attribute.
EPERM	Operation is not permitted.

Related Information

The **getgrpaclattr**, **nextgrpacl**, or **putgrpaclattr** subroutine, **setacldb**, or **endacldb** subroutine.

getinterval, incinterval, absinterval, resinc, resabs, alarm, ualarm, getitimer or setitimer Subroutine

Purpose

Manipulates the expiration time of interval timers.

Library

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

Syntax

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

int getinterval (TimerID, Value)
timer_t TimerID;
struct itimerstruc_t *Value;

int incinterval (TimerID, Value, OValue)
timer_t TimerID;
struct itimerstruc_t *Value, *OValue;

int absinterval (TimerID, Value, OValue)
timer_t TimerID;
struct itimerstruc_t *Value, *OValue;

int resabs (TimerID, Resolution, Maximum)
timer_t TimerID;
struct timestruc_t *Resolution, *Maximum;

int resinc (TimerID, Resolution, Maximum)
timer_t TimerID;
struct timestruc_t *Resolution, *Maximum;


#include <unistd.h>

unsigned int alarm (Seconds)
unsigned int Seconds;

useconds_t ualarm (Value, Interval)
useconds_t Value, Interval;

int setitimer (Which, Value, OValue)
int Which;
struct itimerval *Value, *OValue;

int getitimer (Which, Value)
int Which;
struct itimerval *Value;
```

Description

The **getinterval**, **incinterval**, and **absinterval** subroutines manipulate the expiration time of interval timers. These functions use a timer value defined by the **struct itimerstruc_t** structure, which includes the following fields:

```
struct timestruc_t it_interval;   /* timer interval period
*/
struct timestruc_t it_value;      /* timer interval expiration
*/
```

If the `it_value` field is nonzero, it indicates the time to the next timer expiration. If `it_value` is 0, the per–process timer is disabled. If the `it_interval` member is nonzero, it specifies a value to be used in reloading the `it_value` field when the timer expires. If `it_interval` is 0, the timer is to be disabled after its next expiration (assuming `it_value` is nonzero).

The **getinterval** subroutine returns a value from the **struct itimerstruc_t** structure to the *Value* parameter. The `it_value` field of this structure represents the amount of time in the current interval before the timer expires, should one exist for the per–process timer specified in the *TimerID* parameter. The `it_interval` field has the value last set by the **incinterval** or **absinterval** subroutine. The fields of the *Value* parameter are subject to the resolution of the timer.

The **incinterval** subroutine sets the value of a per–process timer to a given offset from the current timer setting. The **absinterval** subroutine sets the value of the per–process timer to a given absolute value. If the specified absolute time has already expired, the **absinterval** subroutine will succeed and the expiration notification will be made. Both subroutines update the interval timer period. Time values smaller than the resolution of the specified timer are rounded up to this resolution. Time values larger than the maximum value of the specified timer are rounded down to the maximum value.

The **resinc** and **resabs** subroutines return the resolution and maximum value of the interval timer contained in the *TimerID* parameter. The resolution of the interval timer is contained in the *Resolution* parameter, and the maximum value is contained in the *Maximum* parameter. These values might not be the same as the values returned by the corresponding system timer, the **gettimer** subroutine. In addition, it is likely that the maximum values returned by the **resinc** and **resabs** subroutines will be different.

Note: If a nonprivileged user attempts to submit a fine granularity timer (that is, a timer request of less than 10 milliseconds), the timer request is raised to 10 milliseconds.

The **alarm** subroutine causes the system to send the calling thread's process a **SIGALRM** signal after the number of real–time seconds specified by the *Seconds* parameter have elapsed. Since the signal is sent to the process, in a multi–threaded process another thread than the one that called the **alarm** subroutine may receive the **SIGALRM** signal. Processor scheduling delays may prevent the process from handling the signal as soon as it is generated. If the value of the *Seconds* parameter is 0, a pending alarm request, if any, is canceled. Alarm requests are not stacked. Only one **SIGALRM** generation can be scheduled in this manner. If the **SIGALRM** signal has not yet been generated, the call results in rescheduling the time at which the **SIGALRM** signal is generated. If several threads in a process call the **alarm** subroutine, only the last call will be effective.

The **ualarm** subroutine sends a **SIGALRM** signal to the invoking process in a specified number of seconds. The **getitimer** subroutine gets the value of an interval timer. The **setitimer** subroutine sets the value of an interval timer.

Parameters

TimerID	Specifies the ID of the interval timer.
Value	Points to a **struct itimerstruc_t** structure.
OValue	Represents the previous time–out period.
Resolution	Resolution of the timer.
Maximum	Indicates the maximum value of the interval timer.
Seconds	Specifies the number of real–time seconds to elapse before the first **SIGALRM** signal.

Interval — Specifies the number of microseconds between subsequent periodic **SIGALRM** signals. If a nonprivileged user attempts to submit a fine granularity timer (that is, a timer request of less than 10 milliseconds), the timer request interval is automatically raised to 10 milliseconds.

Which — Identifies the type of timer. Valid values are:

ITIMER_REAL — Decrements in real time. A **SIGALRM** signal occurs when this timer expires.

ITIMER_VIRTUAL — Decrements in process virtual time. It runs only during process execution. A **SIGVTALRM** signal occurs when it expires.

ITIMER_PROF — Decrements in process virtual time and when the system runs on behalf of the process. It is designed for use by interpreters in statistically profiling the execution of interpreted programs. Each time the **ITIMER_PROF** timer expires, the **SIGPROF** signal occurs. Because this signal may interrupt in–progress system calls, programs using this timer must be prepared to restart interrupted system calls.

Return Values

If these subroutines are successful, a value of 0 is returned. If an error occurs, a value of –1 is returned and the **errno** global variable is set.

The **alarm** subroutine returns the amount of time (in seconds) remaining before the system is scheduled to generate the **SIGALARM** signal from the previous call to **alarm**. It returns a 0 if there was no previous **alarm** request.

The **ualarm** subroutine returns the number of microseconds previously remaining in the alarm clock.

Error Codes

If the **getinterval**, **incinterval**, **absinterval**, **resinc**, **resabs**, **setitimer**, **getitimer**, or **setitimer** subroutine is unsuccessful , a value of –1 is returned and the **errno** global variable is set to one of the following error codes:

EINVAL — Indicates that the *TimerID* parameter does not correspond to an ID returned by the **gettimerid** subroutine, or a value structure specified a nanosecond value less than 0 or greater than or equal to one thousand million (1,000,000,000).

EIO — Indicates that an error occurred while accessing the timer device.

EFAULT — Indicates that a parameter address has referenced invalid memory.

The **alarm** subroutine is always successful. No return value is reserved to indicate an error for it.

Implementation Specifics

These subroutines are part of Base Operating System (BOS) Runtime.

Related Information

The **gettimer** subroutine, **gettimerid** subroutine, **sigaction**, **sigvec**, or **signal** subroutine.

List of Time Data Manipulation Services, Subroutines Overview in *AIX General Programming Concepts : Writing and Debugging Programs*.

Signal Management in *AIX General Programming Concepts : Writing and Debugging Programs* provides more information about signal management in multi–threaded processes.

getlogin Subroutine

Purpose

Gets a user's login name.

Library

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

Syntax

```
include <sys/types.h>
include <unistd.h>
include <limits.h>

char *getlogin (void)
```

Description

Attention: Do not use the **getlogin** subroutine in a multithreaded environment. To access the thread–safe version of this subroutines, see the **getlogin_r** subroutine.

Attention: The **getlogin** subroutine returns a pointer to an area that may be overwritten by successive calls.

The **getlogin** subroutine returns a pointer to the login name in the /**etc**/**utmp** file. You can use the **getlogin** subroutine with the **getpwnam** subroutine to locate the correct password file entry when the same user ID is shared by several login names.

If the **getlogin** subroutine cannot find the login name in the /**etc**/**utmp** file, it returns the process **LOGNAME** environment variable. If the **getlogin** subroutine is called within a process that is not attached to a terminal, it returns the value of the **LOGNAME** environment variable. If the **LOGNAME** environment variable does not exist, a null pointer is returned.

Return Values

The return value can point to static data whose content is overwritten by each call. If the login name is not found, the **getlogin** subroutine returns a null pointer.

Error Codes

If the **getlogin** function is unsuccessful, it returns one or more of the following error codes:

EMFILE	Indicates that the **OPEN_MAX** file descriptors are currently open in the calling process.
ENFILE	Indicates that the maximum allowable number of files is currently open in the system.
ENXIO	Indicates that the calling process has no controlling terminal.

Files

/**etc**/**utmp**	Contains a record of users logged into the system.

Implementation Specifics

This subroutine is part of Base Operating System (BOS) Runtime.

Related Information

The **getgrent**, **getgrgid**, **getgrnam**, **putgrent**, **setgrent**, or **endgrent** subroutine, **getlogin_r** subroutine, **getpwent**, **getpwuid**, **setpwent**, or **endpwent** subroutine, **getpwnam** subroutine.

List of Security and Auditing Subroutines, Subroutines Overview in *AIX General Programming Concepts : Writing and Debugging Programs*.

getlogin_r Subroutine

Purpose

Gets a user's login name.

Library

Thread–Safe C Library (**libc_r.a**)

Syntax

```
int getlogin_r (Name, Length)
char *Name;
size_t Length;
```

Description

The **getlogin_r** subroutine gets a user's login name from the **/etc/utmp** file and places it in the *Name* parameter. Only the number of bytes specified by the *Length* parameter (including the ending null value) are placed in the *Name* parameter.

Applications that call the **getlogin_r** subroutine must allocate memory for the login name before calling the subroutine. The name buffer must be the length of the *Name* parameter plus an ending null value.

If the **getlogin_r** subroutine cannot find the login name in the **utmp** file or the process is not attached to a terminal, it places the **LOGNAME** environment variable in the name buffer. If the **LOGNAME** environment variable does not exist, the *Name* parameter is set to null and the **getlogin_r** subroutine returns a –1.

Parameters

Name	Specifies a buffer for the login name. This buffer should be the length of the *Length* parameter plus an ending null value.
Length	Specifies the total length in bytes of the *Name* parameter. No more bytes than the number specified by the *Length* parameter are placed in the *Name* parameter, including the ending null value.

Return Values

0	Indicates that the subroutine was successful.
–1	Indicates that the subroutine was not successful.

Error Codes

If the **getlogin_r** subroutine does not succeed, it returns one of the following error codes:

EMFILE	Indicates that the **OPEN_MAX** file descriptors are currently open in the calling process.
ENFILE	Indicates that the maximum allowable number of files are currently open in the system.
ENXIO	Indicates that the calling process has no controlling terminal.

Implementation Specifics

This subroutine is part of Base Operating System (BOS) Runtime. Programs using this subroutine must link to the **libpthreads.a** library.

File

/**etc**/**utmp**	Contains a record of users logged into the system.

Related Information

The **getgrent_r**, **getgrgid_r**, **getgrnam_r**, **setgrent_r**, or **endgrent_r** subroutine, **getlogin** subroutine, **getpwent_r**, **getpwnam_r**, **putpwent_r**, **getpwuid_r**, **setpwent_r**, or **endpwent_r** subroutine.

List of Security and Auditing Subroutines, List of Multithread Subroutines, and Subroutines Overview in *AIX General Programming Concepts : Writing and Debugging Programs*.

getnameinfo Subroutine

Purpose Hostname–to–service name translation [given the binary address and port].

Note: This is the reverse functionality of **getaddrinfo**: hostname–to–address translation.

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

Library

Library (**libc.a**)

Syntax

```
#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 first argument, *sa*, 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 argument, *salen*, 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 argument, *host*, copies the hostname associated with the IP address into a buffer. The argument, *hostlen*, provides the length of this buffer. The service name associated with the port number is copied into the buffer pointed to by the argument *serv*. The argument, *servlen*, provides the length of this buffer.

The final argument 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 the DNS and returned.

NI_NOFQDN	If set, return only the hostname portion of the FQDN. If clear, return the FQDN.
NI_NUMERICHOST	If set, return the numeric form of the host address. If clear, return the name.
NI_NAMEREQD	If set, return an error if the host's name cannot be determined. If clear, 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 clear, return the service name.
NI_DGRAM	If set, consider the desired service to be a datagram service, (i.e., call getservbyport with an agrument of **udp**). If clear, consider the desired service to be a stream service (i.e., call getserbyport with an argument of **tcp**).

Return Values

If successful, the strings for hostname and service are copied into host and serv, respectively. If unsuccessful, zero values for either hostlen or servlen will suppress the associated lookup; in this case no data is copied into the applicable buffer.

Related Information

The **getaddrinfo** subroutine, **freeaddrinfo** subroutine, and **gai_strerror** subroutine.**Subroutines Overview** in *AIX Version 4 General Programming Concepts: Writing and Debugging Programs*.

getopt Subroutine

Purpose

Returns the next flag letter specified on the command line.

Library

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

Syntax

```
#include <unistd.h>

int getopt (ArgumentC, ArgumentV, OptionString)
int ArgumentC;
char *const ArgumentV [ ];
const char *OptionString;

extern int Optind;
extern int Optopt;
extern int Opterr;
extern char *Optarg;
```

Description

The **getopt** subroutine helps programs interpret shell–command–line flags that are passed to it. The *ArgumentC* and *ArgumentV* parameters are the argument count and argument array, respectively, as passed to the main program. The *OptionString* parameter is a string of recognized flag letters. If a letter is followed by a : (colon), the flag takes an argument.

The *Optind* parameter indexes the next element of the *ArgumentV* parameter to be processed. It is initialized to 1 and the **getopt** subroutine updates it after calling each element of the *ArgumentV* parameter.

The **getopt** subroutine returns the next flag letter in the *ArgumentV* parameter list that matches a letter in the *OptionString* parameter. If the flag takes an argument, the **getopt** subroutine sets the *Optarg* parameter to point to the argument as follows:

- If the flag was the last letter in the string pointed to by an element of the *ArgumentV* parameter, the *Optarg* parameter contains the next element of the *ArgumentV* parameter and the *Optind* parameter is incremented by 2. If the resulting value of the *Optind* parameter is not less than the *ArgumentC* parameter, this indicates a missing flag argument, and the **getopt** subroutine returns an error message.
- Otherwise, the *Optarg* parameter points to the string following the flag letter in that element of the *ArgumentV* parameter and the *Optind* parameter is incremented by 1.

Parameters

ArgumentC	Specifies the number of parameters passed to the routine.
ArgumentV	Specifies the list of parameters passed to the routine.
OptionString	Specifies a string of recognized flag letters. If a letter is followed by a : (colon), the flag is expected to take a parameter that may or may not be separated from it by white space.
Optind	Specifies the next element of the *ArgumentV* array to be processed.
Optopt	Specifies any erroneous character in the *OptionString* parameter.
Opterr	Indicates that an error has occurred when set to a value other than 0.
Optarg	Points to the next option flag argument.

Return Values

The **getopt** subroutine returns the next flag letter specified on the command line. A value of –1 is returned when all command line flags have been parsed. When the value of the *ArgumentV* **[***Optind***]** parameter is null, ***ArgumentV* **[***Optind***]** is not the **–** (minus) character, or *ArgumentV* **[***Optind***]** points to the "**–**" (minus) string, the **getopt** subroutine returns a value of –1 without changing the value. If *ArgumentV* **[***Optind***]** points to the "**– –**" (double minus) string, the **getopt** subroutine returns a value of –1 after incrementing the value of the *Optind* parameter.

Error Codes

If the **getopt** subroutine encounters an option character that is not specified by the *OptionString* parameter, a **?** (question mark) character is returned. If it detects a missing option argument and the first character of *OptionString* is a **:** (colon), then a **:** (colon) character is returned. If this subroutine detects a missing option argument and the first character of *OptionString* is not a colon, it returns a **?** (question mark). In either case, the **getopt** subroutine sets the *Optopt* parameter to the option character that caused the error. If the application has not set the *Opterr* parameter to 0 and the first character of *OptionString* is not a **:** (colon), the **getopt** subroutine also prints a diagnostic message to standard error.

Examples

The following code fragment processes the flags for a command that can take the mutually exclusive flags **a** and **b**, and the flags **f** and **o**, both of which require parameters.

```
#include <unistd.h>    /*Needed for access subroutine constants*/
main (argc, argv)
int argc;
char **argv;
{
   int c;
   extern int optind;
   extern char *optarg;
   .
   .
   .
   while ((c = getopt(argc, argv, "abf:o:")) != EOF)

   {
      switch (c)
      {
         case 'a':
            if (bflg)
               errflg++;
            else
               aflg++;
            break;

         case 'b':
            if (aflg)
               errflg++;
            else
               bflg++;
            break;

         case 'f':
            ifile = optarg;
            break;

         case 'o':
            ofile = optarg;
            break;
```

```
            case '?':
               errflg++;
         } /* case */
         if (errflg)
         {
            fprintf(stderr, "usage: . . . ");
            exit(2);
         }
      } /* while */
      for ( ; optind < argc; optind++)
      {
         if (access(argv[optind], R_OK))
         {
            .
            .
            .
         }
      } /* for */
   }   /* main */
```

Implementation Specifics

This subroutine is part of Base Operating System (BOS) Runtime.

Related Information

The **getopt** command.

List of Executable Program Creation Subroutines, Subroutines Overview, and List of Multithread Subroutines in *AIX General Programming Concepts : Writing and Debugging Programs*.

getpagesize Subroutine

Purpose

Gets the system page size.

Library

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

Syntax

```
#include <unistd.h>

int getpagesize( )
```

Description

The **getpagesize** subroutine returns the number of bytes in a page. Page granularity is the granularity for many of the memory management calls.

The page size is determined by the system and may not be the same as the underlying hardware page size.

Implementation Specifics

This subroutine is part of Base Operating System (BOS) Runtime.

Related Information

The **brk** or **sbrk** subroutine.

The **pagesize** command.

Program Address Space Overview and Subroutines Overview in *AIX General Programming Concepts : Writing and Debugging Programs*.

getpass Subroutine

Purpose

Reads a password.

Library

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

Syntax

```
#include <stdlib.h>

char *getpass (Prompt)
char *Prompt;
```

Description

Attention: The characters are returned in a static data area. Subsequent calls to this subroutine overwrite the static data area.

The **getpass** subroutine does the following:

- Opens the controlling terminal of the current process.
- Writes the characters specified by the *Prompt* parameter to that device.
- Reads from that device the number of characters up to the value of the **PASS_MAX** constant until a new–line or end–of–file (EOF) character is detected.
- Restores the terminal state and closes the controlling terminal.

During the read operation, character echoing is disabled.

The **getpass** subroutine is not safe in a multithreaded environment. To use the **getpass** subroutine in a threaded application, the application must keep the integrity of each thread.

Parameters

Prompt	Specifies a prompt to display on the terminal.

Return Values

If this subroutine is successful, it returns a pointer to the string. If an error occurs, the subroutine returns a null pointer and sets the **errno** global variable to indicate the error.

Error Codes

If the **getpass** subroutine is unsuccessful, it returns one or more of the following error codes:

EINTR	Indicates that an interrupt occurred while the **getpass** subroutine was reading the terminal device. If a **SIGINT** or **SIGQUIT** signal is received, the **getpass** subroutine terminates input and sends the signal to the calling process.
ENXIO	Indicates that the process does not have a controlling terminal.

Note: Any subroutines called by the **getpass** subroutine may set other error codes.

Implementation Specifics

This subroutine is part of Base Operating System (BOS) Runtime.

Related Information

The **getuserpw** subroutine, **newpass** subroutine.

List of Security and Auditing Subroutines, Subroutines Overview in *AIX General Programming Concepts : Writing and Debugging Programs*.

getpcred Subroutine

Purpose

Reads the current process credentials.

Library

Security Library (**libc.a**)

Syntax

```
#include <usersec.h>

char **getpcred (Which)
int Which;
```

Description

The getpcred subroutine reads the specified process security credentials and returns them in a character buffer. It is the calling application's responsibility to free this memory.

The **getpcred** subroutine reads the specified process security credentials and returns them in a character buffer.

Parameters

Which Specifies which credentials are read. This parameter is a bit mask and can contain one or more of the following values, as defined in the **usersec.h** file:

CRED_RUID Real user name

CRED_LUID Login user name

CRED_RGID Real group name

CRED_GROUPS Supplementary group ID

CRED_AUDIT Audit class of the current process

Note: A process must have root user authority to retrieve this credential. Otherwise, the **getpcred** subroutine returns a null pointer and the **errno** global variable is set to **EPERM**.

CRED_RLIMITS BSD resource limits

Note: Use the **getrlimit** subroutine to control resource consumption.

CRED_UMASK The umask.If the *Which* parameter is null, all credentials are returned.

Return Values

When successful, the **getpcred** subroutine returns a pointer to a string containing the requested values. If the **getpcred** subroutine is unsuccessful, a null pointer is returned and the **errno** global variable is set to indicate the error.

Error Codes

The **getpcred** subroutine fails if either of the following are true:

EINVAL The *Which* parameter contains invalid credentials requests.

EPERM The process does not have the proper authority to retrieve the requested credentials.

Other errors can also be set by any subroutines invoked by the **getpcred** subroutine.

Implementation Specifics

This subroutine is part of Base Operating System (BOS) Runtime.

Related Information

The **ckuseracct** subroutine, **ckuserID** subroutine, **getpenv** subroutine, **setpenv** subroutine, **setpcred** subroutine.

List of Security and Auditing Subroutines, Subroutines Overview in *AIX General Programming Concepts : Writing and Debugging Programs*.

getpenv Subroutine

Purpose

Reads the current process environment.

Library

Security Library (**libc.a**)

Syntax

```
#include <usersec.h>

char **getpenv (Which)
int Which;
```

Description

The **getpenv** subroutine reads the specified environment variables and returns them in a character buffer.

Parameters

Which — Specifies which environment variables are to be returned. This parameter is a bit mask and may contain one or more of the following values, as defined in the **usersec.h** file:

PENV_USR — The normal user–state environment. Typically, the shell variables are contained here.

PENV_SYS — The system–state environment. This data is located in system space and protected from unauthorized access.

All variables are returned by setting the *Which* parameter to logically OR the **PENV_USER** and **PENV_SYSTEM** values.

The variables are returned in a null–terminated array of character pointers in the form `var=val`. The user–state environment variables are prefaced by the string **USRENVIRON:**, and the system–state variables are prefaced with **SYSENVIRON:**. If a user–state environment is requested, the current directory is always returned in the **PWD** variable. If this variable is not present in the existing environment, the **getpenv** subroutine adds it to the returned string.

Return Values

Upon successful return, the **getpenv** subroutine returns the environment values. If the **getpenv** subroutine fails, a null value is returned and the **errno** global variable is set to indicate the error.

Note: This subroutine can partially succeed, returning only the values that the process permits it to read.

Error Codes

The **getpenv** subroutine fails if one or more of the following are true:

EINVAL — The *Which* parameter contains values other than **PENV_USR** or **PENV_SYS**.

Other errors can also be set by subroutines invoked by the **getpenv** subroutine.

Implementation Specifics

This subroutine is part of Base Operating System (BOS) Runtime.

Related Information

The **ckuseracct** subroutine, **ckuserID** subroutine, **getpcred** subroutine, **setpenv** subroutine.

List of Security and Auditing Subroutines, Subroutines Overview in *AIX General Programming Concepts : Writing and Debugging Programs*.

getpgid Subroutine

Purpose

Returns the process group ID of the calling process.

Library

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

Syntax

#include <unistd.h>

```
pid_t getpgid (Pid)
(pid_ Pid)
```

Description

The **getpgid** subroutine returns the process group ID of the process whose process ID is equal to that specified by the *Pid* parameter. If the value of the *Pid* parameter is equal to **(pid_t)0**, the **getpgid** subroutine returns the process group ID of the calling process.

Parameter

Pid	The process ID of the process to return the process group ID for.

Return Values

id	The process group ID of the requested process
–1	Not successful and **errno** set to one of the following.

Error Code

ESRCH	There is no process with a process ID equal to *Pid*.
EPERM	The process whose process ID is equal to *Pid* is not in the same session as the calling process.
EINVAL	The value of the *Pid* argument is invalid.

Related Information

The **exec** subroutine, **fork** subroutine, **getpid** subroutine, **getsid** subroutine, **setpgid** subroutine, **setsid** subroutine.

getpid, getpgrp, or getppid Subroutine

Purpose

Returns the process ID, process group ID, and parent process ID.

Syntax

```
#include <unistd.h>

pid_t getpid (void)

pid_t getpgrp (void)

pid_t getppid (void)
```

Description

The **getpid** subroutine returns the process ID of the calling process.

The **getpgrp** subroutine returns the process group ID of the calling process.

The **getppid** subroutine returns the process ID of the calling process' parent process.

Implementation Specifics

These subroutines are part of Base Operating System (BOS) Runtime.

Related Information

The **exec** subroutines, **fork** subroutine, **setpgid** subroutine, **setpgrp** subroutine, **sigaction**, **sigvec**, or **signal** subroutine.

Subroutines Overview in *AIX General Programming Concepts : Writing and Debugging Programs*.

getportattr or putportattr Subroutine

Purpose

Accesses the port information in the port database.

Library

Security Library (**libc.a**)

Syntax

```
#include <usersec.h>

int getportattr (Port, Attribute, Value, Type)
char *Port;
char *Attribute;
void *Value;
int Type;

int putportattr (Port, Attribute, Value, Type)
char *Port;
char *Attribute;
void *Value;
int Type;
```

Description

The **getportattr** or **putportattr** subroutine accesses port information. The **getportattr** subroutine reads a specified attribute from the port database. If the database is not already open, the **getportattr** subroutine implicitly opens the database for reading. The **putportattr** subroutine writes a specified attribute into the port database. If the database is not already open, the **putportattr** subroutine implicitly opens the database for reading and writing. The data changed by the **putportattr** subroutine must be explicitly committed by calling the **putportattr** subroutine with a *Type* parameter equal to the **SEC_COMMIT** value. Until all the data is committed, only these subroutines within the process return the written data.

Values returned by these subroutines are in dynamically allocated buffers. You do not need to move the values prior to the next call.

Use the **setuserdb** or **enduserdb** subroutine to open and close the port database.

Parameters

Port Specifies the name of the port for which an attribute is read.

Attribute Specifies the name of the attribute read. This attribute can be one of the following values defined in the **usersec.h** file:

S_HERALD Defines the initial message printed when the **getty** or **login** command prompts for a login name. This value is of the type **SEC_CHAR**.

S_SAKENABLED Indicates whether or not trusted path processing is allowed on this port. This value is of the type **SEC_BOOL**.

S_SYNONYM Defines the set of ports that are **synonym** attributes for the given port. This value is of the type **SEC_LIST**.

S_LOGTIMES Defines when the user can access the port. This value is of the type **SEC_LIST**.

S_LOGDISABLE Defines the number of unsuccessful login attempts that result in the system locking the port. This value is of the type **SEC_INT**.

S_LOGINTERVAL Defines the time interval in seconds within which **S_LOGDISABLE** number of unsuccessful login attempts must occur before the system locks the port. This value is of the type **SEC_INT**.

S_LOGREENABLE Defines the time interval in minutes after which a system–locked port is unlocked. This value is of the type **SEC_INT**.

S_LOGDELAY Defines the delay factor in seconds between unsuccessful login attempts. This value is of the type **SEC_INT**.

S_LOCKTIME Defines the time in seconds since the epoch (zero time, January 1, 1970) that the port was locked. This value is of the type **SEC_INT**.

S_ULOGTIMES Lists the times in seconds since the epoch (midnight, January 1, 1970) when unsuccessful login attempts occurred. This value is of the type **SEC_LIST**.

Value Specifies the address of a buffer in which the attribute is stored with **putportattr** or is to be read **getportattr**.

Type Specifies the type of attribute expected. The following types are valid and defined in the **usersec.h** file:

SEC_INT Indicates the format of the attribute is an integer. The buffer returned by the **getportattr** subroutine and the buffer supplied by the **putportattr** subroutine are defined to contain an integer.

SEC_CHAR Indicates the format of the attribute is a null–terminated character string.

SEC_LIST Indicates the format of the attribute is a list of null–terminated character strings. The list itself is null terminated.

SEC_BOOL	An integer with a value of either 0 or 1, or a pointer to a character pointing to one of the following strings: – True – Yes – Always – False – No – Never
SEC_COMMIT	Indicates that changes to the specified port are committed to permanent storage if specified alone for the **putportattr** subroutine. The *Attribute* and *Value* parameters are ignored. If no port is specified, changes to all modified ports are committed.
SEC_DELETE	Deletes the corresponding attribute from the database.
SEC_NEW	Updates all of the port database files with the new port name when using the **putportattr** subroutine.

Security

Access Control: The calling process must have access to the port information in the port database.

File Accessed:

Modes	File
rw	/etc/security/login.cfg
rw	/etc/security/portlog

Return Values

The **getportattr** and **putportattr** subroutines return a value of 0 if completed successfully. Otherwise, a value of –1 is returned and the **errno** global value is set to indicate the error.

Error Codes

These subroutines are unsuccessful if the following values are true:

EACCES	Indicates that access permission is denied for the data requested.
ENOENT	Indicates that the *Port* parameter does not exist or the attribute is not defined for the specified port.
ENOATTR	Indicates that the specified port attribute does not exist for the specified port.
EINVAL	Indicates that the *Attribute* parameter does not contain one of the defined attributes or is a null value.
EINVAL	Indicates that the *Value* parameter does not point to a valid buffer or to valid data for this type of attribute.
EPERM	Operation is not permitted.

Implementation Specifics

These subroutines are part of Base Operating System (BOS) Runtime.

Related Information

The **setuserdb** or **enduserdb** subroutine.

List of Security and Auditing Services in *AIX General Programming Concepts : Writing and Debugging Programs*.

Subroutines Overview in *AIX General Programming Concepts : Writing and Debugging Programs*.

getpri Subroutine

Purpose

Returns the scheduling priority of a process.

Library

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

Syntax

```
int getpri (ProcessID)
pid_t ProcessID;
```

Description

The **getpri** subroutine returns the scheduling priority of a process.

Parameters

ProcessID	Specifies the process ID. If this value is 0, the current process scheduling priority is returned.

Return Values

Upon successful completion, the **getpri** subroutine returns the scheduling priority of a thread in the process. Otherwise, a value of –1 is returned and the **errno** global variable is set to indicate the error.

Error Codes

The **getpri** subroutine is unsuccessful if one of the following is true:

EPERM	A process was located, but its effective and real user ID did not match those of the process executing the **getpri** subroutine, and the calling process did not have root user authority.
ESRCH	No process can be found corresponding to that specified by the *ProcessID* parameter.

Implementation Specifics

This subroutine is part of Base Operating System (BOS) Runtime.

Related Information

The **setpri** subroutine.

Subroutines Overview in *AIX General Programming Concepts : Writing and Debugging Programs*.

getpriority, setpriority, or nice Subroutine

Purpose

Gets or sets the nice value.

Libraries

getpriority, **setpriority**: Standard C Library (**libc.a**)

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

Berkeley Compatibility Library (**libbsd.a**)

Syntax

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

int getpriority(Which, Who)
int Which;
int Who;

int setpriority(Which, Who, Priority)
int Which;
int Who;
int Priority;

#include <unistd.h>

int nice(Increment)
int Increment;
```

Description

The nice value of the process, process group, or user, as indicated by the *Which* and *Who* parameters is obtained with the **getpriority** subroutine and set with the **setpriority** subroutine.

The **getpriority** subroutine returns the highest priority nice value (lowest numerical value) pertaining to any of the specified processes. The **setpriority** subroutine sets the nice values of all of the specified processes to the specified value. If the specified value is less than –20, a value of –20 is used; if it is greater than 20, a value of 20 is used. Only processes that have root user authority can lower nice values.

The **nice** subroutine increments the nice value by the value of the *Increment* parameter.

Note: Nice values are only used for the scheduling policy **SCHED_OTHER**, where they are combined with a calculation of recent cpu usage to determine the priority value.

Parameters

Which	Specifies one of **PRIO_PROCESS**, **PRIO_PGRP**, or **PRIO_USER**.
Who	Interpreted relative to the *Which* parameter (a process identifier, process group identifier, and a user ID, respectively). A zero value for the *Who* parameter denotes the current process, process group, or user.
Priority	Specifies a value in the range –20 to 20. Negative nice values cause more favorable scheduling.
Increment	Specifies a value that is added to the current process nice value. Negative values can be specified, although values exceeding either the high or low limit are truncated.

Return Values

On successful completion, the **getpriority** subroutine returns an integer in the range –20 to 20. A return value of –1 can also indicate an error, and in this case the **errno** global variable is set.

On successful completion, the **setpriority** subroutine returns 0. Otherwise, –1 is returned and the global variable **errno** is set to indicate the error.

On successful completion, the **nice** subroutine returns the new nice value minus {NZERO}. Otherwise, a value of –1 is returned and the **errno** global variable is set to indicate the error.

Note: A value of –1 can also be returned. In that case, the calling process should also check the **errno** global variable.

Error Codes

The **getpriority** and **setpriority** subroutines are unsuccessful if one of the following is true:

ESRCH	No process was located using the *Which* and *Who* parameter values specified.
EINVAL	The *Which* parameter was not recognized.

In addition to the errors indicated above, the **setpriority** subroutine is unsuccessful if one of the following is true:

EPERM	A process was located, but neither the effective nor real user ID of the caller of the process executing the **setpriority** subroutine has root user authority.
EACCESS	The call to **setpriority** would have changed the priority of a process to a value lower than its current value, and the effective user ID of the process executing the call did not have root user authority.

The **nice** subroutine is unsuccessful if the following is true:

EPERM	The *Increment* parameter is negative or greater than 2 * {NZERO} and the calling process does not have appropriate privileges.

Implementation Specifics

These subroutines are part of Base Operating System (BOS) Runtime.

To provide upward compatibility with older programs, the **nice** interface, originally found in AT&T System V, is supported.

Note: Process priorities in AT&T System V are defined in the range of 0 to 39, rather than –20 to 20 as in BSD, and the **nice** library routine is supported by both. Accordingly, two versions of the **nice** are supported by Version 3 of the operating system. The default version behaves like the AT&T System V version, with the *Increment* parameter treated as the modifier of a value in the range of 0 to 39 (0 corresponds to –20, 39 corresponds to 9, and priority 20 is not reachable with this interface).

If the behavior of the BSD version is desired, compile with the Berkeley Compatibility Library (**libbsd.a**). The *Increment* parameter is treated as the modifier of a value in the range –20 to 20.

Related Information

The **exec** subroutines.

Subroutines Overview in *AIX General Programming Concepts : Writing and Debugging Programs*.

getprocs Subroutine

Purpose

Gets process table entries.

Library

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

Syntax

```
#include <procinfo.h>
#include <sys/types.h>

int
getprocs (ProcessBuffer, ProcessSize, FileBuffer, FileSize,
IndexPointer, Count)
struct procsinfo *ProcessBuffer;
or struct procsinfo64 *ProcessBuffer;
int ProcessSize;
struct fdsinfo *FileBuffer;
int FileSize;
pid_t *IndexPointer;
int Count;
```

Description

The **getprocs** subroutine returns information about processes, including process table information defined by the **procsinfo** structure, and information about the per–process file descriptors defined by the **fdsinfo** structure.

The **getprocs** subroutine retrieves up to *Count* process table entries, starting with the process table entry corresponding to the process identifier indicated by *IndexPointer*, and places them in the array of **procsinfo** structures indicated by the *ProcessBuffer* parameter. File descriptor information corresponding to the retrieved processes are stored in the array of **fdsinfo** structures indicated by the *FileBuffer* parameter.

On return, the process identifier referenced by *IndexPointer* is updated to indicate the next process table entry to be retrieved. The **getprocs** subroutine returns the number of process table entries retrieved.

The **getprocs** subroutine is normally called repeatedly in a loop, starting with a process identifier of zero, and looping until the return value is less than *Count*, indicating that there are no more entries to retrieve.

Note: The process table may change while the **getprocs** subroutine is accessing it. Returned entries will always be consistent, but since processes can be created or destroyed while the **getprocs** subroutine is running, there is no guarantee that retrieved entries will still exist, or that all existing processes have been retrieved.

When used in 32–bit mode, limits larger than can be represented in 32 bits are truncated to RLIM_INFINITY. Large **rusage** and other values are truncated to INT_MAX. Alternatively, the **struct procsinfo64** and *sizeof* (**struct procsinfo64**) can be used by 32–bit **getprocs** to return full 64–bit process information. Note that the **procsinfo** structure not only increases certain **procsinfo** fields from 32 to 64 bits, but that it contains additional information not present in **procsinfo**. The **struct procsinfo64** contains the same data as **struct procsinfo** when compiled ina 64–bit program.

When used in 64–bit mode, the **struct procsinfo** contains 64–bit **rusage** and **rlimit** structures.

Parameters

ProcessBuffer Specifies the starting address of an array of **procsinfo** or **procsinfo64** structures to be filled in with process table entries. If a value of **NULL** is passed for this parameter, the **getprocs** subroutine scans the process table and sets return values as normal, but no process entries are retrieved.

Note: The *ProcessBuffer* parameter of **getprocs** subroutine contains two struct rusage fields named **pi_ru** and **pi_cru**. Each of these fields contains two struct timeval fields named **ru_utime** and **ru_stime**. The **tv_usec** field in both of the struct timeval contain nanoseconds instead of microseconds. These values cone from the struct user fields named **U_ru** and **U_cru**.

ProcessSize Specifies the size of a single **procsinfo** or **procsinfo64** structure.

FileBuffer Specifies the starting address of an array of **fdsinfo** structures to be filled in with per–process file descriptor information. If a value of **NULL** is passed for this parameter, the **getprocs** subroutine scans the process table and sets return values as normal, but no file descriptor entries are retrieved.

FileSize Specifies the size of a single **fdsinfo** structure.

IndexPointer Specifies the address of a process identifier which indicates the required process table entry (this does not have to correspond to an existing process). A process identifier of zero selects the first entry in the table. The process identifier is updated to indicate the next entry to be retrieved.

Count Specifies the number of process table entries requested.

Return Values

If successful, the **getprocs** subroutine returns the number of process table entries retrieved; if this is less than the number requested, the end of the process table has been reached. Otherwise, a value of –1 is returned, and the **errno** global variable is set to indicate the error.

Error Codes

The **getprocs** subroutine does not succeed if the following are true:

EINVAL The *ProcessSize* or *FileSize* parameters are invalid, or the *IndexPointer* parameter does not point to a valid process identifier, or the *Count* parameter is not greater than zero.

EFAULT The copy operation to one of the buffers was not successful.

Implementation Specifics

This subroutine is part of the Base Operating System (BOS) Runtime.

Related Information

The **getpid**, **getpgrp**, or **getppid** subroutines, the **getthrds** subroutine

The **ps** command.

getpw Subroutine

Purpose

Retrieves a user's /etc/passwd file entry.

Library

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

Syntax

int getpw (*UserID*, *Buffer*)

```
uid_t UserID
char *Buffer
```

Description

The **getpw** subroutine opens the /**etc**/**passwd** file and returns, in the *Buffer* parameter, the /**etc**/**passwd** file entry of the user specified by the *UserID* parameter.

Parameters

Buffer	Specifies a character buffer large enough to hold any /**etc**/**passwd** entry.
UserID	Specifies the ID of the user for which the entry is desired.

Return Values

The **getpw** subroutine returns:

0	Successful completion
–1	Not successful.

Implementation Specifics

This subroutine is part of AIX Base Operating System (BOS) Runtime.

Related Information

getpwent, getpwuid, getpwnam, putpwent, setpwent, or endpwent Subroutine

Purpose

Accesses the basic user information in the user database.

Library

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

Syntax

```
#include <sys/types.h>
#include <pwd.h>

struct passwd *getpwent ( )

struct passwd *getpwuid (UserID)
uid_t UserID;

struct passwd *getpwnam (Name)
char *Name;

int putpwent (Password, File)
struct passwd *Password;
FILE *File;

void setpwent ( )

void endpwent ( )
```

Description

Attention: All information generated by the **getpwent**, **getpwnam**, and **getpwuid** subroutines is stored in a static area. Subsequent calls to these subroutines overwrite this static area. To save the information in the static area, applications should copy it.

These subroutines access the basic user attributes.

The **setpwent** subroutine opens the user database if it is not already open. Then, this subroutine sets the cursor to point to the first user entry in the database. The **endpwent** subroutine closes the user database.

The **getpwent**, **getpwnam**, and **getpwuid** subroutines return information about a user. These subroutines do the following:

getpwent	Returns the next user entry in the sequential search.
getpwnam	Returns the first user entry in the database whose name matches the *Name* parameter.
getpwuid	Returns the first user entry in the database whose ID matches the *UserID* parameter.

The **putpwent** subroutine writes a password entry into a file in the colon–separated format of the /**etc**/**passwd** file.

The user Structure

The **getpwent**, **getpwnam**, and **getpwuid** subroutines return a **user** structure. This structure The **user** structure is defined in the **pwd.h** file and has the following fields:

`pw_name`	Contains the name of the user name.
`pw_passwd`	Contains the user's encrypted password. **Note:** If the password is not stored in the **/etc/passwd** file and the invoker does not have access to the shadow file that contains passwords, this field contains an undecryptable string, usually an `*` (asterisk).
`pw_uid`	Contains the user's ID.
`pw_gid`	Identifies the user's principal group ID.
`pw_gecos`	Contains general user information.
`pw_dir`	Identifies the user's home directory.
`pw_shell`	Identifies the user's login shell.

Note: If Network Information Services (NIS) is enabled on the system, these subroutines attempt to retrieve the information from the NIS authentication server before attempting to retrieve the information locally.

Parameters

File	Points to an open file whose format is similar to the **/etc/passwd** file format.
Name	Specifies the user name.
Password	Points to a password structure. This structure contains user attributes.
UserID	Specifies the user ID.

Security

Files Accessed:

Mode	File
rw	**/etc/passwd** (write access for the **putpwent** subroutine only)
r	**/etc/security/passwd** (if the password is desired)

Return Values

The **getpwent**, **getpwnam**, and **getpwuid** subroutines return a pointer to a valid password structure if successful. Otherwise, a null pointer is returned.

The **getpwent** subroutine will return a null pointer and an **errno** value of **ENOATTR** when it detects a corrupt entry. To get subsequent entries following the corrupt entry, call the **getpwent** subroutine again.

Implementation Specifics

These subroutines are part of Base Operating System (BOS) Runtime.

Files

/etc/passwd	Contains user IDs and their passwords

Related Information

The **getgrent** subroutine, **getgroupattr** subroutine, **getpwuid_r**, **getuserattr** subroutine, **getuserpw**, **putuserpw, or putuserpwhist** subroutine, **setuserdb** subroutine.

List of Security and Auditing Subroutines, Subroutines Overview in *AIX General Programming Concepts : Writing and Debugging Programs*.

getrlimit, getrlimit64, setrlimit, setrlimit64, or vlimit Subroutine

Purpose

Controls maximum system resource consumption.

Library

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

Syntax

```
#include <sys/time.h>
#include <sys/resource.h>

int setrlimit(Resource1, RLP)
int Resource1;
struct rlimit *RLP;

int setrlimit64 (Resource1, RLP)
int Resource1;
struct rlimit64 *RLP;

int getrlimit (Resource1, RLP)
int Resource1;
struct rlimit *RLP;

int getrlimit64 (Resource1, RLP)
int Resource1;
struct rlimit64 *RLP;

#include <sys/vlimit.h>

vlimit (Resource2,Value)
int Resource2, Value;
```

Description

The **getrlimit** subroutine returns the values of limits on system resources used by the current process and its children processes. The **setrlimit** subroutine sets these limits. The **vlimit** subroutine is also supported, but the **getrlimit** subroutine replaces it.

A resource limit is specified as either a soft (current) or hard limit. A calling process can raise or lower its own soft limits, but it cannot raise its soft limits above its hard limits. A calling process must have root user authority to raise a hard limit.

The **rlimit** structure specifies the hard and soft limits for a resource, as defined in the **sys/resource.h** file. The **RLIM_INFINITY** value defines an infinite value for a limit.

When compiled in 32–bit mode, RLIM_INFINITY is a 32–bit value; when compiled in 64–bit mode, it is a 64–bit value. 32–bit routines should use **RLIM64_INFINITY** when setting 64–bit limits with the **setrlimit64** routine, and recognize this value when returned by **getrlimit64**.

This information is stored as per–process information. This subroutine must be executed directly by the shell if it is to affect all future processes created by the shell.

Note: Raising the data limit does not raise the program break value. Use the **brk/sbrk** subroutines to raise the break value. If the proper memory segments are not initialized at program load time, raising your memory limit will not allow access to this memory. Use the **–bmaxdata** flag of the **ld** command to set up these segments at load time.

When compiled in 32–bit mode, the **struct rlimit** values may be returned as RLIM_SAVED_MAX or RLIM_SAVED_CUR when the actual resource limit is too large to represent as a 32–bit **rlim_t**.

These values can be used by library routines which set their own **rlimits** to save off potentially 64–bit **rlimit** values (and prevent them from being truncated by the 32–bit **struct rlimit**). Unless the library routine intends to permanently change the **rlimits**, the RLIM_SAVED_MAX and RLIM_SAVED_CUR values can be used to restore the 64–bit **rlimits**.

Parameters

Resource1 Can be one of the following values:

RLIMIT_AS The maximum size of a process' total available memory, in bytes. This limit is not enforced.

RLIMIT_CORE The largest size, in bytes, of a **core** file that can be created. This limit is enforced by the kernel. If the value of the **RLIMIT_FSIZE** limit is less than the value of the **RLIMIT_CORE** limit, the system uses the **RLIMIT_FSIZE** limit value as the soft limit.

RLIMIT_CPU The maximum amount of central processing unit (CPU) time, in seconds, to be used by each process. If a process exceeds its soft CPU limit, the kernel will send a **SIGXCPU** signal to the process.

RLIMIT_DATA The maximum size, in bytes, of the data region for a process. This limit defines how far a program can extend its break value with the **sbrk** subroutine. This limit is enforced by the kernel.

RLIMIT_FSIZE The largest size, in bytes, of any single file that can be created. When a process attempts to write, truncate, or clear beyond its soft **RLIMIT_FSIZE** limit, the operation will fail with **errno** set to **EFBIG**. If the environment variable **XPG_SUS_ENV=ON** is set in the user's environment before the process is executed, then the **SIGXFSZ** signal is also generated.

RLIMIT_NOFILE This is a number one greater than the maximum value that the system may assign to a newly–created descriptor.

RLIMIT_STACK The maximum size, in bytes, of the stack region for a process. This limit defines how far a program stack region can be extended. Stack extension is performed automatically by the system. This limit is enforced by the kernel. When the stack limit is reached, the process receives a **SIGSEGV** signal. If this signal is not caught by a handler using the signal stack, the signal ends the process.

RLIMIT_RSS The maximum size, in bytes, to which the resident set size of a process can grow. This limit is not enforced by the kernel. A process may exceed its soft limit size without being ended.

RLP Points to the **rlimit** or **rlimit64** structure, which contains the soft (current) and hard limits. For the **getrlimit** subroutine, the requested limits are returned in this structure. For the **setrlimit** subroutine, the desired new limits are specified here.

Resource2 The flags for this parameter are defined in the **sys/vlimit.h**, and are mapped to corresponding flags for the **setrlimit** subroutine.

Value Specifies an integer used as a soft–limit parameter to the **vlimit** subroutine.

Return Values

On successful completion, a return value of 0 is returned, changing or returning the resource limit. Otherwise, a value of –1 is returned and the **errno** global variable is set to indicate the error.

Error Codes

The **getrlimit**, **getrlimit64**, **setrlimit**, **setrlimit64**, or **vlimit** subroutine is unsuccessful if one of the following is true:

EFAULT The address specified for the *RLP* parameter is not valid.

EINVAL The *Resource1* parameter is not a valid resource, or the limit specified in the *RLP* parameter is invalid.

EPERM The limit specified to the **setrlimit** subroutine would have raised the maximum limit value, and the caller does not have root user authority.

Implementation Specifics

These subroutines are part of Base Operating System (BOS) Runtime.

Application limits may be further constrained by available memory or implementation defined constants such as **OPEN_MAX** (maximum available open files).

Related Information

The **sigaction**, **sigvec**, or **signal** subroutines, **sigstack** subroutine, **ulimit** subroutine.

getroleattr, nextrole or putroleattr Subroutine

Purpose

Accesses the role information in the roles database.

Library

Security Library (**libc.a**)

Syntax

```
#include <usersec.h>

int getroleattr(Role, Attribute, Value, Type)
char *Role;
char *Attribute;
void *Value;
int Type;

char *nextrole(void)

int putroleattr(Role, Attribute, Value, Type)
char *Role;
char *Attribute;
void *Value;
int Type;
```

Description

The **getroleattr** subroutine reads a specified attribute from the role database. If the database is not already open, this subroutine does an implicit open for reading.

Similarly, the **putroleattr** subroutine writes a specified attribute into the role database. If the database is not already open, this subroutine does an implicit open for reading and writing. Data changed by the **putroleattr** subroutine must be explicitly committed by calling the **putroleattr** subroutine with a Type parameter specifying SEC_COMMIT. Until all the data is committed, only the **getroleattr** subroutine within the process returns written data.

The **nextrole** subroutine returns the next role in a linear search of the role database. The consistency of consecutive searches depends upon the underlying storage–access mechanism and is not guaranteed by this subroutine.

The **setroledb** and **endroledb** subroutines should be used to open and close the role database.

Parameters

Attribute Specifies which attribute is read. The following possible attributes are defined in the **usersec.h** file:

S_ROLELIST List of roles included by this role. The attribute type is **SEC_LIST**.

S_AUTHORIZATIONS List of authorizations included by this role. The attribute type is **SEC_LIST**.

S_GROUPS List of groups required for this role. The attribute type is **SEC_LIST**.

S_SCREENS List of SMIT screens required for this role. The attribute type is **SEC_LIST**.

S_VISIBILITY Number value stating the visibility of the role. The attribute type is **SEC_INT**.

S_MSGCAT Message catalog number. The attribute type is SEC_INT.

S_MSGNUMBER
Message number within the catalog. The attribute type is **SEC_INT**.

Type Specifies the type of attribute expected. Valid types are defined in the **usersec.h** file and include:

SEC_INT The format of the attribute is an integer.

For the **getroleattr** subroutine, the user should supply a pointer to a defined integer variable.

For the **putroleattr** subroutine, the user should supply an integer.

SEC_CHAR The format of the attribute is a null–terminated character string.

For the **getroleattr** subroutine, the user should supply a pointer to a defined character pointer variable. For the **putroleattr** subroutine, the user should supply a character pointer.

SEC_LIST The format of the attribute is a series of concatenated strings, each null–terminated. The last string in the series must be an empty (zero character count) string.

For the **getroleattr** subroutine, the user should supply a pointer to a defined character pointer variable. For the **putroleattr** subroutine, the user should supply a character pointer.

SEC_COMMIT For the **putroleattr** subroutine, this value specified by itself indicates that changes to the named role are to be committed to permanent storage. The *Attribute* and *Value* parameters are ignored. If no role is specified, the changes to all modified roles are committed to permanent storage.

SEC_DELETE The corresponding attribute is deleted from the database.

SEC_NEW Updates the role database file with the new role name when using the **putroleattr** subroutine.

Value Specifies a buffer, a pointer to a buffer, or a pointer to a pointer depending on the *Attribute* and *Type* parameters. See the *Type* parameter for more details.

Return Values

If successful, the **getroleattr** returns 0. Otherwise, a value of –1 is returned and the **errno** global variables is set to indicate the error.

Error Codes

Possible return codes are:

EACCES	Access permission is denied for the data request.
ENOENT	The specified *Role* parameter does not exist or the attribute is not defined for this user.
ENOATTR	The specified role attribute does not exist for this role.
EINVAL	The *Attribute* parameter does not contain one of the defined attributes or null.
EINVAL	The *Value* parameter does not point to a valid buffer or to valid data for this type of attribute.
EPERM	Operation is not permitted.

Related Information

The **getuserattr**, **nextusracl**, or **putusraclattr** subroutine, **setroledb**, or **endacldb** subroutine.

getrpcent, getrpcbyname, getrpcbynumber, setrpcent, or endrpcent Subroutine

Purpose

Accesses the **/etc/rpc** file.

Library

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

Syntax

```
#include <netdb.h>

struct rpcent *getrpcent ()
struct rpcent *getrpcbyname (Name)
char *Name;
struct rpcent *getrpcbynumber (Number)
int Number;
void setrpcent (StayOpen)
int StayOpen
void endrpcent
```

Description

Attention: Do not use the **getrpcent**, **getrpcbyname**, **getrpcbynumber**, **setrpcent**, or **endrpcent** subroutine in a multithreaded environment.

Attention: The information returned by the **getrpcbyname**, and **getrpcbynumber** subroutines is stored in a static area and is overwritten on subsequent calls. Copy the information to save it.

The **getprcbyname** and **getrpcbynumber** subroutines each return a pointer to an object with the **rpcent** structure. This structure contains the broken–out fields of a line from the **/etc/rpc** file. The **getprcbyname** and **getrpcbynumber** subroutines searches the **rpc** file sequentially from the beginning of the file until it finds a matching RPC program name or number, or until it reaches the end of the file. The **getrpcent** subroutine reads the next line of the file, opening the file if necessary.

The **setrpcent** subroutine opens and rewinds the **/etc/rpc** file. If the *StayOpen* parameter does not equal 0, the **rpc** file is not closed after a call to the **getrpcent** subroutine.

The **setrpcent** subroutine rewinds the **rpc** file. The **endrpcent** subroutine closes it.

The **rpc** file contains information about Remote Procedure Call (RPC) programs. The **rpcent** structure is in the **/usr/include/sys/rpcent.h** file and contains the following fields:

`r_name`	Contains the name of the server for an RPC program
`r_aliases`	Contains an alternate list of names for RPC programs. This list ends with a 0.
`r_number`	Contains a number associated with an RPC program.

Parameters

Name	Specifies the name of a server for **rpc** program.
Number	Specifies the **rpc** program number for service.
StayOpen	Contains a value used to indicate whether to close the **rpc** file.

Return Values

These subroutines return a null pointer when they encounter the end of a file or an error.

Implementation Specifics

These subroutines are part of Base Operating System (BOS) Runtime.

Files

/etc/rpc	Contains information about Remote Procedure Call (RPC) programs.

Related Information

Remote Procedure Call (RPC) for Programming in *AIX General Programming Concepts : Writing and Debugging Programs*

getrusage, getrusage64, times, or vtimes Subroutine

Purpose

Displays information about resource use.

Libraries

getrusage, **getrusage64**, **times**: Standard C Library (**libc.a**)

vtimes: Berkeley Compatibility Library (**libbsd.a**)

Syntax

```
#include <sys/times.h>
#include <sys/resource.h>

int getrusage (Who, RUsage)
int Who;
struct rusage *RUsage;

int getrusage64 (Who, RUsage)
int Who;
struct rusage64 *RUsage;

#include <sys/types.h>
#include <sys/times.h>

clock_t times (Buffer)
struct tms *Buffer;

#include <sys/times.h>

vtimes (ParentVM, ChildVM)
struct vtimes *ParentVm, ChildVm;
```

Description

The **getrusage** subroutine displays information about how resources are used by the current process or all completed child processes.

When compiled in 64–bit mode, **rusage** counters are 64 bits. If **getrusage** is compiled in 32–bit mode, **rusage** counters are 32 bits. If the kernel's value of a **usage** counter has exceeded the capacity of the corresponding 32–bit **rusage** value being returned, the **rusage** value is set to INT_MAX.

The **getrusage64** subroutine can be called to make 64–bit **rusage** counters explicitly available in a 32–bit environment.

The **times** subroutine fills the structure pointed to by the *Buffer* parameter with time–accounting information. All time values reported by the **times** subroutine are measured in terms of the number of clock ticks used. Applications should use **sysconf** (**_SC_CLK_TCK**) to determine the number of clock ticks per second.

The **tms** structure defined in the **/usr/include/sys/times.h** file contains the following fields:

```
time_t   tms_utime;
time_t   tms_stime;
time_t   tms_cutime;
time_t   tms_cstime;
```

This information is read from the calling process as well as from each completed child process for which the calling process executed a **wait** subroutine.

`tms_utime`	The CPU time used for executing instructions in the user space of the calling process
`tms_stime`	The CPU time used by the system on behalf of the calling process.
`tms_cutime`	The sum of the `tms_utime` and the `tms_cutime` values for all the child processes.
`tms_cstime`	The sum of the `tms_stime` and the `tms_cstime` values for all the child processes.

Note: The system measures time by counting clock interrupts. The precision of the values reported by the **times** subroutine depends on the rate at which the clock interrupts occur.

Parameters

Who Specifies a value of either **RUSAGE_SELF** or **RUSAGE_CHILDREN**.

RUsage Points to a buffer described in the /**usr**/**include**/**sys**/**resource.h** file. The fields are interpreted as follows:

`ru_utime` The total amount of time running in user mode.

`ru_stime` The total amount of time spent in the system executing on behalf of the processes.

`ru_maxrss` The maximum size, in kilobytes, of the used resident set size.

`ru_ixrss` An integral value indicating the amount of memory used by the text segment that was also shared among other processes. This value is expressed in units of kilobytes * seconds–of–execution and is calculated by adding the number of shared memory pages in use each time the internal system clock ticks, and then averaging over one–second intervals.

`ru_idrss` An integral value of the amount of unshared memory in the data segment of a process (expressed in units of kilobytes * seconds–of–execution).

`ru_minflt` The number of page faults serviced without any I/O activity. In this case, I/O activity is avoided by reclaiming a page frame from the list of pages awaiting reallocation.

`ru_majflt` The number of page faults serviced that required I/O activity.

`ru_nswap` The number of times a process was swapped out of main memory.

`ru_inblock` The number of times the file system performed input.

`ru_oublock` The number of times the file system performed output.

Note: The numbers that the `ru_inblock` and `ru_oublock` fields display account for real I/O only; data supplied by the caching mechanism is charged only to the first process to read or write the data.

`ru_msgsnd` The number of IPC messages sent.

`ru_msgrcv` The number of IPC messages received.

`ru_nsignals` The number of signals delivered.

`ru_nvcsw` The number of times a context switch resulted because a process voluntarily gave up the processor before its time slice was completed. This usually occurs while the process waits for availability of a resource.

`ru_nivcsw` The number of times a context switch resulted because a higher priority process ran or because the current process exceeded its time slice.

Buffer Points to a **tms** structure.

ParentVm	Points to a **vtimes** structure that contains the accounting information for the current process.
ChildVm	Points to a **vtimes** structure that contains the accounting information for the terminated child processes of the current process.

Return Values

Upon successful completion, the **getrusage** and **getrusage64** subroutines return a value of 0. Otherwise, a value of –1 is returned and the **errno** global variable is set to indicate the error.

Upon successful completion, the **times** subroutine returns the elapsed real time in units of ticks, whether profiling is enabled or disabled. This reference time does not change from one call of the **times** subroutine to another. If the **times** subroutine fails, it returns a value of –1 and sets the **errno** global variable to indicate the error.

Error Codes

The **getrusage** and **getrusage64** subroutines do not run successfully if either of the following is true:

EINVAL	The *Who* parameter is not a valid value.
EFAULT	The address specified for *RUsage* is not valid.

The **times** subroutine does not run successfully if the following is true:

EFAULT	The address specified by the *buffer* parameter is not valid.

Implementation Specifics

These subroutines are part of Base Operating System (BOS) Runtime.

The **vtimes** subroutine is supported to provide compatibility with earlier programs.

The **vtimes** subroutine returns accounting information for the current process and for the completed child processes of the current process. Either the *ParentVm* parameter, the *ChildVm* parameter, or both may be 0. In that case, only the information for the nonzero pointers is returned.

After a call to the **vtimes** subroutine, each buffer contains information as defined by the contents of the **/usr/include/sys/vtimes.h** file.

Related Information

The **gettimer**, **settimer**, **restimer**, **stime**, or **time** subroutine, **wait**, **waitpid**, or **wait3** subroutine.

gets or fgets Subroutine

Purpose

Gets a string from a stream.

Library

Standard I/O Library (**libc.a**)

Syntax

```
#include <stdio.h>
char *gets (String)
char *String;

char *fgets (String, Number, Stream)
char *String;
int Number;
FILE *Stream;
```

Description

The **gets** subroutine reads bytes from the standard input stream, **stdin**, into the array pointed to by the *String* parameter. It reads data until it reaches a new–line character or an end–of–file condition. If a new–line character stops the reading process, the **gets** subroutine discards the new–line character and terminates the string with a null character.

The **fgets** subroutine reads bytes from the data pointed to by the *Stream* parameter into the array pointed to by the *String* parameter. The **fgets** subroutine reads data up to the number of bytes specified by the *Number* parameter minus 1, or until it reads a new–line character and transfers that character to the *String* parameter, or until it encounters an end–of–file condition. The **fgets** subroutine then terminates the data string with a null character.

The first successful run of the **fgetc**, **fgets**, **fgetwc**, **fgetws**, **fread**, **fscanf**, **getc**, **getchar**, **gets** or **scanf** subroutine using a stream that returns data not supplied by a prior call to the **ungetc** or **ungetwc** subroutine marks the `st_atime` field for update.

Parameters

String	Points to a string to receive bytes.
Stream	Points to the **FILE** structure of an open file.
Number	Specifies the upper bound on the number of bytes to read.

Return Values

If the **gets** or **fgets** subroutine encounters the end of the file without reading any bytes, it transfers no bytes to the *String* parameter and returns a null pointer. If a read error occurs, the **gets** or **fgets** subroutine returns a null pointer and sets the **errno** global variable (errors are the same as for the **fgetc** subroutine). Otherwise, the **gets** or **fgets** subroutine returns the value of the *String* parameter.

Note: Depending upon which library routine the application binds to, this subroutine may return **EINTR**. Refer to the **signal** subroutine regarding the **SA_RESTART** value.

Implementation Specifics

These subroutines are part of Base Operating System (BOS) Runtime.

Related Information

The **feof**, **ferror**, **clearerr**, or **fileno** macro, **fopen**, **freopen**, or **fdopen** subroutine, **fread** subroutine, **getc**, **getchar**, **fgetc**, or **getw** subroutine, **getwc**, **fgetwc**, or **getwchar** subroutine, **getws** or **fgetws** subroutine, **puts** or **fputs** subroutine, **putws** or **fputws** subroutine, **scanf**, **fscanf**, or **sscanf** subroutine, **ungetc** or **ungetwc** subroutine.

List of String Manipulation Services, Subroutines Overview in *AIX General Programming Concepts : Writing and Debugging Programs*.

getsid Subroutine

Purpose

Returns the session ID of the calling process.

Library

(**libc.a**)

Syntax

#include <unistd.h>

pid_t getsid (pid_ t *pid***)**

Description

The **getsid** subroutine returns the process group ID of the process that is the session leader of the process specified by *pid*. If *pid* is equal to **pid_t** subroutine, it specifies the calling process.

Parameters

pid	A process ID of the process being queried.

Return Values

Upon successful completion, **getsid** subroutine returns the process group ID of the session leaded of the specified process. Otherwise, it returns (**pid_t**)–1 and set **errno** to indicate the error.

id	The session ID of the requested process.
–1	Not successful and the **errno** global variable is set to one of the following error codes.

Error Codes

ESRCH	There is no process with a process ID equal to *pid.*
EPERM	The process specified by pid is not in the same session as the calling process.
ESRCH	There is no process with a process ID equal to *pid*.

Related Information

The **exec** subroutines, **fork** subroutines, **getpid** subroutines, **setpgid** subroutines.

getssys Subroutine

Purpose

Reads a subsystem record.

Library

System Resource Controller Library (**libsrc.a**)

Syntax

```
#include <sys/srcobj.h>
#include <spc.h>

int getssys( SubsystemName, SRCSubsystem)
char *SubsystemName;
struct SRCsubsys *SRCSubsystem;
```

Description

The **getssys** subroutine reads a subsystem record associated with the specified subsystem and returns the ODM record in the **SRCsubsys** structure.

The **SRCsubsys** structure is defined in the **sys/srcobj.h** file.

Parameters

SRCSubsystem	Points to the **SRCsubsys** structure.
SubsystemName	Specifies the name of the subsystem to be read.

Return Values

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

Error Codes

If the **getssys** subroutine fails, the following is returned:

SRC_NOREC	Subsystem name does not exist.

Implementation Specifics

This subroutine is part of Base Operating System (BOS) Runtime.

Files

/etc/objrepos/SRCsubsys	SRC Subsystem Configuration object class.

Related Information

The **addssys** subroutine, **delssys** subroutine, **getsubsvr** subroutine.

Defining Your Subsystem to the SRC, List of SRC Subroutines, System Resource Controller (SRC) Overview for Programmers in *AIX General Programming Concepts : Writing and Debugging Programs*.

getsubopt Subroutine

Purpose

Parse suboptions from a string.

Library

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

Syntax

```
#include <stdlib.h>

int getsubopt (char **optionp,
char * const * tokens,
char ** valuep)
```

Description

The **getsubopt** subroutine parses suboptions in a flag parameter that were initially parsed by the **getopt** subroutine. These suboptions are separated by commas and may consist of either a single token, or a token–value pair separated by an equal sign. Because commas delimit suboptions in the option string, they are not allowed to be part of the suboption or the value of a suboption. similarly, because the equal sign separates a token from its value, a token must not contain an equal sign.

The **getsubopt** subroutine takes the address of a pointer to the option string, a vector of possible tokens, and the address of a value string pointer. It returns the index of the token that matched the suboption in the input string or –1 if there was no match. If the option string at **optionp* contains only one suboption, the **getsubopt** subroutine updates **optionp* to point to the start of the next suboption. It the suboption has an associated value, the **getsubopt** subroutine updates **valuep* to point to the value's first character. Otherwise it sets **valuep* to a NULL pointer.

The token vector is organized as a series of pointers to strings. The end of the token vector is identified by a NULL pointer.

When the **getsubopt** subroutine returns, if **valuep* is not a NULL pointer then the suboption processed included a value. The calling program may use this information to determine if the presence or lack of a value for this suboption is an error.

Additionally, when the **getsubopt** subroutine fails to match the suboption with the tokens in the *tokens* array, the calling program should decide if this is an error, or if the unrecognized option should be passed on to another program.

Return Values

The **getsubopt** subroutine returns the index of the matched token string, or –1 if no token strings were matched.

Implementation Specifics

This subroutine is part of Base Operating System (BOS) Runtime.

Related Information

The **getopt** subroutine.

getsubsvr Subroutine

Purpose

Reads a subsystem record.

Library

System Resource Controller Library (**libsrc.a**)

Syntax

```
#include <sys/srcobj.h>
#include <spc.h>

int getsubsvr( SubserverName, SRCSubserver)
char *SubserverName;
struct SRCSubsvr *SRCSubserver;
```

Description

The **getsubsvr** subroutine reads a subsystem record associated with the specified subserver and returns the ODM record in the **SRCsubsvr** structure.

The **SRCsubsvr** structure is defined in the **sys/srcobj.h** file and includes the following fields:

char	sub_type[30];
char	subsysname[30];
short	sub_code;

Parameters

SRCSubserver	Points to the **SRCsubsvr** structure.
SubserverName	Specifies the subserver to be read.

Return Values

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

Error Codes

If the **getsubsvr** subroutine fails, the following is returned:

SRC_NOREC	The specified **SRCsubsvr** record does not exist.

Implementation Specifics

This subroutine is part of Base Operating System (BOS) Runtime.

Files

/etc/objrepos/SRCsubsvr	SRC Subserver Configuration object class.

Related Information

The **getssys** subroutine.

Defining Your Subsystem to the SRC, List of SRC Subroutines, System Resource Controller (SRC) Overview for Programmers in *AIX General Programming Concepts : Writing and Debugging Programs*.

getthrds Subroutine

Purpose

Gets kernel thread table entries.

Library

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

Syntax

```
#include <procinfo.h>
#include <sys/types.h>

int
getthrds (ProcessIdentifier, ThreadBuffer, ThreadSize,
IndexPointer, Count)
pid_t ProcessIdentifier;
struct thrdsinfo *ThreadBuffer;
or struct thrdsinfo64 *ThreadBuffer;
int ThreadSize;
tid_t *IndexPointer;
int Count;
```

Description

The **getthrds** subroutine returns information about kernel threads, including kernel thread table information defined by the **thrdsinfo** or **thrdsinfo64** structure.

The **getthrds** subroutine retrieves up to *Count* kernel thread table entries, starting with the entry corresponding to the thread identifier indicated by *IndexPointer*, and places them in the array of **thrdsinfo** or **thrdsinfo64** structures indicated by the *ThreadBuffer* parameter.

On return, the kernel thread identifier referenced by *IndexPointer* is updated to indicate the next kernel thread table entry to be retrieved. The **getthrds** subroutine returns the number of kernel thread table entries retrieved.

If the *ProcessIdentifier* parameter indicates a process identifier, only kernel threads belonging to that process are considered. If this parameter is set to –1, all kernel threads are considered.

The **getthrds** subroutine is normally called repeatedly in a loop, starting with a kernel thread identifier of zero, and looping until the return value is less than *Count*, indicating that there are no more entries to retrieve.

1. Do not use information from the **procsinfo** structure (see the **getprocs** subroutine) to determine the value of the *Count* parameter; a process may create or destroy kernel threads in the interval between a call to **getprocs** and a subsequent call to **getthrds**.
2. The kernel thread table may change while the **getthrds** subroutine is accessing it. Returned entries will always be consistent, but since kernel threads can be created or destroyed while the **getthrds** subroutine is running, there is no guarantee that retrieved entries will still exist, or that all existing kernel threads have been retrieved.

When used in 32–bit mode, limits larger than can be represented in 32 bits are truncated to RLIM_INFINITY. Large values are truncated to INT_MAX. Alternatively, the **struct thrdsinfo64** and *sizeof* (**struct thrdsinfo64**) can be used by 32–bit **getthrds** to return full 64–bit thread information. Note that the **thrdsinfo64** structure not only inceases certain **thrdsinfo** fields from 32 to 64 bits, but that it contains additional information not present in **thrdsinfo**. The **struct thrdsinfo64** contains the same data as **struct thrdsinfo** when compiled in a 64–bit program.

Parameters

ProcessIdentifier	Specifies the process identifier of the process whose kernel threads are to be retrieved. If this parameter is set to –1, all kernel threads in the kernel thread table are retrieved.
ThreadBuffer	Specifies the starting address of an array of **thrdsinfo** or **thrdsinfo64** structures which will be filled in with kernel thread table entries. If a value of **NULL** is passed for this parameter, the **getthrds** subroutine scans the kernel thread table and sets return values as normal, but no kernel thread table entries are retrieved.
ThreadSize	Specifies the size of a single **thrdsinfo** or **thrdsinfo64** structure.
IndexPointer	Specifies the address of a kernel thread identifier which indicates the required kernel thread table entry (this does not have to correspond to an existing kernel thread). A kernel thread identifier of zero selects the first entry in the table. The kernel thread identifier is updated to indicate the next entry to be retrieved.
Count	Specifies the number of kernel thread table entries requested.

Return Value

If successful, the **getthrds** subroutine returns the number of kernel thread table entries retrieved; if this is less than the number requested, the end of the kernel thread table has been reached. Otherwise, a value of –1 is returned, and the **errno** global variable is set to indicate the error.

Error Codes

The **getthrds** subroutine fails if the following are true:

EINVAL	The *ThreadSize* is invalid, or the *IndexPointer* parameter does not point to a valid kernel thread identifier, or the *Count* parameter is not greater than zero.
ESRCH	The process specified by the *ProcessIdentifier* parameter does not exist.
EFAULT	The copy operation to one of the buffers failed.

Implementation Specifics

This subroutine is part of the Base Operating System (BOS) Runtime.

Related Information

The **getpid**, **getpgrp**, or **getppid** subroutines, the **getprocs** subroutine.

The **ps** command.

gettimeofday, settimeofday, or ftime Subroutine

Purpose

Displays, gets and sets date and time.

Libraries

gettimeofday, **settimeofday**: Standard C Library (**libc.a**)

ftime: Berkeley Compatibility Library (**libbsd.a**)

Syntax

```
#include <sys/time.h>
int gettimeofday (Tp, Tzp)
struct timeval *Tp;
void *Tzp;
int settimeofday (Tp, Tzp)
struct timeval *Tp;
struct timezone *Tzp;

#include <sys/types.h>
#include <sys/timeb.h>
int ftime (Tp)
struct timeb *Tp;
```

Description

Current Greenwich time and the current time zone are displayed with the **gettimeofday** subroutine, and set with the **settimeofday** subroutine. The time is expressed in seconds and microseconds since midnight (0 hour), January 1, 1970. The resolution of the system clock is hardware–dependent, and the time may be updated either continuously or in "ticks." If the *Tzp* parameter has a value of 0, the time zone information is not returned or set.

The *Tp* parameter returns a pointer to a **timeval** structure that contains the time since the epoch began in seconds and microseconds.

The **timezone** structure indicates both the local time zone (measured in minutes of time westward from Greenwich) and a flag that, if nonzero, indicates that daylight saving time applies locally during the appropriate part of the year.

In addition to the difference in timer granularity, the **timezone** structure distinguishes these subroutines from the POSIX **gettimer** and **settimer** subroutines, which deal strictly with Greenwich Mean Time.

The **ftime** subroutine fills in a structure pointed to by its argument, as defined by **<sys/timeb.h>**. The structure contains the time in seconds since 00:00:00 UTC (Coordinated Universal Time), January 1, 1970, up to 1000 milliseconds of more–precise interval, the local timezone (measured in minutes of time westward from UTC), and a flag that, if nonzero, indicates that Daylight Saving time is in effect, and the values stored in the timeb structure have been adjusted accordingly.

Parameters

Tp	Pointer to a **timeval** structure, defined in the **sys/time.h** file.
Tzp	Pointer to a **timezone** structure, defined in the **sys/time.h** file.

Return Values

If the subroutine succeeds, a value of 0 is returned. If an error occurs, a value of –1 is returned and **errno** is set to indicate the error.

Error Codes

If the **settimeofday** subroutine is unsuccessful, the **errno** value is set to **EPERM** to indicate that the process's effective user ID does not have root user authority.

No errors are defined for the **gettimeofday** or **ftime** subroutine.

gettimer, settimer, restimer, stime, or time Subroutine

Purpose

Gets or sets the current value for the specified systemwide timer.

Library

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

Syntax

```
#include <sys/time.h>
#include <sys/types.h>

int gettimer(TimerType, Value)
timer_t TimerType;
struct timestruc_t * Value;

#include <sys/timers.h>
#include <sys/types.h>

int gettimer(TimerType, Value)
timer_t TimerType;
struct itimerspec * Value;

int settimer(TimerType, TimePointer)
int TimerType;
const struct timestruc_t *TimePointer;

int restimer(TimerType, Resolution, MaximumValue)
int TimerType;
struct timestruc_t *Resolution, *MaximumValue;

int stime(Tp)
long *Tp;

#include <sys/types.h>

time_t time(Tp)
time_t *Tp;
```

Description

The **settimer** subroutine is used to set the current value of the *TimePointer* parameter for the systemwide timer, specified by the *TimerType* parameter.

When the **gettimer** subroutine is used with the function prototype in **sys/timers.h**, then except for the parameters, the **gettimer** subroutine is identical to the **getinterval** subroutine. Use of the **getinterval** subroutine is recommended, unless the **gettimer** subroutine is required for a standards–conformant application. The description and semantics of the **gettimer** subroutine are subject to change between releases, pending changes in the draft standard upon which the current **gettimer** subroutine description is based.

When the **gettimer** subroutine is used with the function prototype in **/sys/timers.h**, the **gettimer** subroutine returns an **itimerspec** structure to the pointer specified by the *Value* parameter. The **it_value** member of the **itimerspec** structure represents the amount of time in the current interval before the timer (specified by the *TimerType* parameter) expires, or a zero interval if the timer is disabled. The members of the pointer specified by the *Value* parameter are subject to the resolution of the timer.

When the **gettimer** subroutine is used with the function prototype in **sys/time.h**, the **gettimer** subroutine returns a **timestruc** structure to the pointer specified by the *Value* parameter. This structure holds the current value of the system wide timer specified by the *Value* parameter.

The resolution of any timer can be obtained by the **restimer** subroutine. The *Resolution* parameter represents the resolution of the specified timer. The *MaximumValue* parameter represents the maximum possible timer value. The value of these parameters are the resolution accepted by the **settimer** subroutine.

Note: If a nonprivileged user attempts to submit a fine granularity timer (that is, a timer request of less than 10 milliseconds), the timer request is raised to 10 milliseconds.

The **time** subroutine returns the time in seconds since the Epoch (that is, 00:00:00 GMT, January 1, 1970). The *Tp* parameter points to an area where the return value is also stored. If the *Tp* parameter is a null pointer, no value is stored.

Parameters

Value	Points to a structure of type **itimerspec**.
TimerType	Specifies the systemwide timer: **TIMEOFDAY** (POSIX system clock timer) This timer represents the time–of–day clock for the system. For this timer, the values returned by the **gettimer** subroutine and specified by the **settimer** subroutine represent the amount of time since 00:00:00 GMT, January 1, 1970.
TimePointer	Points to a structure of type **struct timestruc_t**.
Resolution	The resolution of a specified timer.
MaximumValue	The maximum possible timer value.
Tp	Points to a structure containing the time in seconds.

Return Values

The **gettimer**, **settimer**, **restimer**, and **stime** subroutines return a value of 0 (zero) if the call is successful. A return value of –1 indicates an error occurred, and **errno** is set.

The **time** subroutine returns the value of time in seconds since Epoch. Otherwise, a value of ((**time_t**) – 1) is returned and the **errno** global variable is set to indicate the error.

Error Codes

If an error occurs in the **gettimer**, **settimer**, **restimer**, or **stime** subroutine, a return value of – 1 is received and the **errno** global variable is set to one of the following error codes:

EINVAL	The *TimerType* parameter does not specify a known systemwide timer, or the *TimePointer* parameter of the **settimer** subroutine is outside the range for the specified systemwide timer.
EFAULT	A parameter address referenced memory that was not valid.
EIO	An error occurred while accessing the timer device.
EPERM	The requesting process does not have the appropriate privilege to set the specified timer.

If the **time** subroutine is unsuccessful, a return value of –1 is received and the **errno** global variable is set to the following:

EFAULT	A parameter address referenced memory that was not valid.

Implementation Specifics

These subroutines are part of Base Operating System (BOS) Runtime.

The **stime** subroutine is implemented to provide compatibility with older AIX, AT&T System V, and BSD systems. It calls the **settimer** subroutine using the **TIMEOFDAY** timer.

Related Information

The **asctime** subroutine, **clock** subroutine, **ctime** subroutine, **difftime** subroutine, **getinterval** subroutine, **gmtime** subroutine, **localtime** subroutine, **mktime** subroutine, **strftime** subroutine, **strptime** subroutine, **utime** subroutine.

List of Time Data Manipulation Services and Subroutines Overview in *AIX General Programming Concepts : Writing and Debugging Programs*.

gettimerid Subroutine

Purpose

Allocates a per–process interval timer.

Library

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

Syntax

```
#include <sys/time.h>
#include <sys/events.h>

timer_t gettimerid(TimerType, NotifyType)
int TimerType;
int NotifyType;
```

Description

The **gettimerid** subroutine is used to allocate a per–process interval timer based on the timer with the given timer type. The unique ID is used to identify the interval timer in interval timer requests. (See **getinterval** subroutine). The particular timer type, the *TimerType* parameter, is defined in the **sys/time.h** file and can identify either a systemwide timer or a per–process timer. The mechanism by which the process is to be notified of the expiration of the timer event is the *NotifyType* parameter, which is defined in the **sys/events.h** file.

The *TimerType* parameter represents one of the following timer types:

TIMEOFDAY	(POSIX system clock timer) This timer represents the time–of–day clock for the system. For this timer, the values returned by the **gettimer** subroutine and specified by the **settimer** subroutine represent the amount of time since 00:00:00 GMT, January 1, 1970, in nanoseconds.
TIMERID_ALRM	(Alarm timer) This timer schedules the delivery of a **SIGALRM** signal at a timer specified in the call to the **settimer** subroutine.
TIMERID_REAL	(Real–time timer) The real–time timer decrements in real time. A **SIGALRM** signal is delivered when this timer expires.
TIMERID_VIRTUAL	(Virtual timer) The virtual timer decrements in process virtual time. it runs only when the process is executing in user mode. A **SIGVTALRM** signal is delivered when it expires.
TIMERID_PROF	(Profiling timer) The profiling timer decrements both when running in user mode and when the system is running for the process. It is designed to be used by processes to profile their execution statistically. A **SIGPROF** signal is delivered when the profiling timer expires.

Interval timers with a notification value of **DELIVERY_SIGNAL** are inherited across an **exec** subroutine.

Parameters

NotifyType	Notifies the process of the expiration of the timer event.
TimerType	Identifies either a systemwide timer or a per–process timer.

Return Values

If the **gettimerid** subroutine succeeds, it returns a **timer_t** structure that can be passed to the per–process interval timer subroutines, such as the **getinterval** subroutine. If an error occurs, the value –1 is returned and **errno** is set.

Error Codes

If the **gettimerid** subroutine fails, the value –1 is returned and **errno** is set to one of the following error codes:

EAGAIN	The calling process has already allocated all of the interval timers associated with the specified timer type for this implementation.
EINVAL	The specified timer type is not defined.

Implementation Specifics

These subroutines are part of Base Operating System (BOS) Runtime.

Related Information

The **exec** subroutine, **fork** subroutine, **getinterval**, **incinterval**, **absinterval**, **resinc**, or **resabs** subroutine, **gettimer**, **settimer**, or **restimer** subroutine, **reltimerid** subroutine.

List of Time Data Manipulation Services and Subroutines Overview in *AIX General Programming Concepts : Writing and Debugging Programs*.

getttyent, getttynam, setttyent, or endttyent Subroutine

Purpose

Gets a tty description file entry.

Library

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

Syntax

```
#include <ttyent.h>

struct ttyent *getttyent()
struct ttyent *getttynam(Name)
char *Name;
void setttyent()
void endttyent()
```

Description

Attention: Do not use the **getttyent**, **getttynam**, **setttyent**, or **endttyent** subroutine in a multithreaded environment.

The **getttyent** and **getttynam** subroutines each return a pointer to an object with the **ttyent** structure. This structure contains the broken–out fields of a line from the tty description file. The **ttyent** structure is in the **/usr/include/sys/ttyent.h** file and contains the following fields:

`tty_name`	The name of the character special file in the /**dev** directory. The character special file must reside in the /**dev** directory.
`ty_getty`	The command that is called by the **init** process to initialize tty line characteristics. This is usually the **getty** command, but any arbitrary command can be used. A typical use is to initiate a terminal emulator in a window system.
`ty_type`	The name of the default terminal type connected to this tty line. This is typically a name from the **termcap** database. The **TERM** environment variable is initialized with this name by the **getty** or **login** command.
`ty_status`	A mask of bit fields that indicate various actions to be allowed on this tty line. The following is a description of each flag: **TTY_ON** Enables logins (that is, the **init** process starts the specified **getty** command on this entry). **TTY_SECURE** Allows a user with root user authority to log in to this terminal. The **TTY_ON** flag must be included.
`ty_window`	The command to execute for a window system associated with the line. The window system is started before the command specified in the `ty_getty` field is executed. If none is specified, this is null.
`ty_comment`	The trailing comment field. A leading delimiter and white space is removed.

The **getttyent** subroutine reads the next line from the tty file, opening the file if necessary. The **setttyent** subroutine rewinds the file. The **endttyent** subroutine closes it.

The **getttynam** subroutine searches from the beginning of the file until a matching name (specified by the *Name* parameter) is found (or until the EOF is encountered).

Parameters

Name Specifies the name of a tty description file.

Return Values

These subroutines return a null pointer when they encounter an EOF (end–of–file) character or an error.

Implementation Specifics

These subroutines are part of Base Operating System (BOS) Runtime.

Files

/usr/lib/libodm.a	Specifies the ODM (Object Data Manager) library.
/usr/lib/libcfg.a	Archives device configuration subroutines.
/etc/termcap	Defines terminal capabilities.

Related Information

The **ttyslot** subroutine.

The **getty** command, **init** command, **login** command.

List of Files and Directories Subroutines in *AIX General Programming Concepts : Writing and Debugging Programs*.

getuid or geteuid Subroutine

Purpose

Gets the real or effective user ID of the current process.

Library

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

Syntax

```
#include <sys/types.h>
#include <unistd.h>

uid_t getuid(void)

uid_t geteuid(void)
```

Description

The **getuid** subroutine returns the real user ID of the current process. The **geteuid** subroutine returns the effective user ID of the current process.

Return Values

The **getuid** and **geteuid** subroutines return the corresponding user ID.

Note: The **getuid** and **geteuid** subroutines always succeed.

Implementation Specifics

These subroutines are part of Base Operating System (BOS) Runtime.

Related Information

The **setuid** subroutine.

List of Security and Auditing Subroutines, Subroutines Overview in *AIX General Programming Concepts : Writing and Debugging Programs*.

getuinfo Subroutine

Purpose

Finds a value associated with a user.

Library

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

Syntax

```
char *getuinfo (Name)
char *Name;
```

Description

The **getuinfo** subroutine finds a value associated with a user. This subroutine searches a user information buffer for a string of the form *Name***=***Value* and returns a pointer to the *Value* substring if the *Name* value is found. A null value is returned if the *Name* value is not found.

The **INuibp** global variable points to the user information buffer:

```
extern char *INuibp;
```

This variable is initialized to a null value.

If the **INuibp** global variable is null when the **getuinfo** subroutine is called, the **usrinfo** subroutine is called to read user information from the kernel into a local buffer. The **INUuibp** is set to the address of the local buffer. If the **INuibp** external variable is not set, the **usrinfo** subroutine is automatically called the first time the **getuinfo** subroutine is called.

Parameter

Name	Specifies a user name.

Implementation Specifics

This subroutine is part of Base Operating System (BOS) Runtime.

Related Information

List of Security and Auditing Subroutines, Subroutines Overview in *AIX General Programming Concepts : Writing and Debugging Programs*.

getuserattr, IDtouser, nextuser, or putuserattr Subroutine

Purpose

Accesses the user information in the user database.

Library

Security Library (**libc.a**)

Syntax

```
#include <usersec.h>

int getuserattr (User, Attribute, Value, Type)
char *User;
char *Attribute;
void *Value;
int Type;

char *IDtouser(UID)
uid_t UID;

char *nextuser (Mode, Argument)
int Mode, Argument;

int putuserattr (User, Attribute, Value, Type)
char *User;
char *Attribute;
void *Value;
int Type;
```

Description

Attention: These subroutines and the **setpwent** and **setgrent** subroutines should not be used simultaneously. The results can be unpredictable.

These subroutines access user information. Because of their greater granularity and extensibility, you should use them instead of the **getpwent** routines.

The **getuserattr** subroutine reads a specified attribute from the user database. If the database is not already open, this subroutine does an implicit open for reading. A call to the **getuserattr** subroutine for every new user verifies that the user exists.

Similarly, the **putuserattr** subroutine writes a specified attribute into the user database. If the database is not already open, this subroutine does an implicit open for reading and writing. Data changed by the **putuserattr** subroutine must be explicitly committed by calling the **putuserattr** subroutine with a *Type* parameter specifying **SEC_COMMIT**. Until all the data is committed, only these subroutines within the process return written data.

New entries in the user and group databases must first be created by invoking **putuserattr** with the **SEC_NEW** type.

The **IDtouser** subroutine translates a user ID into a user name.

The **nextuser** subroutine returns the next user in a linear search of the user database. The consistency of consecutive searches depends upon the underlying storage–access mechanism and is not guaranteed by this subroutine.

The **setuserdb** and **enduserdb** subroutines should be used to open and close the user database.

Parameters

Argument Presently unused and must be specified as null.

Attribute Specifies which attribute is read. The following possible attributes are defined in the **usersec.h** file:

S_ID User ID. The attribute type is **SEC_INT**.

S_PGRP Principle group name. The attribute type is **SEC_CHAR**.

S_GROUPS Groups to which the user belongs. The attribute type is **SEC_LIST**.

S_ADMGROUPS Groups for which the user is an administrator. The attribute type is **SEC_LIST**.

S_ADMIN Administrative status of a user. The attribute type is **SEC_BOOL**.

S_AUDITCLASSES Audit classes to which the user belongs. The attribute type is **SEC_LIST**.

S_AUTHSYSTEM Defines the user's authentication method. The attribute type is **SEC_CHAR.**

S_HOME Home directory. The attribute type is **SEC_CHAR**.

S_SHELL Initial program run by a user. The attribute type is **SEC_CHAR**.

S_GECOS Personal information for a user. The attribute type is **SEC_CHAR**.

S_USRENV User–state environment variables. The attribute type is **SEC_LIST**.

S_SYSENV Protected–state environment variables. The attribute type is **SEC_LIST**.

S_LOGINCHK Specifies whether the user account can be used for local logins. The attribute type is **SEC_BOOL**.

S_HISTEXPIRE Defines the period of time (in weeks) that a user cannot reuse a password. The attribute type is **SEC_INT**.

S_HISTSIZE Specifies the number of previous passwords that the user cannot reuse. The attribute type is **SEC_INT**.

S_MAXREPEAT Defines the maximum number of times a user can repeat a character in a new password. The attribute type is **SEC_INT**.

S_MINAGE Defines the minimum age in weeks that the user's password must exist before the user can change it. The attribute type is **SEC_INT**.

S_PWDCHECKS Defines the password restriction methods for this account. The attribute type is **SEC_LIST**.

S_MINALPHA Defines the minimum number of alphabetic characters required in a new user's password. The attribute type is **SEC_INT**.

S_MINDIFF Defines the minimum number of characters required in a new password that were not in the old password. The attribute type is **SEC_INT**.

S_MINLEN Defines the minimum length of a user's password. The attribute type is **SEC_INT**.

S_MINOTHER Defines the minimum number of non–alphabetic characters required in a new user's password. The attribute type is **SEC_INT**.

S_DICTIONLIST Defines the password dictionaries for this account. The attribute type is **SEC_LIST**.

S_SUCHK Specifies whether the user account can be accessed with the **su** command. Type **SEC_BOOL**.

S_REGISTRY Defines the user's authentication registry. The attribute type is **SEC_CHAR**.

S_RLOGINCHK Specifies whether the user account can be used for remote logins using the **telnet** or **rlogin** commands. The attribute type is **SEC_BOOL**.

S_DAEMONCHK Specifies whether the user account can be used for daemon execution of programs and subsystems using the **cron** daemon or **src**. The attribute type is **SEC_BOOL**.

S_TPATH Defines how the account may be used on the trusted path. The attribute type is **SEC_CHAR**. This attribute must be one of the following values:

- **nosak** The secure attention key is not enabled for this account.
- **notsh** The trusted shell cannot be accessed from this account.
- **always** This account may only run trusted programs.
- **on** Normal trusted–path processing applies.

S_TTYS List of ttys that can or cannot be used to access this account. The attribute type is **SEC_LIST**.

S_SUGROUPS Groups that can or cannot access this account. The attribute type is **SEC_LIST**.

S_EXPIRATION Expiration date for this account is a string in the form MMDDhhmmyy, where MM is the month, DD is the day, hh is the hour in 0 to 24 hour notation, mm is the minutes past the hour, and yy is the last two digits of the year. The attribute type is **SEC_CHAR**.

S_AUTH1 Primary authentication methods for this account. The attribute type is **SEC_LIST**.

S_AUTH2 Secondary authentication methods for this account. The attribute type is **SEC_LIST**.

S_UFSIZE Process file size soft limit. The attribute type is **SEC_INT**.

S_UCPU Process CPU time soft limit. The attribute type is **SEC_INT**.

S_UDATA Process data segment size soft limit. The attribute type is **SEC_INT**.

S_USTACK Process stack segment size soft limit. Type: **SEC_INT**.

S_URSS Process real memory size soft limit. Type: **SEC_INT**.

S_UCORE Process core file size soft limit. The attribute type is **SEC_INT**.

S_UNOFILE Process file descriptor table size soft limit. The attribute type is **SEC_INT**.

S_PWD Specifies the value of the `passwd` field in the **/etc/passwd** file. The attribute type is **SEC_CHAR**.

S_UMASK File creation mask for a user. The attribute type is **SEC_INT**.

S_LOCKED Specifies whether the user's account can be logged into. The attribute type is **SEC_BOOL**.

S_ROLES Defines the administrative roles for this account. The attribute type is **SEC_LIST**.

S_UFSIZE_HARD Process file size hard limit. The attribute type is **SEC_INT**.

S_UCPU_HARD Process CPU time hard limit. The attribute type is **SEC_INT**.

S_UDATA_HARD Process data segment size hard limit. The attribute type is **SEC_INT**.

S_USREXPORT Specifies if the DCE registry can overwrite the local user information with the DCE user information during a DCE export operation. The attribute type is **SEC_BOOL**.

S_USTACK_HARD
Process stack segment size hard limit. Type: **SEC_INT**.

S_URSS_HARD Process real memory size hard limit. Type: **SEC_INT**.

S_UCORE_HARD Process core file size hard limit. The attribute type is **SEC_INT**.

S_UNOFILE_HARD
Process file descriptor table size hard limit. The attribute type is **SEC_INT**.

Note: These values are string constants that should be used by applications both for convenience and to permit optimization in latter implementations.

Additional user–defined attributes may be used and will be stored in the format specified by the *Type* parameter.

Mode	Specifies the search mode. This parameter can be used to delimit the search to one or more user credentials databases. Specifying a non–null *Mode* value also implicitly rewinds the search. A null *Mode* value continues the search sequentially through the database. This parameter must include one of the following values specified as a bit mask; these are defined in the **usersec.h** file:

S_LOCAL	Locally defined users are included in the search.
S_SYSTEM	All credentials servers for the system are searched.

Type	Specifies the type of attribute expected. Valid types are defined in the **usersec.h** file and include:

SEC_INT	The format of the attribute is an integer. For the **getuserattr** subroutine, the user should supply a pointer to a defined integer variable. For the **putuserattr** subroutine, the user should supply an integer.
SEC_CHAR	The format of the attribute is a null–terminated character string. For the **getuserattr** subroutine, the user should supply a pointer to a defined character pointer variable. For the **putuserattr** subroutine, the user should supply a character pointer.
SEC_LIST	The format of the attribute is a series of concatenated strings, each null–terminated. The last string in the series is terminated by two successive null characters. For the **getuserattr** subroutine, the user should supply a pointer to a defined character pointer variable. For the **putuserattr** subroutine, the user should supply a character pointer.
SEC_BOOL	The format of the attribute from **getuserattr** is an integer with the value of either 0 (false) or 1 (true). The format of the attribute for **putuserattr** is a null–terminated string containing one of the following strings: true, false, yes, no, always, or never. For the **getuserattr** subroutine, the user should supply a pointer to a defined integer variable. For the **putuserattr** subroutine, the user should supply a character pointer.
SEC_COMMIT	For the **putuserattr** subroutine, this value specified by itself indicates that changes to the named user are to be committed to permanent storage. The *Attribute* and *Value* parameters are ignored. If no user is specified, the changes to all modified users are committed to permanent storage.
SEC_DELETE	The corresponding attribute is deleted from the database.
SEC_NEW	Updates all the user database files with the new user name when using the **putuserattr** subroutine.

UID	Specifies the user ID to be translated into a user name.

User	Specifies the name of the user for which an attribute is to be read.
Value	Specifies a buffer, a pointer to a buffer, or a pointer to a pointer depending on the *Attribute* and *Type* parameters. See the *Type* parameter for more details.

Security

Files Accessed:

Mode	File
rw	/etc/passwd
rw	/etc/group
rw	/etc/security/user
rw	/etc/security/limits
rw	/etc/security/group
rw	/etc/security/environ

Return Values

If successful, the **getuserattr** subroutine with the **S_LOGINCHK** or **S_RLOGINCHK** attribute specified and the **putuserattr** subroutine return 0. Otherwise, a value of –1 is returned and the **errno** global variable is set to indicate the error. For all other attributes, the **getuserattr** subroutine returns 0.

If successful, the **IDtouser** and **nextuser** subroutines return a character pointer to a buffer containing the requested user name. Otherwise, a null pointer is returned and the **errno** global variable is set to indicate the error.

Error Codes

If any of these subroutines fail, the following is returned:

EACCES	Access permission is denied for the data request.

If the **getuserattr** and **putuserattr** subroutines fail, one or more of the following is returned:

ENOENT	The specified *User* parameter does not exist or the attribute is not defined for this user.
EINVAL	The *Attribute* parameter does not contain one of the defined attributes or null.
EINVAL	The *Value* parameter does not point to a valid buffer or to valid data for this type of attribute. Limited testing is possible and all errors may not be detected.
EPERM	Operation is not permitted.

If the **IDtouser** subroutine fails, one or more of the following is returned:

ENOENT	The *UID* parameter could not be translated into a valid user name on the system.

If the **nextuser** subroutine fails, one or more of the following is returned:

EINVAL	The *Mode* parameter is not one of null, **S_LOCAL**, or **S_SYSTEM**.
EINVAL	The *Argument* parameter is not null.
ENOENT	The end of the search was reached.

Implementation Specifics

These subroutines are part of Base Operating System (BOS) Runtime.

Files

/etc/passwd	Contains user IDs.

Related Information

The **getgroupattr** subroutine, **getuserpw** subroutine, **setpwdb** subroutine, **setuserdb** subroutine.

List of Security and Auditing Subroutines, Subroutines Overview in *AIX General Programming Concepts : Writing and Debugging Programs*.

GetUserAuths Subroutine

Purpose

Accesses the set of authorizations of a user.

Library

Security Library (**libc.a**)

Syntax

```
#include <usersec.h>

char *GetUserAuths(void);
```

Description

The **GetUserAuths** subroutine returns the list of authorizations associated with the real user ID and group set of the process. By default, the ALL authorization is returned for the root user.

Return Values

If successful, the **GetUserAuths** subroutine returns a list of authorizations associated with the user. The format of the list is a series of concatenated strings, each null–terminated. A null string terminates the list. Otherwise, a null pointer is returned and the **errno** global variable is set to indicate the error.

getuserpw, putuserpw, or putuserpwhist Subroutine

Purpose

Accesses the user authentication data.

Library

Security Library (**libc.a**)

Syntax

```
#include <userpw.h>

struct userpw *getuserpw (User)
char *User;

int putuserpw (Password)
struct userpw *Password;

int putuserpwhist (Password, Message)
struct userpw *Password;
char **Message;
```

Description

These subroutines may be used to access user authentication information. Because of their greater granularity and extensibility, you should use them instead of the **getpwent** routines.

The **getuserpw** subroutine reads the user's locally defined password information. If the **setpwdb** subroutine has not been called, the **getuserpw** subroutine will call it as `setpwdb (S_READ)`. This can cause problems if the **putuserpw** subroutine is called later in the program.

The **putuserpw** subroutine updates or creates a locally defined password information stanza in the **/etc/security/passwd** file. The password entry created by the **putuserpw** subroutine is used only if there is an `!` (exclamation point) in the **/etc/passwd** file's `password` field. The user application can use the **putuserattr** subroutine to add an ! to this field.

The **putuserpw** subroutine will open the authentication database read/write if no other access has taken place, but the program should call `setpwdb (S_READ | S_WRITE)` before calling the **putuserpw** subroutine.

The **putuserpwhist** subroutine updates or creates a locally defined password information stanza in the **etc/security/passwd** file. The subroutine also manages a database of previous passwords used for password reuse restriction checking. It is recommended to use the **putuserpwhist** subroutine, rather than the **putuserpw** subroutine, to ensure the password is added to the password history database.

Parameters

Password Specifies the password structure used to update the password information for this user. This structure is defined in the **userpw.h** file and contains the following members:

upw_name Specifies the user's name. (The first eight characters must be unique, since longer names are truncated.)

upw_passwd Specifies the user's password.

upw_lastupdate
Specifies the time, in seconds, since the epoch (that is, 00:00:00 GMT, January 1, 1970), when the password was last updated.

upw_flags Specifies attributes of the password. This member is a bit mask of one or more of the following values, defined in the **userpw.h** file.

PW_NOCHECK
Specifies that new passwords need not meet password restrictions in effect for the system.

PW_ADMCHG Specifies that the password was last set by an administrator and must be changed at the next successful use of the **login** or **su** command**.**

PW_ADMIN Specifies that password information for this user may only be changed by the root user.

Message Indicates a message that specifies an error occurred while updating the password history database. Upon return, the value is either a pointer to a valid string within the memory allocated storage or a null pointer.

User Specifies the name of the user for which password information is read. (The first eight characters must be unique, since longer names are truncated.)

Security

Files Accessed:

Mode	File
rw	**/etc/security/passwd**

Return Values

If successful, the **getuserpw** subroutine returns a valid pointer to a **pw** structure. Otherwise, a null pointer is returned and the **errno** global variable is set to indicate the error.

If successful, the **putuserpwhist** subroutine returns a value of 0. If the subroutine failed to update or create a locally defined password information stanza in the **/etc/security/passwd** file, the **putuserpwhist** subroutine returns a nonzero value. If the subroutine was unable to update the password history database, a message is returned in the *Message* parameter and a return code of 0 is returned.

Error Codes

If the **getuserpw, putuserpw,** and **putuserpwhist** subroutines fail if one of the following values is true:

ENOENT The user does not have an entry in the **/etc/security/passwd** file.

Subroutines invoked by the **getuserpw, putuserpw,** or **putuserpwhist** subroutines can also set errors.

Implementation Specifics

These subroutines are part of Base Operating System (BOS) Runtime.

Files

/etc/security/passwd	Contains user passwords.

Related Information

The **getgroupattr** subroutine, **getuserattr**, **IDtouser**, **nextuser**, or **putuserattr** subroutine, **setpwdb** or **endpwdb** subroutine, **setuserdb** subroutine.

List of Security and Auditing Subroutines and Subroutines Overview in *AIX General Programming Concepts : Writing and Debugging Programs*.

getusraclattr, nextusracl or putusraclattr Subroutine

Purpose

Accesses the user screen information in the SMIT ACL database.

Library

Security Library (**libc.a**)

Syntax

```
#include <usersec.h>

int getusraclattr(User, Attribute, Value, Type)
char *User;
char *Attribute;
void *Value;
int Type;

char *nextusracl(void)

int putusraclattr(User, Attribute, Value, Type)
char *User;
char *Attribute;
void *Value;
int Type;
```

Description

The **getusraclattr** subroutine reads a specified user attribute from the SMIT ACL database. If the database is not already open, this subroutine does an implicit open for reading.

Similarly, the **putusraclattr** subroutine writes a specified attribute into the user SMIT ACL database. If the database is not already open, this subroutine does an implicit open for reading and writing. Data changed by the **putusraclattr** subroutine must be explicitly committed by calling the **putusraclattr** subroutine with a *Type* parameter specifying **SEC_COMMIT**. Until all the data is committed, only the **getusraclattr** subroutine within the process returns written data.

The **nextusracl** subroutine returns the next user in a linear search of the user SMIT ACL database. The consistency of consecutive searches depends upon the underlying storage–access mechanism and is not guaranteed by this subroutine.

The **setacldb** and **endacldb** subroutines should be used to open and close the database.

Parameters

Attribute Specifies which attribute is read. The following possible attributes are defined in the **usersec.h** file:

S_SCREENS String of SMIT screens. The attribute type is **SEC_LIST**.

S_ACLMODE String specifying the SMIT ACL database search scope. The attribute type is **SEC_CHAR**.

S_FUNCMODE String specifying the databases to be searched. The attribute type is **SEC_CHAR**.

Type Specifies the type of attribute expected. Valid types are defined in the **usersec.h** file and include:

SEC_CHAR The format of the attribute is a null–terminated character string.

For the **getusraclattr** subroutine, the user should supply a pointer to a defined character pointer variable. For the **putusraclattr** subroutine, the user should supply a character pointer.

SEC_LIST The format of the attribute is a series of concatenated strings, each null–terminated. The last string in the series must be an empty (zero character count) string.

For the **getusraclattr** subroutine, the user should supply a pointer to a defined character pointer variable. For the **putusraclattr** subroutine, the user should supply a character pointer.

SEC_COMMIT For the **putusraclattr** subroutine, this value specified by itself indicates that changes to the named user are to be committed to permanent storage. The *Attribute* and *Value* parameters are ignored. If no user is specified, the changes to all modified users are committed to permanent storage.

SEC_DELETE The corresponding attribute is deleted from the user SMIT ACL database.

SEC_NEW Updates the user SMIT ACL database file with the new user name when using the **putusraclattr** subroutine.

Value Specifies a buffer, a pointer to a buffer, or a pointer to a pointer depending on the *Attribute* and *Type* parameters. See the *Type* parameter for more details.

Return Values

If successful, the **getusraclattr** returns 0. Otherwise, a value of –1 is returned and the **errno** global variable is set to indicate the error.

Error Codes

Possible return codes are:

EACCES Access permission is denied for the data request.

ENOENT The specified User parameter does not exist or the attribute is not defined for this user.

ENOATTR The specified user attribute does not exist for this user.

EINVAL	The *Attribute* parameter does not contain one of the defined attributes or null.
EINVAL	The *Value* parameter does not point to a valid buffer or to valid data for this type of attribute.
EPERM	Operation is not permitted.

Related Information

The **getgrpaclattr**, **nextgrpacl**, or **putgrpaclattr** subroutine, **setacldb**, or **endacldb** subroutine.

getutent, getutid, getutline, pututline, setutent, endutent, or utmpname Subroutine

Purpose

Accesses **utmp** file entries.

Library

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

Syntax

```
#include <utmp.h>

struct utmp *getutent ( )

struct utmp *getutid (ID)
struct utmp *ID;

struct utmp *getutline (Line)
struct utmp *Line;

void pututline (Utmp)
struct utmp *Utmp;

void setutent ( )

void endutent ( )

void utmpname (File)
char *File;
```

Description

The **getutent**, **getutid**, and **getutline** subroutines return a pointer to a structure of the following type:

```
#define    ut_name   ut_user
#define    ut_id      ut_line
struct utmp
{
char   ut_user[8];     /* User name                      */
char ut_id[14];        /* /etc/inittab ID                */
char   ut_line[12];    /* Device name (console, lnxx) */
short   ut_pid;        /* Process ID                     */
short   ut_type;       /* Type of entry                  */
struct   exit_status
{
short   e_termination;    /* Process termination status    */
short   e_exit;           /* Process exit status           */
} ut_exit;                /* Exit status of a DEAD_PROCESS */
time_t   ut_time;         /* Time entry was made           */
char ut_host[16];         /* Host name                     */
};
```

The **getutent** subroutine reads the next entry from a **utmp**–like file. If the file is not open, this subroutine opens it. If the end of the file is reached, the **getutent** subroutine fails.

The **pututline** subroutine writes the supplied *Utmp* parameter structure into the **utmp** file. It is assumed that the user of the **pututline** subroutine has searched for the proper entry point using one of the **getut** subroutines. If not, the **pututline** subroutine calls **getutid** to search forward for the proper place. If so, **pututline** does not search. If the **pututline** subroutine does not find a matching slot for the entry, it adds a new entry to the end of the file.

The **setutent** subroutine resets the input stream to the beginning of the file. Issue a **setuid** call before each search for a new entry if you want to examine the entire file.

The **endutent** subroutine closes the file currently open.

The **utmpname** subroutine changes the name of a file to be examined from **/etc/utmp** to any other file. The name specified is usually **/var/adm/wtmp**. If the specified file does not exist, no indication is given. You are not aware of this fact until your first attempt to reference the file. The **utmpname** subroutine does not open the file. It closes the old file, if currently open, and saves the new file name.

The most current entry is saved in a static structure. To make multiple accesses, you must copy or use the structure between each access. The **getutid** and **getutline** subroutines examine the static structure first. If the contents of the static structure match what they are searching for, they do not read the **utmp** file. Therefore, you must fill the static structure with zeros after each use if you want to use these subroutines to search for multiple occurrences.

If the **pututline** subroutine finds that it is not already at the correct place in the file, the implicit read it performs does not overwrite the contents of the static structure returned by the **getutent** subroutine, the **getuid** subroutine, or the **getutline** subroutine. This allows you to get an entry with one of these subroutines, modify the structure, and pass the pointer back to the **pututline** subroutine for writing.

These subroutines use buffered standard I/O for input. However, the **pututline** subroutine uses an unbuffered nonstandard write to avoid race conditions between processes trying to modify the **utmp** and **wtmp** files.

Parameters

ID — If you specify a type of **RUN_LVL**, **BOOT_TIME**, **OLD_TIME**, or **NEW_TIME** in the *ID* parameter, the **getutid** subroutine searches forward from the current point in the **utmp** file until an entry with a `ut_type` matching `ID->ut_type` is found.

If you specify a type of **INIT_PROCESS**, **LOGIN_PROCESS**, **USER_PROCESS**, or **DEAD_PROCESS** in the *ID* parameter, the **getutid** subroutine returns a pointer to the first entry whose type is one of these four and whose `ut_id` field matches `Id->ut_id`. If the end of the file is reached without a match, the **getutid** subroutine fails.

Line — The **getutline** subroutine searches forward from the current point in the **utmp** file until it finds an entry of type **LOGIN_PROCESS** or **USER_PROCESS** that also has a `ut_line` string matching the `Line->ut_line` parameter string. If the end of file is reached without a match, the **getutline** subroutine fails.

Utmp — Points to the **utmp** structure.

File — Specifies the name of the file to be examined.

Return Values

These subroutines fail and return a null pointer if a read or write fails due to a permission conflict or because the end of the file is reached.

Files

/etc/utmp	Path to the **utmp** file, which contains a record of users logged into the system.
/var/adm/wtmp	Path to the **wtmp** file, which contains accounting information about users logged in.

Implementation Specifics

These subroutines are part of Base Operating System (BOS) Runtime.

Related Information

The **ttyslot** subroutine.

The **failedlogin**, **utmp**, or **wtmp** file.

getvfsent, getvfsbytype, getvfsbyname, getvfsbyflag, setvfsent, or endvfsent Subroutine

Purpose

Gets a **vfs** file entry.

Library

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

Syntax

```
#include <sys/vfs.h>
#include <sys/vmount.h>

struct vfs_ent *getvfsent( )

struct vfs_ent *getvfsbytype(vfsType)
int vfsType;

struct vfs_ent *getvfsbyname(vfsName)
char *vfsName;

struct vfs_ent *getvfsbyflag(vfsFlag)
int vfsFlag;

void setvfsent( )

void endvfsent( )
```

Description

Attention: All information is contained in a static area and so must be copied to be saved.

The **getvfsent** subroutine, when first called, returns a pointer to the first **vfs_ent** structure in the file. On the next call, it returns a pointer to the next **vfs_ent** structure in the file. Successive calls are used to search the entire file.

The **vfs_ent** structure is defined in the **vfs.h** file and it contains the following fields:

```
char vfsent_name;
int vfsent_type;
int vfsent_flags;
char *vfsent_mnt_hlpr;
char *vfsent_fs_hlpr;
```

The **getvfsbytype** subroutine searches from the beginning of the file until it finds a **vfs** type matching the *vfsType* parameter. The subroutine then returns a pointer to the structure in which it was found.

The **getvfsbyname** subroutine searches from the beginning of the file until it finds a **vfs** name matching the *vfsName* parameter. The search is made using flattened names; the search–string uses ASCII equivalent characters.

The **getvfsbytype** subroutine searches from the beginning of the file until it finds a type matching the *vfsType* parameter.

The **getvfsbyflag** subroutine searches from the beginning of the file until it finds the entry whose flag corresponds flags defined in the **vfs.h** file. Currently, these are **VFS_DFLT_LOCAL** and **VFS_DFLT_REMOTE**.

The **setvfsent** subroutine rewinds the **vfs** file to allow repeated searches.

The **endvfsent** subroutine closes the **vfs** file when processing is complete.

Parameters

vfsType	Specifies a **vfs** type.
vfsName	Specifies a **vfs** name.
vfsFlag	Specifies either **VFS_DFLT_LOCAL** or **VFS_DFLT_REMOTE**.

Return Values

The **getvfsent**, **getvfsbytype**, **getvfsbyname**, and **getvfsbyflag** subroutines return a pointer to a **vfs_ent** structure containing the broken–out fields of a line in the /**etc**/**vfs** file. If an end–of–file character or an error is encountered on reading, a null pointer is returned.

Implementation Specifics

These subroutines are part of Base Operating System (BOS) Runtime.

Files

/**etc**/**vfs**	Describes the virtual file system (VFS) installed on the system.

Related Information

The **getfsent**, **getfsspec**, **getfsfile**, **getfstype**, **setfsent**, or **endfsent** subroutine.

National Language Support Overview for Programming in *AIX General Programming Concepts : Writing and Debugging Programs*.

getwc, fgetwc, or getwchar Subroutine

Purpose

Gets a wide character from an input stream.

Library

Standard I/O Package (**libc.a**)

Syntax

```
#include <stdio.h>

win_t getwc (Stream)
FILE *Stream;

win_t fgetwc (Stream)
FILE *Stream;

win_t getwchar (void)
```

Description

The **fgetwc** subroutine obtains the next wide character from the input stream specified by the *Stream* parameter, converts it to the corresponding wide character code, and advances the file position indicator the number of bytes corresponding to the obtained multibyte character. The **getwc** subroutine is equivalent to the **fgetwc** subroutine, except that when implemented as a macro, it may evaluate the *Stream* parameter more than once. The **getwchar** subroutine is equivalent to the **getwc** subroutine with **stdin** (the standard input stream).

The first successful run of the **fgetc**, **fgets**, **fgetwc**, **fgetws**, **fread**, **fscanf**, **getc**, **getchar**, **gets**, or **scanf** subroutine using a stream that returns data not supplied by a prior call to the **ungetc** or **ungetwc** subroutine marks the `st_atime` field for update.

Parameters

Stream	Specifies input data.

Return Values

Upon successful completion, the **getwc** and **fgetwc** subroutines return the next wide character from the input stream pointed to by the *Stream* parameter. The **getwchar** subroutine returns the next wide character from the input stream pointed to by stdin.

If the end of the file is reached, an indicator is set and **WEOF** is returned. If a read error occurs, an error indicator is set, **WEOF** is returned, and the **errno** global variable is set to indicate the error.

Error Codes

If the **getwc**, **fgetwc**, or **getwchar** subroutine is unsuccessful because the stream is not buffered or data needs to be read into the buffer, it returns one of the following error codes:

EAGAIN	Indicates that the **O_NONBLOCK** flag is set for the file descriptor underlying the *Stream* parameter, delaying the process.
EBADF	Indicates that the file descriptor underlying the *Stream* parameter is not valid and cannot be opened for reading.
EINTR	Indicates that the process has received a signal that terminates the read operation.

EIO	Indicates that a physical error has occurred, or the process is in a background process group attempting to read from the controlling terminal, and either the process is ignoring or blocking the **SIGTTIN** signal or the process group is orphaned.
EOVERFLOW	Indicates that the file is a regular file and an attempt has been made to read at or beyond the offset maximum associated with the corresponding stream.

The **getwc**, **fgetwc**, or **getwchar** subroutine is also unsuccessful due to the following error conditions:

ENOMEM	Indicates that storage space is insufficient.
ENXIO	Indicates that the process sent a request to a nonexistent device, or the device cannot handle the request.
EILSEQ	Indicates that the **wc** wide–character code does not correspond to a valid character.

Implementation Specifics

These subroutines are part of Base Operating System (BOS) Runtime.

Related Information

Other wide character I/O subroutines: **getws** or **fgetws** subroutine, **putwc**, **putwchar**, or **fputwc** subroutine, **putws** or **fputws** subroutine, **ungetwc** subroutine.

Related standard I/O subroutines: **fopen**, **freopen**, or **fdopen** subroutine, **gets** or **fgets** subroutine, **fread** subroutine, **fwrite** subroutine, **printf**, **fprintf**, **sprintf**, **wsprintf**, **vprintf**, **vfprintf**, **vsprintf**, or **vwsprintf** subroutine, **putc**, **putchar**, **fputc**, or **putw** subroutine, **puts** or **fputs** subroutine.

National Language Support Overview for Programming, Subroutines Overview, Understanding Wide Character Input/Output Subroutines in *AIX General Programming Concepts : Writing and Debugging Programs*.

getwd Subroutine

Purpose

Gets current directory path name.

Library

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

Syntax

```
#include <unistd.h>

char *getwd (PathName)
char *PathName;
```

Description

The **getwd** subroutine determines the absolute path name of the current directory, then copies that path name into the area pointed to by the *PathName* parameter.

The maximum path–name length, in characters, is set by the **PATH_MAX** value, as specified in the **limits.h** file.

Parameters

PathName	Points to the full path name.

Return Values

If the call to the **getwd** subroutine is successful, a pointer to the absolute path name of the current directory is returned. If an error occurs, the **getwd** subroutine returns a null value and places an error message in the *PathName* parameter.

Implementation Specifics

This subroutine is part of Base Operating System (BOS) Runtime.

Related Information

The **getcwd** subroutine.

Files, Directories, and File Systems for Programmers in *AIX General Programming Concepts : Writing and Debugging Programs*.

getws or fgetws Subroutine

Purpose

Gets a string from a stream.

Library

Standard I/O Library (**libc.a**)

Syntax

```
#include <stdio.h>

wchar_t *fgetws (WString, Number, Stream)
wchar_t *WString;
int Number;
FILE *Stream;

wchar_t *getws (WString)
wchar_t *WString;
```

Description

The **fgetws** subroutine reads characters from the input stream, converts them to the corresponding wide character codes, and places them in the array pointed to by the *WString* parameter. The subroutine continues until either the number of characters specified by the *Number* parameter minus 1 are read or the subroutine encounters a new–line or end–of–file character. The **fgetws** subroutine terminates the wide character string specified by the *WString* parameter with a null wide character.

The **getws** subroutine reads wide characters from the input stream pointed to by the standard input stream (**stdin**) into the array pointed to by the *WString* parameter. The subroutine continues until it encounters a new–line or the end–of–file character, then it discards any new–line character and places a null wide character after the last character read into the array.

Parameters

WString	Points to a string to receive characters.
Stream	Points to the **FILE** structure of an open file.
Number	Specifies the maximum number of characters to read.

Return Values

If the **getws** or **fgetws** subroutine reaches the end of the file without reading any characters, it transfers no characters to the *String* parameter and returns a null pointer. If a read error occurs, the **getws** or **fgetws** subroutine returns a null pointer and sets the **errno** global variable to indicate the error.

Error Codes

If the **getws** or **fgetws** subroutine is unsuccessful because the stream is not buffered or data needs to be read into the stream's buffer, it returns one or more of the following error codes:

EAGAIN	Indicates that the **O_NONBLOCK** flag is set for the file descriptor underlying the *Stream* parameter, and the process is delayed in the **fgetws** subroutine.
EBADF	Indicates that the file descriptor specifying the *Stream* parameter is not a read–access file.

EINTR	Indicates that the read operation is terminated due to the receipt of a signal, and either no data was transferred or the implementation does not report partial transfer for this file.
EIO	Indicates that insufficient storage space is available.
ENOMEM	Indicates that insufficient storage space is available.
EILSEQ	Indicates that the data read from the input stream does not form a valid character.

Implementation Specifics

These subroutines are part of Base Operating System (BOS) Runtime.

Related Information

Other wide character I/O subroutines: **fgetwc** subroutine, **fputwc** subroutine, **fputws** subroutine, **getwc** subroutine, **getwchar** subroutine, **putwc** subroutine, **putwchar** subroutine, **putws** subroutine, **ungetwc** subroutine.

Related standard I/O subroutines: **fdopen** subroutine, **fgetc** subroutine, **fgets** subroutine, **fopen** subroutine, **fprintf** subroutine, **fputc** subroutine, **fputs** subroutine, **fread** subroutine, **freopen** subroutine, **fscanf** subroutine, **fwrite** subroutine, **getc** subroutine, **getchar** subroutine, **gets** subroutine, **printf** subroutine, **putc** subroutine, **putchar** subroutine, **puts** subroutine, **putw** subroutine, **scanf** subroutine, **sprintf** subroutine, **ungetc** subroutine.

National Language Support Overview for Programming, Understanding Wide Character Input/Output Subroutines, Subroutines Overview in *AIX General Programming Concepts : Writing and Debugging Programs*.

glob Subroutine

Purpose

Generates path names.

Library

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

Syntax

```
#include <glob.h>

int glob (Pattern, Flags, (Errfunc)(), Pglob)
const char *Pattern;
int Flags;
int *Errfunc (Epath, Eerrno)
const char *Epath;
int Eerrno;
glob_t *Pglob;
```

Description

The **glob** subroutine constructs a list of accessible files that match the *Pattern* parameter.

The **glob** subroutine matches all accessible path names against this pattern and develops a list of all matching path names. To have access to a path name, the **glob** subroutine requires search permission on every component of a path except the last, and read permission on each directory of any file name component of the *Pattern* parameter that contains any of the special characters * (asterisk), ? (question mark), or [(left bracket). The **glob** subroutine stores the number of matched path names and a pointer to a list of pointers to path names in the *Pglob* parameter. The path names are in sort order, based on the setting of the **LC_COLLATE** category in the current locale. The first pointer after the last path name is a null character. If the pattern does not match any path names, the returned number of matched paths is zero.

Parameters

Pattern Contains the file name pattern to compare against accessible path names.

Flags Controls the customizable behavior of the **glob** subroutine.

The *Flags* parameter controls the behavior of the **glob** subroutine. The *Flags* value is the bitwise inclusive OR of any of the following constants, which are defined in the **glob.h** file:

GLOB_APPEND Appends path names located with this call to any path names previously located. If the **GLOB_APPEND** constant is not set, new path names overwrite previous entries in the *Pglob* array. The **GLOB_APPEND** constant should not be set on the first call to the **glob** subroutine. It may, however, be set on subsequent calls.

The **GLOB_APPEND** flag can be used to append a new set of path names to those found in a previous call to the **glob** subroutine. If the **GLOB_APPEND** flag is specified in the *Flags* parameter, the following rules apply:

- If the application sets the **GLOB_DOOFFS** flag in the first call to the **glob** subroutine, it is also set in the second. The value of the *Pglob* parameter is not modified between the calls.
- If the application did not set the **GLOB_DOOFFS** flag in the first call to the **glob** subroutine, it is not set in the second.
- After the second call, the *Pglob* parameter points to a list containing the following:
  - Zero or more null characters, as specified by the **GLOB_DOOFFS** flag.
  - Pointers to the path names that were in the *Pglob* list before the call, in the same order as after the first call to the **glob** subroutine.
  - Pointers to the new path names generated by the second call, in the specified order.
- The count returned in the *Pglob* parameter is the total number of path names from the two calls.
- The application should not modify the *Pglob* parameter between the two calls.

It is the caller's responsibility to create the structure pointed to by the *Pglob* parameter. The **glob** subroutine allocates other space as needed.

GLOB_DOOFFS Uses the **gl_offs** structure to specify the number of null pointers to add to the beginning of the **gl_pathv** component of the *Pglob* parameter.

GLOB_ERR Causes the **glob** subroutine to return when it encounters a directory that it cannot open or read. If the **GLOB_ERR** flag is not set, the **glob** subroutine continues to find matches if it encounters a directory that it cannot open or read.

GLOB_MARK	Specifies that each path name that is a directory should have a / (slash) appended.
GLOB_NOCHECK	If the *Pattern* parameter does not match any path name, the **glob** subroutine returns a list consisting only of the *Pattern* parameter, and the number of matched patterns is one.
GLOB_NOSORT	Specifies that the list of path names need not be sorted. If the **GLOB_NOSORT** flag is not set, path names are collated according to the current locale.
GLOB_QUOTE	If the **GLOB_QUOTE** flag is set, a \ (backslash) can be used to escape metacharacters.

Errfunc Specifies an optional subroutine that, if specified, is called when the **glob** subroutine detects an error condition.

Pglob Contains a pointer to a **glob_t** structure. The structure is allocated by the caller. The array of structures containing the file names matching the *Pattern* parameter are defined by the **glob** subroutine. The last entry is a null pointer.

Epath Specifies the path that failed because a directory could not be opened or read.

Eerrno Specifies the **errno** value of the failure indicated by the *Epath* parameter. This value is set by the **opendir**, **readdir**, or **stat** subroutines.

Return Values

On successful completion, the **glob** subroutine returns a value of 0. The *Pglob* parameter returns the number of matched path names and a pointer to a null–terminated list of matched and sorted path names. If the number of matched path names in the *Pglob* parameter is zero, the pointer in the *Pglob* parameter is undefined.

Error Codes

If the **glob** subroutine terminates due to an error, it returns one of the nonzero constants below. These are defined in the **glob.h** file. In this case, the *Pglob* values are still set as defined in the Return Values section.

GLOB_ABORTED	Indicates the scan was stopped because the **GLOB_ERROR** flag was set or the subroutine specified by the **errfunc** parameter returned a nonzero value.
GLOB_NOSPACE	Indicates a failed attempt to allocate memory.

If, during the search, a directory is encountered that cannot be opened or read and the *Errfunc* parameter is not a null value, the **glob** subroutine calls the subroutine specified by the **errfunc** parameter with two arguments:

- The *Epath* parameter specifies the path that failed.
- The *Eerrno* parameter specifies the value of the **errno** global variable from the failure, as set by the **opendir**, **readdir**, or **stat** subroutine.

If the subroutine specified by the *Errfunc* parameter is called and returns nonzero, or if the **GLOB_ERR** flag is set in the *Flags* parameter, the **glob** subroutine stops the scan and returns **GLOB_ABORTED** after setting the *Pglob* parameter to reflect the paths already scanned. If **GLOB_ERR** is not set and either the *Errfunc* parameter is null or `*errfunc` returns zero, the error is ignored.

The *Pglob* parameter has meaning even if the **glob** subroutine fails. Therefore, the **glob** subroutine can report partial results in the event of an error. However, if the number of matched path names is 0, the pointer in the *Pglob* parameter is unspecified even if the **glob** subroutine did not return an error.

Examples

The **GLOB_NOCHECK** flag can be used with an application to expand any path name using wildcard characters. However, the **GLOB_NOCHECK** flag treats the pattern as just a string by default. The **sh** command can use this facility for option parameters, for example.

The **GLOB_DOOFFS** flag can be used by applications that build an argument list for use with the **execv**, **execve**, or **execvp** subroutine. For example, an application needs to do the equivalent of `ls -l *.c`, but for some reason cannot. The application could still obtain approximately the same result using the sequence:

```
globbuf.gl_offs = 2;
glob ("*.c", GLOB_DOOFFS, NULL, &globbuf);
globbuf.gl_pathv[0] = "ls";
globbuf.gl_pathv[1] ="-l";
execvp ("ls", &globbuf.gl_pathv[0]);
```

Using the same example, `ls -l *.c *.h` could be approximated using the **GLOB_APPEND** flag as follows:

```
globbuf.gl_offs = 2;
glob ("*.c", GLOB_DOOFFS, NULL, &globbuf);
glob ("*.h", GLOB_DOOFFS|GLOB_APPEND, NULL, &globbuf);
```

The new path names generated by a subsequent call with the **GLOB_APPEND** flag set are not sorted together with the previous path names. This is the same way the shell handles path name expansion when multiple expansions are done on a command line.

Implementation Specifics

This subroutine is part of Base Operating System (BOS) Runtime.

Related Information

The **exec: execl**, **execv**, **execle**, **execve**, **execlp**, **execvp**, or **exect** subroutine, **fnmatch** subroutine, **opendir**, **readdir**, **telldir**, **seekdir**, **rewinddir**, or **closedir** subroutine, **statx**, **stat**, **lstat**, **fstatx**, **fstat**, **fullstat**, or **ffullstat** subroutine.

The **ls** command.

National Language Support Overview for Programming in *AIX General Programming Concepts : Writing and Debugging Programs*.

globfree Subroutine

Purpose

Frees all memory associated with the *pglob* parameter.

Library

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

Syntax

```
#include <glob.h>

void globfree (pglob)
glob_t *pglob;
```

Description

The **globfree** subroutine frees any memory associated with the *pglob* parameter due to a previous call to the **glob** subroutine.

Parameters

pglob	Structure containing the results of a previous call to the **glob** subroutine.

Implementation Specifics

This subroutine is part of Base Operating System (BOS) Runtime.

Related Information

The **glob** subroutine.

National Language Support Overview for Programming in *AIX General Programming Concepts : Writing and Debugging Programs*.

grantpt Subroutine

Purpose

Changes the mode and ownership of a pseudo–terminal device.

Library

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

Syntax

```
#include <stdlib.h>

int grantpt (FileDescriptor)
int FileDescriptor;
```

Description

The **grantpt** subroutine changes the mode and the ownership of the slave pseudo–terminal associated with the master pseudo–terminal device defined by the *FileDescriptor* parameter. The user ID of the slave pseudo–terminal is set to the real UID of the calling process. The group ID of the slave pseudo–terminal is set to an unspecified group ID. The permission mode of the slave pseudo–terminal is set to readable and writeable by the owner, and writeable by the group.

Parameters

FileDescriptor	Specifies the file descriptor of the master pseudo–terminal device.

Return Value

Upon successful completion, a value of 0 is returned. Otherwise, a value of –1 is returned and the **errno** global variable is set to indicate the error.

Error Codes

The **grantpt** function may fail if:

EBADF	The *fildes* argument is not a valid open file descriptor.
EINVAL	The *fildes* argument is not associated with a master pseudo–terminal device.
EACCES	The corresponding slave pseudo–terminal device could not be accessed.

Implementation Specifics

This subroutine is part of Base Operating System (BOS) Runtime.

Related Information

The **unlockpt** subroutine.

The Input and Output Handling Programmer's Overview in *AIX General Programming Concepts : Writing and Debugging Programs*.

hsearch, hcreate, or hdestroy Subroutine

Purpose

Manages hash tables.

Library

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

Syntax

```
#include <search.h>


ENTRY *hsearch (Item, Action)
ENTRY Item;
Action Action;

int hcreate (NumberOfElements)
size_t NumberOfElements;
void hdestroy ( )
```

Description

Attention: Do not use the **hsearch**, **hcreate**, or **hdestroy** subroutine in a multithreaded environment.

The **hsearch** subroutine searches a hash table. It returns a pointer into a hash table that indicates the location of the given item. The **hsearch** subroutine uses open addressing with a multiplicative hash function.

The **hcreate** subroutine allocates sufficient space for the table. You must call the **hcreate** subroutine before calling the **hsearch** subroutine. The *NumberOfElements* parameter is an estimate of the maximum number of entries that the table will contain. This number may be adjusted upward by the algorithm in order to obtain certain mathematically favorable circumstances.

The **hdestroy** subroutine deletes the hash table. This action allows you to start a new hash table since only one table can be active at a time. After the call to the **hdestroy** subroutine, the data can no longer be considered accessible.

Parameters

Item — Identifies a structure of the type **ENTRY** as defined in the **search.h** file. It contains two pointers:

- **Item.key** — Points to the comparison key. The key field is of the **char** type.
- **Item.data** — Points to any other data associated with that key. The data field is of the **void** type.

Pointers to data types other than the **char** type should be declared to pointer–to–character.

Action — Specifies the value of the *Action* enumeration parameter that indicates what is to be done with an entry if it cannot be found in the table. Values are:

- **ENTER** — Enters the value of the *Item* parameter into the table at the appropriate point. If the table is full, the **hsearch** subroutine returns a null pointer.
- **FIND** — Does not enter the value of the *Item* parameter into the table. If the value of the *Item* parameter cannot be found, the **hsearch** subroutine returns a null pointer. If the value of the *Item* parameter is found, the subroutine returns the address of the item in the hash table.

NumberOfElements — Provides an estimate of the maximum number of entries that the table contains. Under some circumstances, the **hcreate** subroutine may actually make the table larger than specified.

Return Values

The **hcreate** subroutine returns a value of 0 if it cannot allocate sufficient space for the table.

Implementation Specifics

These subroutines are part of Base Operating System (BOS) Runtime.

Related Information

The **bsearch** subroutine, **lsearch** subroutine, **malloc** subroutine, **strcmp** subroutine, **tsearch** subroutine.

Searching and Sorting Example Program and Subroutines Overview in *AIX General Programming Concepts : Writing and Debugging Programs*.

hypot Subroutine

Purpose

Computes the Euclidean distance function and complex absolute value.

Libraries

IEEE Math Library (**libm.a**)
System V Math Library (**libmsaa.a**)

Syntax

```
#include <math.h>

double hypot (x, y)
double x, y;
```

Description

Computes the square root of (*x***2 + *y***2) so that underflow does not occur and overflow occurs only if the final result warrants it.

Note: Compile any routine that uses subroutines from the **libm.a** library with the **–lm** flag. To compile the **hypot.c** file, for example:

```
cc hypot.c -lm
```

Parameters

x	Specifies some double–precision floating–point value.
y	Specifies some double–precision floating–point value.
z	Specifies a structure that has two double elements (*z* = *x*i + *y*j).

Error Codes

When using the **libm.a** (**–lm**) library, if the correct value overflows, the **hypot** subroutine returns a **HUGE_VAL** value.

Note: **(hypot (INF,** *value*) and **hypot (***value***,** INF**)** are both equal to **+INF** for all values, even if *value* = NaN.

When using **libmsaa.a** (**–lmsaa**), if the correct value overflows, the **hypot** subroutine returns **HUGE_VAL** and sets the global variable **errno** to **ERANGE**.

These error–handling procedures may be changed with the **matherr** subroutine when using the **libmsaa.a** (**–lmsaa**) library.

Implementation Specifics

These subroutines are part of Base Operating System (BOS) Runtime.

Related Information

The **matherr** subroutine, **sqrt** subroutine.

Subroutines Overview in *AIX General Programming Concepts : Writing and Debugging Programs*.

iconv_close Subroutine

Purpose

Closes a specified code set converter.

Library

iconv Library (**libiconv.a**)

Syntax

```
#include <iconv.h>

int iconv_close (CD)
iconv_t CD;
```

Description

The **iconv_close** subroutine closes a specified code set converter and deallocates any resources used by the converter.

Parameters

CD	Specifies the conversion descriptor to be closed.

Return Values

When successful, the **iconv_close** subroutine returns a value of 0. Otherwise, it returns a value of –1 and sets the **errno** global variable to indicate the error.

Error Codes

The following error code is defined for the **iconv_close** subroutine:

EBADF	The conversion descriptor is not valid.

Implementation Specifics

This command is part of Base Operating System (BOS) Runtime.

Related Information

The **iconv** subroutine, **iconv_open** subroutine.

The **genxlt** command, **iconv** command.

Converters Overview for Programming and the National Language Support Overview for Programming in *AIX General Programming Concepts : Writing and Debugging Programs*.

iconv Subroutine

Purpose

Converts a string of characters in one character code set to another character code set.

Library

The **iconv** Library (**libiconv.a**)

Syntax

```
#include <iconv.h>

size_t iconv (CD, InBuf, InBytesLeft, OutBuf, OutBytesLeft)
iconv_t CD;
char **OutBuf, **InBuf;
size_t *OutBytesLeft, *InBytesLeft;
```

Description

The **iconv** subroutine converts the string specified by the *InBuf* parameter into a different code set and returns the results in the *OutBuf* parameter. The required conversion method is identified by the *CD* parameter, which must be valid conversion descriptor returned by a previous, successful call to the **iconv_open** subroutine.

On calling, the *InBytesLeft* parameter indicates the number of bytes in the *InBuf* buffer to be converted, and the *OutBytesLeft* parameter indicates the number of available bytes in the *OutBuf* buffer. These values are updated upon return so they indicate the new state of their associated buffers.

For state–dependent encodings, calling the **iconv** subroutine with the *InBuf* buffer set to null will reset the conversion descriptor in the *CD* parameter to its initial state. Subsequent calls with the *InBuf* buffer, specifying other than a null pointer, may cause the internal state of the subroutine to be altered a necessary.

Parameters

CD	Specifies the conversion descriptor that points to the correct code set converter.
InBuf	Points to a buffer that contains the number of bytes in the *InBytesLeft* parameter to be converted.
InBytesLeft	Points to an integer that contains the number of bytes in the *InBuf* parameter.
OutBuf	Points to a buffer that contains the number of bytes in the *OutBytesLeft* parameter that has been converted.
OutBytesLeft	Points to an integer that contains the number of bytes in the *OutBuf* parameter.

Return Values

Upon successful conversion of all the characters in the *InBuf* buffer and after placing the converted characters in the *OutBuf* buffer, the **iconv** subroutine returns 0, updates the *InBytesLeft* and *OutBytesLeft* parameters, and increments the *InBuf* and *OutBuf* pointers. Otherwise, it updates the varibles pointed to by the parameters to indicate the extent to the conversion, returns the number of bytes still left to be converted in the input buffer, and sets the **errno** global variable to indicate the error.

Error Codes

If the **iconv** subroutine is unsuccessful, it updates the variables to reflect the extent of the conversion before it stopped and sets the **errno** global variable to one of the following values:

EILSEQ	Indicates an unusable character. If an input character does not belong to the input code set, no conversion is attempted on the unusable on the character. In *InBytesLeft* parameters indicates the bytes left to be converted, including the first byte of the unusable character. *InBuf* parameter points to the first byte of the unusable character sequence. The values of *OutBuf* and *OutBytesLeft* are updated according to the number of bytes that were previously converted.
E2BIG	Indicates an output buffer overflow. If the *OutBuf* buffer is too small to contain all the converted characters, the character that causes the overflow is not converted. The *InBytesLeft* parameter indicates the bytes left to be converted (including the character that caused the overflow). The *InBuf* parameter points to the first byte of the characters left to convert.
EINVAL	Indicates the input buffer was truncated. If the original value of *InBytesLeft* is exhausted in the middle of a character conversion or shift/lock block, the *InBytesLeft* parameter indicates the number of bytes undefined in the character being converted. If an input character of shift sequence is truncated by the *InBuf* buffer, no conversion is attempted on the truncated data, and the *InBytesLeft* parameter indicates the bytes left to be converted. The *InBuf* parameter points to the first bytes if the truncated sequence. The *OutBuf* and *OutBytesLeft* values are updated according to the number of characters that were previously0 converted. Because some encoding may have ambiguous data, the **EINVAL** return value has a special meaning at the end of stream conversion. As such, if a user detects an EOF character on a stream that is being converted and the last return code from the **iconv** subroutine was **EINVAL**, the **iconv** subroutine should be called again, with the same *InBytesLeft* parameter and the same character string pointed to by the *InBuf* parameter as when the **EINVAL** return occurred. As a result, the converter will either convert the string as is or declare it an unusable sequence (**EILSEQ**).

Implementation Specifics

The **iconv** subroutine is part of Base Operating System (BOS) Runtime.

Files

/**usr**/**lib**/**nls**/**loc**/**iconv**/*	Contains code set converter methods.

Related Information

The **iconv** command, **genxlt** command.

The **iconv_close** subroutine, **iconv_open** subroutine.

iconv_open Subroutine

Purpose

Opens a character code set converter.

Library

iconv Library (**libiconv.a**)

Syntax

```
#include <iconv.h>

iconv_t iconv_open (ToCode, FromCode)
const char *ToCode, *FromCode;
```

Description

The **iconv_open** subroutine initializes a code set converter. The code set converter is used by the **iconv** subroutine to convert characters from one code set to another. The **iconv_open** subroutine finds the converter that performs the character code set conversion specified by the *FromCode* and *ToCode* parameters, initializes that converter, and returns a conversion descriptor of type **iconv_t** to identify the code set converter.

The **iconv_open** subroutine first searches the **LOCPATH** environment variable for a converter, using the two user–provided code set names, based on the file name convention that follows:

```
FromCode: ”IBM-850”
ToCode: ”ISO8859-1”
conversion file: ”IBM-850_ISO8859-1”
```

The conversion file name is formed by concatenating the *ToCode* code set name onto the *FromCode* code set name, with an _ (underscore) between them.

The **LOCPATH** environment variable contains a list of colon–separated directory names. The system default for the **LOCPATH** environment variable is:

```
LOCPATH=/usr/lib/nls/loc
```

See the ”Locale Overview for System Management” in *AIX 4.3 System Management Guide: Operating System and Devices* for more information on the **LOCPATH** environment variable.

The **iconv_open** subroutine first attempts to find the specified converter in an **iconv** subdirectory under any of the directories specified by the **LOCPATH** environmental variable, for example, **/usr/lib/nls/loc/iconv**. If the **iconv_open** subroutine cannot find a converter in any of these directories, it looks for a conversion table in an **iconvTable** subdirectory under any of the directories specified by the **LOCPATH** environment variable, for example, **/usr/lib/nls/loc/iconvTable**.

If the **iconv_open** subroutine cannot find the specified converter in either of these locations, it returns (**iconv_t**) –1 to the calling process and sets the **errno** global variable.

The **iconvTable** directories are expected to contain conversion tables that are the output of the **genxlt** command. The conversion tables are limited to single–byte stateless code sets. See the ”List of PC, ISO, and EBCDIC Code Set Converters” in *AIX General Programming Concepts : Writing and Debugging Programs* for more information.

If the named converter is found, the **iconv_open** subroutine will perform the **load** subroutine operation and initialize the converter. A converter descriptor (**iconv_t**) is returned.

Note: When a process calls the **exec** subroutine or a **fork** subroutine, all of the opened converters are discarded.

The **iconv_open** subroutine links the converter function using the **load** subroutine, which is similar to the **exec** subroutine and effectively performs a run–time linking of the converter program. Since the **iconv_open** subroutine is called as a library function, it must ensure that security is preserved for certain programs. Thus, when the **iconv_open** subroutine is called from a set root ID program (a program with permission **——s—s—x**), it will ignore the **LOCPATH** environment variable and search for converters only in the **/usr/lib/nls/loc/iconv** directory.

Parameters

ToCode	Specifies the destination code set.
FromCode	Specifies the originating code set.

Return Values

A conversion descriptor (**iconv_t**) is returned if successful. Otherwise, the subroutine returns –1, and the **errno** global variable is set to indicate the error.

Error Codes

EINVAL	The conversion specified by the *FromCode* and *ToCode* parameters is not supported by the implementation.
EMFILE	The number of file descriptors specified by the **OPEN_MAX** configuration variable is currently open in the calling process.
ENFILE	Too many files are currently open in the system.
ENOMEM	Insufficient storage space is available.

Implementation Specifics

This command is part of Base Operating System (BOS) Runtime.

Files

/usr/lib/nls/loc/iconv	Contains loadable method converters.
/usr/lib/nls/loc/iconvTable	Contains conversion tables for single–byte stateless code sets.

Related Information

The **iconv** subroutine, **iconv_close** subroutine.

The **genxlt** command, **iconv** command.

Code Set Overview in *AIX Kernel Extensions and Device Support Programming Concepts*.

The List of PC, ISO, and EBCDIC Code Set Converters, the National Language Support Overview for Programming, Converters Overview for Programming in *AIX General Programming Concepts : Writing and Debugging Programs*.

if_freenameindex Subroutine

Purpose Frees memory allocated by if_nameindex

Library

Library (**libinet.a**)

Syntax

#include <net/if.h>

void if_freenameindex (struct if_nameindex *ptr);

Description

The argument to this function must be a pointer that was returned by **if_nameindex**.

Related Information

The **if_nametoindex** subroutine, **if_indextoname** subroutine, and **if_nameindex** subroutine.

Subroutines Overview in *AIX Version 4 General Programming Concepts: Writing and Debugging Programs*.

if_indextoname Subroutine

Purpose

Determines the interface name associated with a particular index. The second of four functions, **if_indextoname** maps an interface index into its corresponding name.

Library

Library (**libinet.a**)

Syntax

```
#include <net/if.h>>
char *
if_indextoname (index, ifname)
unsigned int index;
char *ifname;
```

Description

The second of four functions for Interface Identification. The first argument is the interface index whose name is to be retrieved. The second argument is a buffer of at least IFNAMSIZ bytes, into which the name is to be copied.

Note: The **if_indextoname** argument must point to a buffer of at least *IFNAMESIZ* bytes into which the interface name corresponding to the specified index is returned. *IFNAMSIZ* is also defined in <net/if.h> and its value includes a terminating null byte at the end of the interface name.

Return Values

If successful, **if_indextoname** returns a pointer to a valid name corresponding to the specified index. A null pointer is returned if no interface name corresponds to the specified index.

Related Information

The **if_nametoindex** subroutine, **if_nameindex** subroutine, and **if_freenameindex** subroutine.

Subroutines Overview in *AIX Version 4 General Programming Concepts: Writing and Debugging Programs*.

if_nameindex Subroutine

Purpose

Retrieves index and name information for *all* interfaces.

Library

Library (**libinet.a**)

Syntax

```
#include <net/if.h>

struct if_nameindex {
unsigned int if_index; /* 1, 2, ... */
char * if_name; /* null terminated name: "le0", ... */
};
struct if_nameindex *if_nameindex (void);
```

Description

The final function of four for interface identification. The **if_nameindex** subroutine returns an array of **if_nameindex** structures, one structure per interface.

Note: The memory used for this array of structures along with the interface names pointed to by the **if_name** members is obtained dynamically. Use **if_freenameindex** to free memory allocated by **if_nameindex**.

Return Values

If successful, the end of the array of structures is indicated by a structure with an **if_index** of 0 and an **if_name** of NULL. The function returns a NULL pointer upon an error.

Related Information

The **if_nametoindex** subroutine, **if_indextoname** subroutine, and **if_freenameindex** subroutine.

Subroutines Overview in *AIX Version 4 General Programming Concepts: Writing and Debugging Programs*.

if_nametoindex Subroutine

Purpose

Retrieves the interface index associated with a particular interface name. The first of four functions, **if_nametoindex** maps an interface name into its corresponding index.

Library

Library (**libinet.a)**

Syntax

```
#include <net/if.h>
unsigned int
if_nametoindex (ifname)
const char *ifname
```

Description

The first of four functions for Interface Identification. The argument is a null–terminated interface name (for example, `en0`, `tr1`, ...).

Return Values

If successful, **if_nametoindex** returns the interface index associated with the name. If unsuccessful, 0 (zero) is returned.

Related Information

The **if_indextoname** subroutine, **if_nameindex** subroutine, and **if_freenameindex** subroutine.

Subroutines Overview in *AIX Version 4 General Programming Concepts: Writing and Debugging Programs*.

IMAIXMapping Subroutine

Purpose

Translates a pair of *Key* and *State* parameters to a string and returns a pointer to this string.

Library

Input Method Library (**libIM.a**)

Syntax

```
caddr_t IMAIXMapping(IMMap, Key, State, NBytes)
IMMap IMMap;
KeySym Key;
uint State;
int *NBytes;
```

Description

The **IMAIXMapping** subroutine translates a pair of *Key* and *State* parameters to a string and returns a pointer to this string.

This function handles the diacritic character sequence and Alt–NumPad key sequence.

Parameters

IMMap	Identifies the keymap.
Key	Specifies the key symbol to which the string is mapped.
State	Specifies the state to which the string is mapped.
NBytes	Returns the length of the returning string.

Return Values

If the length set by the *NBytes* parameter has a positive value, the **IMAIXMapping** subroutine returns a pointer to the returning string.

Note: The returning string is not null–terminated.

IMAuxCreate Callback Subroutine

Purpose

Tells the application program to create an auxiliary area.

Syntax

```
int IMAuxCreate(IM, AuxiliaryID, UData)
IMObject IM;
caddr_t *AuxiliaryID;
caddr_t UData;
```

Description

The **IMAuxCreate** subroutine is invoked by the input method of the operating system to create an auxiliary area. The auxiliary area can contain several different forms of data and is not restricted by the interface.

Most input methods display one auxiliary area at a time, but callbacks must be capable of handling multiple auxiliary areas.

Parameters

IM	Indicates the input method instance.
AuxiliaryID	Identifies the newly created auxiliary area.
UData	Identifies an argument passed by the **IMCreate** subroutine.

Return Values

On successful return of the **IMAuxCreate** subroutine, a newly created auxiliary area is set to the *AuxiliaryID* value and the **IMError** global variable is returned. Otherwise, the **IMNoError** value is returned.

Implementation Specifics

This subroutine is provided by applications that use input methods.

Related Information

The **IMCreate** subroutine.

Input Method Overview and National Language Support Overview for Programming in *AIX General Programming Concepts : Writing and Debugging Programs*.

Understanding Callbacks in *AIX General Programming Concepts : Writing and Debugging Programs*.

IMAuxDestroy Callback Subroutine

Purpose

Tells the application to destroy the auxiliary area.

Syntax

```
int IMAuxDestroy(IM, AuxiliaryID, UData)
IMObject IM;
caddr_t AuxiliaryID;
caddr_t UData;
```

Description

The **IMAuxDestroy** subroutine is called by the input method of the operating system to tell the application to destroy an auxiliary area.

Parameters

IM	Indicates the input method instance.
AuxiliaryID	Identifies the auxiliary area to be destroyed.
UData	An argument passed by the **IMCreate** subroutine.

Return Values

If an error occurs, the **IMAuxDestroy** subroutine returns the **IMError** global variable. Otherwise, the **IMNoError** value is returned.

Implementation Specifics

This subroutine is provided by applications that use input methods.

Related Information

The **IMCreate** subroutine.

Input Method Overview and National Language Support Overview for Programming in *AIX General Programming Concepts : Writing and Debugging Programs*.

Understanding Callbacks in *AIX General Programming Concepts : Writing and Debugging Programs*.

IMAuxDraw Callback Subroutine

Purpose

Tells the application program to draw the auxiliary area.

Syntax

```
int IMAuxDraw(IM, AuxiliaryID, AuxiliaryInformation, UData)
IMObject IM;
caddr_t AuxiliaryID;
IMAuxInfo *AuxiliaryInformation;
caddr_t UData;
```

Description

The **IMAuxDraw** subroutine is invoked by the input method to draw an auxiliary area. The auxiliary area should have been previously created.

Parameters

IM	Indicates the input method instance.
AuxiliaryID	Identifies the auxiliary area.
AuxiliaryInformation	Points to the **IMAuxInfo** structure.
UData	An argument passed by the **IMCreate** subroutine.

Return Values

If an error occurs, the **IMAuxDraw** subroutine returns the **IMError** global variable. Otherwise, the **IMNoError** value is returned.

Implementation Specifics

This subroutine is provided by applications that use input methods.

Related Information

The **IMAuxCreate** subroutine, **IMCreate** subroutine.

Input Method Overview and National Language Support Overview for Programming in *AIX General Programming Concepts : Writing and Debugging Programs*.

Understanding Callbacks in *AIX General Programming Concepts : Writing and Debugging Programs*.

IMAuxHide Callback Subroutine

Purpose

Tells the application program to hide an auxiliary area.

Syntax

```
int IMAuxHide(IM, AuxiliaryID, UData)

IMObject IM;
caddr_t AuxiliaryID;
caddr_t UData;
```

Description

The **IMAuxHide** subroutine is called by the input method to hide an auxiliary area.

Parameters

IM	Indicates the input method instance.
AuxiliaryID	Identifies the auxiliary area to be hidden.
UData	An argument passed by the **IMCreate** subroutine.

Return Values

If an error occurs, the **IMAuxHide** subroutine returns the **IMError** global variable. Otherwise, the **IMNoError** value is returned.

Implementation Specifics

This subroutine is provided by applications that use input methods.

Related Information

The **IMAuxCreate** subroutine, **IMCreate** subroutine.

Input Method Overview and National Language Support Overview for Programming in *AIX General Programming Concepts : Writing and Debugging Programs*.

Understanding Callbacks in *AIX General Programming Concepts : Writing and Debugging Programs*.

IMBeep Callback Subroutine

Purpose

Tells the application program to emit a beep sound.

Syntax

```
int IMBeep(IM, Percent, UData)
IMObject IM;
int Percent;
caddr_t UData;
```

Description

The **IMBeep** subroutine tells the application program to emit a beep sound.

Parameters

IM	Indicates the input method instance.
Percent	Specifies the beep level. The value range is from –100 to 100, inclusively. A –100 value means no beep.
UData	An argument passed by the **IMCreate** subroutine.

Return Values

If an error occurs, the **IMBeep** subroutine returns the **IMError** global variable. Otherwise, the **IMNoError** value is returned.

Implementation Specifics

This subroutine is provided by applications that use input methods.

Related Information

The **IMCreate** subroutine.

Input Method Overview and National Language Support Overview for Programming in *AIX General Programming Concepts : Writing and Debugging Programs*.

Understanding Callbacks in *AIX General Programming Concepts : Writing and Debugging Programs*.

IMClose Subroutine

Purpose

Closes the input method.

Library

Input Method Library (**libIM.a**)

Syntax

```
void IMClose(IMfep)
IMFep IMfep;
```

Description

The **IMClose** subroutine closes the input method. Before the **IMClose** subroutine is called, all previously created input method instances must be destroyed with the **IMDestroy** subroutine, or memory will not be cleared.

Parameters

IMfep	Specifies the input method.

Implementation Specifics

This subroutine is part of Base Operating System (BOS) Runtime.

Related Information

The **IMDestroy** subroutine.

Input Method Overview and National Language Support Overview for Programming in *AIX General Programming Concepts : Writing and Debugging Programs*.

IMCreate Subroutine

Purpose

Creates one instance of an **IMObject** object for a particular input method.

Library

Input Method Library (**libIM.a**)

Syntax

```
IMObject IMCreate(IMfep, IMCallback, UData)
IMFep IMfep;
IMCallback *IMCallback;
caddr_t UData;
```

Description

The **IMCreate** subroutine creates one instance of a particular input method. Several input method instances can be created under one input method.

Parameters

IMfep	Specifies the input method.
IMCallback	Specifies a pointer to the caller–supplied **IMCallback** structure.
UData	Optionally specifies an application's own information to the callback functions. With this information, the application can avoid external references from the callback functions. The input method does not change this parameter, but merely passes it to the callback functions. The *UData* parameter is usually a pointer to the application data structure, which contains the information about location, font ID, and so forth.

Return Values

The **IMCreate** subroutine returns a pointer to the created input method instance of type **IMObject**. If the subroutine is unsuccessful, a null value is returned and the **imerrno** global variable is set to indicate the error.

Implementation Specifics

This subroutine is part of Base Operating System (BOS) Runtime.

Related Information

The **IMDestroy** subroutine, **IMFilter** subroutine, **IMLookupString** subroutine, **IMProcess** subroutine.

Input Method Overview and National Language Support Overview for Programming in *AIX General Programming Concepts : Writing and Debugging Programs*.

IMDestroy Subroutine

Purpose

Destroys an input method instance.

Library

Input Method Library (**libIM.a**)

Syntax

```
void IMDestroy(IM)
IMObject IM;
```

Description

The **IMDestroy** subroutine destroys an input method instance.

Parameters

IM	Specifies the input method instance to be destroyed.

Implementation Specifics

This subroutine is part of Base Operating System (BOS) Runtime.

Related Information

The **IMClose** subroutine, **IMCreate** subroutine.

Input Method Overview and National Language Support Overview for Programming in *AIX General Programming Concepts : Writing and Debugging Programs*.

IMFilter Subroutine

Purpose

Determines if a keyboard event is used by the input method for internal processing.

Library

Input Method Library (**libIM.a**)

Syntax

```
int IMFilter(Im, Key, State, String, Length)
IMObect Im;
Keysym Key;
uint State, *Length;
caddr_t  *String;
```

Description

The **IMFilter** subroutine is used to process a keyboard event and determine if the input method for this operating system uses this event. The return value indicates:

- The event is filtered (used by the input method) if the return value is **IMInputUsed**. Otherwise, the input method did not accept the event.
- Independent of the return value, a string may be generated by the keyboard event if pre–editing is complete.

Note: The buffer returned from the **IMFilter** subroutine is owned by the input method editor and can not continue between calls.

Parameters

Im	Specifies the input method instance.
Key	Specifies the keysym for the event.
State	Defines the state of the keysym. A value of 0 means that the keysym is not redefined.
String	Holds the returned string if one exists. A null value means that no composed string is ready.
Length	Defines the length of the input string. If the string is not null, returns the length.

Return Values

IMInputUsed	The input method for this operating system filtered the event.
IMInputNotUsed	The input method for this operating system did not use the event.

Implementation Specifics

This subroutine is part of Base Operating System (BOS) Runtime.

Related Information

Input Method Overview in *AIX General Programming Concepts : Writing and Debugging Programs*.

National Language Support Overview for Programming in *AIX General Programming Concepts : Writing and Debugging Programs*.

IMFreeKeymap Subroutine

Purpose

Frees resources allocated by the **IMInitializeKeymap** subroutine.

Library

Input Method Library (**libIM.a**)

Syntax

```
void IMFreeKeymap(IMMap)
IMMap IMMap;
```

Description

The **IMFreeKeymap** subroutine frees resources allocated by the **IMInitializeKeymap** subroutine.

Parameters

IMMap	Identifies the keymap.

Implementation Specifics

This subroutine is part of Base Operating System (BOS) Runtime.

Related Information

The **IMInitializeKeymap** subroutine.

Input Method Overview and National Language Support Overview for Programming in *AIX General Programming Concepts : Writing and Debugging Programs*.

IMIndicatorDraw Callback Subroutine

Purpose

Tells the application program to draw the indicator.

Syntax

```
int IMIndicatorDraw(IM, IndicatorInformation, UData)
IMObject IM;
IMIndicatorInfo *IndicatorInformation;
caddr_t UData;
```

Description

The **IMIndicatorDraw** callback subroutine is called by the input method when the value of the indicator is changed. The application program then draws the indicator.

Parameters

IM	Indicates the input method instance.
IndicatorInformation	Points to the **IMIndicatorInfo** structure that holds the current value of the indicator. The interpretation of this value varies among phonic languages. However, the input method provides a function to interpret this value.
UData	An argument passed by the **IMCreate** subroutine.

Return Values

If an error happens, the **IMIndicatorDraw** subroutine returns the **IMError** global variable. Otherwise, the **IMNoError** value is returned.

Implementation Specifics

This subroutine is provided by applications that use input methods.

Related Information

The **IMCreate** subroutine, **IMIndicatorHide** subroutine.

Input Method Overview and National Language Support Overview for Programming in *AIX General Programming Concepts : Writing and Debugging Programs*.

Understanding Callbacks in *AIX General Programming Concepts : Writing and Debugging Programs*.

IMIndicatorHide Callback Subroutine

Purpose

Tells the application program to hide the indicator.

Syntax

```
int IMIndicatorHide(IM, UData)
IMObject IM;
caddr_t UData;
```

Description

The **IMIndicatorHide** subroutine is called by the input method to tell the application program to hide the indicator.

Parameters

IM	Indicates the input method instance.
UData	Specifies an argument passed by the **IMCreate** subroutine.

Return Values

If an error occurs, the **IMIndicatorHide** subroutine returns the **IMError** global variable. Otherwise, the **IMNoError** value is returned.

Implementation Specifics

This subroutine is provided by applications that use input methods.

Related Information

The **IMCreate** subroutine, **IMIndicatorDraw** subroutine.

Input Method Overview and National Language Support Overview for Programming in *AIX General Programming Concepts : Writing and Debugging Programs*.

Understanding Callbacks in *AIX General Programming Concepts : Writing and Debugging Programs*.

IMInitialize Subroutine

Purpose

Initializes the input method for a particular language.

Library

Input Method Library (**libIM.a**)

Syntax

```
IMFep IMInitialize(Name)
char *Name;
```

Description

The **IMInitialize** subroutine initializes an input method. The **IMCreate**, **IMFilter**, and **IMLookupString** subroutines use the input method to perform input processing of keyboard events in the form of keysym state modifiers. The **IMInitialize** subroutine finds the input method that performs the input processing specified by the *Name* parameter and returns an Input Method Front End Processor (**IMFep**) descriptor.

Before calling any of the key event–handling functions, the application must create an instance of an *IMObject* object using the **IMFep** descriptor. Each input method can produce one or more instances of *IMObject* object with the **IMCreate** subroutine.

When the **IMInitialize** subroutine is called, strings returned from the input method are encoded in the code set of the locale. Each **IMFep** description inherits the code set of the locale when the input method is initialized. The locale setting does not change the code set of the **IMFep** description after it is created.

The **IMInitialize** subroutine calls the **load** subroutine to load a file whose name is in the form *Name*.**im**. The *Name* parameter is passed to the **IMInitialize** subroutine. The loadable input method file is accessed in the directories specified by the **LOCPATH** environment variable. The default location for loadable input–method files is the /**usr**/**lib**/**nls**/**loc** directory. If none of the **LOCPATH** directories contain the input method specified by the *Name* parameter, the default location is searched.

Note: All **setuid** and **setgid** programs will ignore the **LOCPATH** environment variable.

The name of the input method file usually corresponds to the locale name, which is in the form **Language_territory.codesest@***modifier*. In the environment, the modifier is in the form **@im**=*modifier*. The **IMInitialize** subroutine converts the **@im**= substring to @ when searching for loadable input–method files.

Parameters

Name	Specifies the language to be used. Each input method is dynamically linked to the application program.

Return Values

If **IMInitialize** succeeds, it returns an **IMFep** handle. Otherwise, null is returned and the **imerrno** global variable is set to indicate the error.

Implementation Specifics

This subroutine is part of Base Operating System (BOS) Runtime.

Files

/usr/lib/nls/loc Contains loadable input–method files.

Related Information

The **IMCreate** subroutine.

Input Method Overview and National Language Support Overview for Programming in *AIX General Programming Concepts : Writing and Debugging Programs*.

IMInitializeKeymap Subroutine

Purpose

Initializes the keymap associated with a specified language.

Library

Input Method Library (**libIM.a**)

Syntax

```
IMMap IMInitalizeKeymap(Name)
char *Name;
```

Description

The **IMInitializeKeymap** subroutine initializes an input method keymap (imkeymap). The **IMAIXMapping** and **IMSimpleMapping** subroutines use the imkeymap to perform mapping of keysym state modifiers to strings. The **IMInitializeKeymap** subroutine finds the imkeymap that performs the keysym mapping and returns an imkeymap descriptor, **IMMap**. The strings returned by the imkeymap mapping functions are treated as unsigned bytes.

The applications that use input methods usually do not need to manage imkeymaps separately. The imkeymaps are managed internally by input methods.

The **IMInitializeKeymap** subroutine searches for an imkeymap file whose name is in the form *Name*.**im**. The *Name* parameter is passed to the **IMInitializeKeymap** subroutine. The imkeymap file is accessed in the directories specified by the **LOCPATH** environment variable. The default location for input method files is the **/usr/lib/nls/loc** directory. If none of the **LOCPATH** directories contain the keymap method specified by the *Name* parameter, the default location is searched.

Note: All **setuid** and **setgid** programs will ignore the **LOCPATH** environment variable.

The name of the imkeymap file usually corresponds to the locale name, which is in the form **Language_territory.codesest@***modifier*. In the AIXwindows environment, the modifier is in the form **@im**=*modifier*. The **IMInitializeKeymap** subroutine converts the **@im**= s*ubstring* to @ (at sign) when searching for loadable input method files.

Parameters

Name	Specifies the name of the imkeymap.

Return Values

The **IMInitializeKeymap** subroutine returns a descriptor of type **IMMap**. Returning a null value indicates the occurrence of an error. The **IMMap** descriptor is defined in the **im.h** file as the **caddr_t** structure. This descriptor is used for keymap manipulation functions.

Implementation Specifics

This subroutine is part of Base Operating System (BOS) Runtime.

Files

/usr/lib/nls/loc	Contains loadable input–method files.

Related Information

The **IMFreeKeymap**, **IMQueryLanguage** subroutine.

Input Method Overview and National Language Support Overview for Programming in *AIX General Programming Concepts : Writing and Debugging Programs*.

IMIoctl Subroutine

Purpose

Performs a variety of control or query operations on the input method.

Library

Input Method Library (**libIM.a**)

Syntax

```
int IMIoctl(IM, Operation, Argument)
IMObject IM;
int Operation;
char *Argument;
```

Description

The **IMIoctl** subroutine performs a variety of control or query operations on the input method specified by the *IM* parameter. In addition, this subroutine can be used to control the unique function of each language input method because it provides input method–specific extensions. Each input method defines its own function.

Parameters

IM
Specifies the input method instance.

Operation
Specifies the operation.

Argument
The use of this parameter depends on which of the following operations is performed.

IM_Refresh
Refreshes the text area, auxiliary areas, and indicator by calling the needed callback functions if these areas are not empty. The *Argument* parameter is not used.

IM_GetString
Gets the current pre–editing string. The *Argument* parameter specifies the address of the **IMSTR** structure supplied by the caller. The callback function is invoked to clear the pre–editing if it exists.

IM_Clear
Clears the text and auxiliary areas if they exist. If the *Argument* parameter is not a null value, this operation invokes the callback functions to clear the screen. The keyboard state remains the same.

IM_Reset
Clears the auxiliary area if it currently exists. If the *Argument* parameter is a null value, this operation clears only the internal buffer of the input method. Otherwise, the **IMAuxHide** subroutine is called, and the input method returns to its initial state.

IM_ChangeLength
Changes the maximum length of the pre–editing string.

IMNormalMode Specifies the normal mode of pre–editing.

IMSuppressedMode
Suppresses pre–editing.

IM_QueryState Returns the status of the text area, the auxiliary area, and the indicator. It also returns the beep status and the processing mode. The results are stored into the caller–supplied **IMQueryState** structure pointed to by the *Argument* parameter.

IM_QueryText Returns detailed information about the text area. The results are stored in the caller–supplied **IMQueryText** structure pointed to by the *Argument* parameter.

IM_QueryAuxiliary
Returns detailed information about the auxiliary area. The results are stored in the caller–supplied **IMQueryAuxiliary** structure pointed to by the *Argument* parameter.

IM_QueryIndicator
Returns detailed information about the indicator. The results are stored in the caller–supplied **IMQueryIndicator** structure pointed to by the *Argument* parameter.

IM_QueryIndicatorString
Returns an indicator string corresponding to the current indicator. Results are stored in the caller–supplied **IMQueryIndicatorString** structure pointed to by the *Argument* parameter. The caller can request either a short or long form with the format member of the **IMQueryIndicatorString** structure.

IM_SupportSelection
Informs the input method whether or not an application supports an auxiliary area selection list. The application must support selections inside the auxiliary area and determine how selections are displayed. If this operation is not performed, the input method assumes the application does not support an auxiliary area selection list.

Return Values

The **IMIoctl** subroutine returns a value to the **IMError** global variable that indicates the type of error encountered. Some error types are provided in the **/usr/include/imerrno.h** file.

Implementation Specifics

This subroutine is part of Base Operating System (BOS) Runtime.

Related Information

The **IMFilter** subroutine, **IMLookupString** subroutine, **IMProcess** subroutine.

Input Method Overview and National Language Support Overview for Programming in *AIX General Programming Concepts : Writing and Debugging Programs*.

IMLookupString Subroutine

Purpose

Maps a *Key/State* (key symbol/state) pair to a string.

Library

Input Method Library (**libIM.a**)

Syntax

```
int IMLookupString(Im, Key, State, String, Length)
IMObject Im;
KeySym Key;
uint State, *Length;
caddr_t *String;
```

Description

The **IMLookupString** subroutine is used to map a *Key/State* pair to a localized string. It uses an internal input method keymap (**imkeymap**) file to map a keysym/modifier to a string. The string returned is encoded in the same code set as the locale of **IMObject** and IM Front End Processor.

Note: The buffer returned from the **IMLookupString** subroutine is owned by the input method editor and can not continue between calls.

Parameters

Im	Specifies the input method instance.
Key	Specifies the key symbol for the event.
State	Defines the state for the event. A value of 0 means that the key is not redefined.
String	Holds the returned string, if one exists. A null value means that no composed string is ready.
Length	Defines the length string on input. If the string is not null, identifies the length returned.

Return Values

IMError	Error encountered.
IMReturnNothing	No string or keysym was returned.
IMReturnString	String returned.

Implementation Specifics

This subroutine is part of Base Operating System (BOS) Runtime.

Related Information

Input Method Overview in *AIX General Programming Concepts : Writing and Debugging Programs*.

National Language Support Overview for Programming in *AIX General Programming Concepts : Writing and Debugging Programs*.

IMProcess Subroutine

Purpose

Processes keyboard events and language–specific input.

Library

Input Method Library (**libIM.a**)

Note: This subroutine will be removed in future releases. Use the **IMFilter** and **IMLookupString** subroutines to process keyboard events.

Syntax

```
int IMProcess (IM, KeySymbol, State, String, Length)
IMObject IM;
KeySym KeySymbol;
uint State;
caddr_t *String;
uint *Length;
```

Description

This subroutine is a main entry point to the input method of the operating system. The **IMProcess** subroutine processes one keyboard event at a time. Processing proceeds as follows:

- Validates the *IM* parameter.
- Performs keyboard translation for all supported modifier states.
- Invokes internal function to do language–dependent processing.
- Performs any necessary callback functions depending on the internal state.
- Returns to application, setting the *String* and *Length* parameters appropriately.

Parameters

IM	Specifies the input method instance.
KeySymbol	Defines the set of keyboard symbols that will be handled.
State	Specifies the state of the keyboard.
String	Holds the returned string. Returning a null value means that the input is used or discarded by the input method. **Note:** The *String* parameter is not a null–terminated string.
Length	Stores the length, in bytes, of the *String* parameter.

Return Values

This subroutine returns the **IMError** global variable if an error occurs. The **IMerrno** global variable is set to indicate the error. Some of the variable values include:

IMError	Error occurred during this subroutine.
IMTextAndAuxiliaryOff	No text string in the Text area, and the Auxiliary area is not shown.
IMTextOn	Text string in the Text area, but no Auxiliary area.
IMAuxiliaryOn	No text string in the Text area, and the Auxiliary area is shown.
IMTextAndAuxiliaryOn	Text string in the Text area, and the Auxiliary is shown.

Implementation Specifics

This subroutine is part of Base Operating System (BOS) Runtime.

Related Information

The **IMClose** subroutine, **IMCreate** subroutine **IMFilter** subroutine, **IMLookupString** subroutine.

Input Method Overview and National Language Support Overview for Programming in *AIX General Programming Concepts : Writing and Debugging Programs*.

IMProcessAuxiliary Subroutine

Purpose

Notifies the input method of input for an auxiliary area.

Library

Input Method Library (**libIM.a**)

Syntax

```
int IMProcessAuxiliary(IM, AuxiliaryID, Button, PanelRow
      PanelColumn, ItemRow, ItemColumn, String, Length)

IMObject IM;
caddr_t AuxiliaryID;
uint Button;
uint PanelRow;
uint PanelColumn;
uint ItemRow;
uint ItemColumn;
caddr_t *String;
uint *Length;
```

Description

The **IMProcessAuxiliary** subroutine notifies the input method instance of input for an auxiliary area.

Parameters

IM — Specifies the input method instance.

AuxiliaryID — Identifies the auxiliary area.

Button — Specifies one of the following types of input:

- **IM_ABORT** — Abort button is pushed.
- **IM_CANCEL** — Cancel button is pushed.
- **IM_ENTER** — Enter button is pushed.
- **IM_HELP** — Help button is pushed.
- **IM_IGNORE** — Ignore button is pushed.
- **IM_NO** — No button is pushed.
- **IM_OK** — OK button is pushed.
- **IM_RETRY** — Retry button is pushed.
- **IM_SELECTED** — Selection has been made. Only in this case do the *PanelRow*, *PanelColumn*, *ItemRow*, and *ItemColumn* parameters have meaningful values.
- **IM_YES** — Yes button is pushed.

PanelRow — Indicates the panel on which the selection event occurred.

PanelColumn — Indicates the panel on which the selection event occurred.

ItemRow — Indicates the selected item.

ItemColumn — Indicates the selected item.

String	Holds the returned string. If a null value is returned, the input is used or discarded by the input method. Note that the *String* parameter is not a null–terminated string.
Length	Stores the length, in bytes, of the *String* parameter.

Implementation Specifics

This subroutine is part of Base Operating System (BOS) Runtime.

Related Information

The **IMAuxCreate** subroutine.

Input Method Overview and National Language Support Overview for Programming in *AIX General Programming Concepts : Writing and Debugging Programs*.

IMQueryLanguage Subroutine

Purpose

Checks to see if the specified input method is supported.

Library

Input Method Library (**libIM.a**)

Syntax

```
uint IMQueryLanguage(Name)
IMLanguage Name;
```

Description

The **IMQueryLanguage** subroutine checks to see if the input method specified by the *Name* parameter is supported.

Parameters

Name	Specifies the input method.

Return Values

The **IMQueryLanguage** subroutine returns a true value if the specified input method is supported, a false value if not.

Implementation Specifics

This subroutine is part of Base Operating System (BOS) Runtime.

Related Information

The **IMClose** subroutine, **IMInitialize** subroutine.

Input Method Overview, National Language Support Overview for Programming, Understanding Keyboard Mapping contains a list of supported languages in *AIX General Programming Concepts : Writing and Debugging Programs*.

IMSimpleMapping Subroutine

Purpose

Translates a pair of *KeySymbol* and *State* parameters to a string and returns a pointer to this string.

Library

Input Method Library (**libIM.a**)

Syntax

```
caddr_t IMSimpleMapping (IMMap, KeySymbol, State, NBytes)
IMMap IMMap;
KeySym KeySymbol;
uint State;
int *NBytes;
```

Description

Like the **IMAIXMapping** subroutine, the **IMSimpleMapping** subroutine translates a pair of *KeySymbol* and *State* parameters to a string and returns a pointer to this string. The parameters have the same meaning as those in the **IMAIXMapping** subroutine.

The **IMSimpleMapping** subroutine differs from the **IMAIXMapping** subroutine in that it does not support the diacritic character sequence or the Alt–NumPad key sequence.

Parameters

IMMap	Identifies the keymap.
KeySymbol	Key symbol to which the string is mapped.
State	Specifies the state to which the string is mapped.
NBytes	Returns the length of the returning string.

Implementation Specifics

This subroutine is part of Base Operating System (BOS) Runtime.

Related Information

The **IMAIXMapping** subroutine, **IMFreeKeymap** subroutine, **IMInitializeKeymap** subroutine.

Input Method Overview and National Language Support Overview for Programming in *AIX General Programming Concepts : Writing and Debugging Programs*.

IMTextCursor Callback Subroutine

Purpose

Asks the application to move the text cursor.

Syntax

```
int IMTextCursor(IM, Direction, Cursor, UData)
IMObject IM;
uint Direction;
int *Cursor;
caddr_t UData;
```

Description

The **IMTextCursor** subroutine is called by the Input Method when the Cursor Up or Cursor Down key is input to the **IMFilter** and **IMLookupString** subroutines.

This subroutine sets the new display cursor position in the text area to the integer pointed to by the *Cursor* parameter. The cursor position is relative to the top of the text area. A value of –1 indicates the cursor should not be moved.

Because the input method does not know the actual length of the screen it always treats a text string as one–dimensional (a single line). However, in the terminal emulator, the text string sometimes wraps to the next line. The **IMTextCursor** subroutine performs this conversion from single–line to multiline text strings. When you move the cursor up or down, the subroutine interprets the cursor position on the text string relative to the input method.

Parameters

IM	Indicates the Input Method instance.
Direction	Specifies up or down.
Cursor	Specifies the new cursor position or –1.
UData	Specifies an argument passed by the **IMCreate** subroutine.

Return Values

If an error occurs, the **IMTextCursor** subroutine returns the **IMError** global variable. Otherwise, the **IMNoError** value is returned.

Implementation Specifics

This subroutine is provided by applications that use input methods.

Related Information

The **IMCreate** subroutine, **IMFilter** subroutine, **IMLookupString** subroutine, **IMTextDraw** subroutine.

Input Method Overview and National Language Support Overview for Programming in *AIX General Programming Concepts : Writing and Debugging Programs*.

Understanding Callbacks in *AIX General Programming Concepts : Writing and Debugging Programs*.

IMTextDraw Callback Subroutine

Purpose

Tells the application program to draw the text string.

Syntax

int IMTextDraw(*IM*, *TextInfo*, *UData***)**
IMObject *IM*;
**IMTextInfo ****TextInfo*;
caddr_t *UData*;

Description

The **IMTextDraw** subroutine is invoked by the Input Method whenever it needs to update the screen with its internal string. This subroutine tells the application program to draw the text string.

Parameters

IM	Indicates the input method instance.
TextInfo	Points to the **IMTextInfo** structure.
UData	An argument passed by the **IMCreate** subroutine.

Return Values

If an error occurs, the **IMTextDraw** subroutine returns the **IMError** global variable. Otherwise, the **IMNoError** value is returned.

Implementation Specifics

This subroutine is provided by applications that use input methods.

Related Information

The **IMCreate** subroutine.

Input Method Overview and National Language Support Overview for Programming in *AIX General Programming Concepts : Writing and Debugging Programs*.

Understanding Callbacks in *AIX General Programming Concepts : Writing and Debugging Programs*.

IMTextHide Callback Subroutine

Purpose

Tells the application program to hide the text area.

Syntax

```
int IMTextHide(IM, UData)
IMObject IM;
caddr_t UData;
```

Description

The **IMTextHide** subroutine is called by the input method when the text area should be cleared. This subroutine tells the application program to hide the text area.

Parameters

IM	Indicates the input method instance.
UData	Specifies an argument passed by the **IMCreate** subroutine.

Return Values

If an error occurs, the **IMTextHide** subroutine returns an **IMError** value. Otherwise, an **IMNoError** value is returned.

Implementation Specifics

This subroutine is provided by applications that use input methods.

Related Information

The **IMTextDraw** subroutine.

Input Method Overview and National Language Support Overview for Programming in *AIX General Programming Concepts : Writing and Debugging Programs*.

Understanding Callbacks in *AIX General Programming Concepts : Writing and Debugging Programs*.

IMTextStart Callback Subroutine

Purpose

Notifies the application program of the length of the pre–editing space.

Syntax

```
int IMTextStart(IM, Space, UData)
IMObject IM;
int *Space;
caddr_t UData;
```

Description

The **IMTextStart** subroutine is called by the input method when the pre–editing is started, but prior to calling the **IMTextDraw** callback subroutine. This subroutine notifies the input method of the length, in terms of bytes, of pre–editing space. It sets the length of the available space (`>=0`) on the display to the integer pointed to by the *Space* parameter. A value of –1 indicates that the pre–editing space is dynamic and has no limit.

Parameters

IM	Indicates the input method instance.
Space	Maximum length of pre–editing string.
UData	An argument passed by the **IMCreate** subroutine.

Implementation Specifics

This subroutine is provided by applications that use input methods.

Related Information

The **IMCreate** subroutine, **IMTextDraw** subroutine.

Input Method Overview and National Language Support Overview for Programming in *AIX General Programming Concepts : Writing and Debugging Programs*.

Understanding Callbacks in *AIX General Programming Concepts : Writing and Debugging Programs*.

inet_net_ntop Subroutine

Purpose

Converts between binary and text address formats.

Library

Library (**libc.a**)

Syntax

```
char *inet_net_ntop
int af;
const void *src;
int bits;
char *dst;
size_t 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 argument, *af*, specifies the family of the address. The argument, *src*, points to a buffer holding an IPv4 address if the *af* argument is `AF_INET`. The argument, *dst*, points to a buffer where the function stores the resulting text string.

Return Values

If successful, a pointer to a buffer containing the text string is returned. If unsuccessful, NULL is returned. Upon failure, **errno** is set to EAFNOSUPPORT if the *af* argument is invalid or ENOSPC if the size of the result buffer is inadquate.

Related Information

The **inet_net_pton** subroutine, **inet_ntop** subroutine, **inet_pton** subroutine.

Subroutines Overview in *AIX Version 4 General Programming Concepts: Writing and Debugging Programs*.

inet_net_pton Subroutine

Purpose

Converts between text and binary address formats.

Library

Library (**libc.a**)

Syntax

```
char *inet_net_ntop
int af;
const char *src;
void *dst;
size_t size;
```

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 argument, af, specifies the family of the address. The argument, src, points the string being passed in. The argument, dst, points to a buffer where the function will store the resulting numeric address.

Return Values

If successful, 1 (one) is returned. If unseccessful, 0 (zero) is returned if the input is not a valid IPv4 string; or a –1 (negative one) with errno set to EAFNOSUPPORT if the af argument is unknown.

Related Information

The **inet_net_ntop** subroutine, **inet_ntop** subroutine, **inet_pton** subroutine.

Subroutines Overview in *AIX Version 4 General Programming Concepts: Writing and Debugging Programs*.

inet_ntop Subroutine

Purpose

Converts between binary and text address formats.

Library

Library (**libc.a**)

Syntax

```
char *inet_ntop
int af;
const void *src;
char *dst;
size_t size;
```

Description

This function converts from an address in binary format (as specified by *src*) to standard text format, and places the result in *dst* (if size, which specifies the space available in *dst*, is sufficient). The argument *af* specifies the family of the address. This can be AF_INET or AF_INET6.

The argument, *src*, points to a buffer holding an IPv4 address if the *af* argument is AF_INET, or an IPv6 address if the *af* argument is AF_INET6. The argument *dst* points to a buffer where the function will store the resulting text string. The size argument specifies the size of this buffer. The application must specify a non–NULL dst argument. For IPv6 addresses, the buffer must be at least 46–octets. For IPv4 addresses, the buffer must be at least 16–octets.

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 <netinet/in.h>:

```
#define INET_ADDRSTRLEN 16
#define INET6_ADDRSTRLEN 46
```

Return Values

If successful, a pointer to the buffer containing the converted address is returned. If unsuccessful, NULL is returned. Upon failure, errno is set to EAFNOSUPPORT if the specified address family (af) is unsupported, or to ENOSPC if the size indicates the destination buffer is too small.

Related Information

The **inet_net_ntop** subroutine, **inet_net_pton** subroutine, and **inet_pton** subroutine.

Subroutines Overview in *AIX Version 4 General Programming Concepts: Writing and Debugging Programs*.

inet_pton Subroutine

Purpose

Converts between text and binary address formats.

Library

Library (**libc.a**)

Syntax

```
char *inet_net_ntop
int af;
const char *src;
void *dst;
```

Description

This function converts an address in its standard text format into its numeric binary form. The argument *af* specifies the family of the address. Note, AF_INET and AF_INET6 address families are currently supported.

The argument *src* points to the string being passed in. The argument *dst* points to a buffer into which the function stores the numeric address. The address is returned in network byte order.

Return Values

If successful, 1 (one) is returned. If unseccessful, 0 (zero) is returned if the input is not a valid IPv4 dotted–decimal string or a valid IPv6 address string; or a –1 (negative one) with errno set to EAFNOSUPPORT if the *af* argument is unknown. The calling application must ensure that the buffer referred to by *dst* is large enough to hold the numeric address (4 bytes for AF_INET or 16 bytes for AF_INET6). If the *af* argument 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 that many implementations of the existing inet_addr and inet_aton functions accept nonstandard input: octal numbers, hexadecimal numbers, and fewer than four numbers. **inet_pton** does not accept these formats.

If the *af* argument is AF_INET6, then the function acdepts a string in one of the standard IPv6 text forms defined the addressing architecture specification..

Related Information

The **inet_net_ntop** subroutine, **inet_net_pton** subroutine, and **inet_ntop** subroutine.

Subroutines Overview in *AIX Version 4 General Programming Concepts: Writing and Debugging Programs*.

initgroups Subroutine

Purpose

Initializes supplementary group ID.

Library

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

Syntax

```
int initgroups (User, BaseGID)
char *User;
int BaseGID;
```

Description

Attention: The **initgroups** subroutine uses the **getgrent** and **getpwent** family of subroutines. If the program that invokes the **initgroups** subroutine uses any of these subroutines, calling the **initgroups** subroutine overwrites the static storage areas used by these subroutines.

The **initgroups** subroutine reads the defined group membership of the specified *User* parameter and sets the supplementary group ID of the current process to that value. The *BaseGID* parameter is always included in the supplementary group ID. The supplementary group is normally the principal user's group. If the user is in more than **NGROUPS_MAX** groups, set in the **limits.h** file, only **NGROUPS_MAX** groups are set, including the *BaseGID* group.

Parameters

User	Identifies a user.
BaseGID	Specifies an additional group to include in the group set.

Return Values

0	Indicates that the subroutine was success.
–1	Indicates that the subroutine failed. The **errno** global variable is set to indicate the error.

Implementation Specifics

This subroutine is part of Base Operating System (BOS) Runtime.

Related Information

The **getgid** subroutine, **getgrent**, **getgrgid**, **getgrnam**, **putgrent**, **setgrent**, or **endgrent** subroutine, **getgroups** subroutine, **setgroups** subroutine.

The **groups** command, **setgroups** command.

List of Security and Auditing Subroutines, Subroutines Overview in *AIX General Programming Concepts : Writing and Debugging Programs*.

initialize Subroutine

Purpose

Performs printer initialization.

Library

None (provided by the formatter).

Syntax

```
#include <piostruct.h>

int initialize ()
```

Description

The **initialize** subroutine is invoked by the formatter driver after the **setup** subroutine returns.

If the **–j** flag passed from the **qprt** command has a nonzero value (true), the **initialize** subroutine uses the **piocmdout** subroutine to send a command string to the printer. This action initializes the printer to the proper state for printing the file. Any variables referenced by the command string should be the attribute values from the database, overridden by values from the command line.

If the **–j** flag passed from the **qprt** command has a nonzero value (true), any necessary fonts should be downloaded.

Return Values

0	Indicates a successful operation.

If the **initialize** subroutine detects an error, it uses the **piomsgout** subroutine to invoke an error message. It then invokes the **pioexit** subroutine with a value of **PIOEXITBAD**.

Note: If either the **piocmdout** or **piogetstr** subroutine detects an error, it issues its own error messages and terminates the print job.

Related Information

The **piocmdout** subroutine, **pioexit** subroutine, **piogetstr** subroutine, **piomsgout** subroutine, **setup** subroutine.

Adding a New Printer Type to Your System, Printer Addition Management Subsystem: Programming Overview, Understanding Embedded References in Printer Attribute Strings in *AIX Kernel Extensions and Device Support Programming Concepts*.

Example of Print Formatter in *AIX General Programming Concepts : Writing and Debugging Programs*.

insque or remque Subroutine

Purpose

Inserts or removes an element in a queue.

Library

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

Syntax

```
#include <search.h>


insque (Element, Pred)
void *Element, *Pred;

remque (Element)
void *Element;
```

Description

The **insque** and **remque** subroutines manipulate queues built from double–linked lists. Each element in the queue must be in the form of a **qelem** structure. The **next** and **prev** elements of that structure must point to the elements in the queue immediately before and after the element to be inserted or deleted.

The **insque** subroutine inserts the element pointed to by the *Element* parameter into a queue immediately after the element pointed to by the *Pred* parameter.

The **remque** subroutine removes the element defined by the *Element* parameter from a queue.

Parameters

Pred	Points to the element in the queue immediately before the element to be inserted or deleted.
Element	Points to the element in the queue immediately after the element to be inserted or deleted.

Implementation Specifics

These subroutines are part of Base Operating System (BOS) Runtime.

Related Information

Searching and Sorting Example Program in *AIX General Programming Concepts : Writing and Debugging Programs*.

Subroutines Overview in *AIX General Programming Concepts : Writing and Debugging Programs*.

ioctl, ioctlx, ioctl32, or ioctl32x Subroutine

Purpose

Performs control functions associated with open file descriptors.

Library

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

BSD Library (**libbsd.a**)

Syntax

#include <sys/ioctl.h>
#include <sys/types.h>
#include <unistd.h>

int ioctl (*FileDescriptor***,** *Command***,** *Argument***)**
int *FileDescriptor***,** *Command***;**
void **Argument***;**

int ioctlx (*FileDescriptor***,** *Command***,** *Argument***,** *Ext***)**
int *FileDescriptor***,** *Command***;**
void **Argument***;**
int *Ext***;**

int ioct132 (*FileDescriptor***,** *Command***,** *Argument***)**
int *FileDescriptor***,** *Command***;**
unsigned int *Argument***;**

int ioct132x (*FileDescriptor***,** *Command***,** *Argument***,** *Ext***)**
int *FileDescriptor***,** *Command***;**
unsigned int *Argument***;**
unsigned int *Ext***;**

Description

The **ioctl** subroutine performs a variety of control operations on the object associated with the specified open file descriptor. This function is typically used with character or block special files, sockets, or generic device support such as the **termio** general terminal interface.

The control operation provided by this function call is specific to the object being addressed, as are the data type and contents of the *Argument* parameter. The **ioctlx** form of this function can be used to pass an additional extension parameter to objects supporting it.

The control operation provided by this function call is specific to the object being addressed, as are the data type and contents of the *Argument* parameter. The **ioctlx** form of this function can be used to pass an additional extension parameter to objects supporting it. The **ioct132** and **ioct132x** forms of this function behave in the same way as **ioctl** and **ioctlx**, but allow 64–bit applications to call the **ioctl** routine for an object that does not normally work with 64–bit applications.

Performing an ioctl function on a file descriptor associated with an ordinary file results in an error being returned.

Parameters

FileDescriptor Specifies the open file descriptor for which the control operation is to be performed.

Command Specifies the control function to be performed. The value of this parameter depends on which object is specified by the *FileDescriptor* parameter.

Argument Specifies additional information required by the function requested in the *Command* parameter. The data type of this parameter (a **void** pointer) is object–specific, and is typically used to point to an object device–specific data structure. However, in some device–specific instances, this parameter is used as an integer.

Ext Specifies an extension parameter used with the **ioctlx** subroutine. This parameter is passed on to the object associated with the specified open file descriptor. Although normally of type **int**, this parameter can be used as a pointer to a device–specific structure for some devices.

File Input/Output (FIO) ioctl Command Values

A number of file input/output (FIO) ioctl commands are available to enable the **ioctl** subroutine to function similar to the **fcntl** subroutine:

FIOCLEX and FIONCLEX Manipulate the **close–on–exec** flag to determine if a file descriptor should be closed as part of the normal processing of the **exec** subroutine. If the flag is set, the file descriptor is closed. If the flag is clear, the file descriptor is left open.

The following code sample illustrates the use of the **fcntl** subroutine to set and clear the **close–on–exec** flag:

```
/* set the close-on-exec flag for fd1 */
fcntl(fd1,F_SETFD,FD_CLOEXEC);
/* clear the close-on-exec flag for fd2 */
fcntl(fd2,F_SETFD,0);
```

Although the **fcntl** subroutine is normally used to set the **close–on–exec** flag, the **ioctl** subroutine may be used if the application program is linked with the Berkeley Compatibility Library (**libbsd.a**) or the Berkeley Thread Safe Library (**libbsd_r.a**) (4.2.1 and later versions). The following ioctl code fragment is equivalent to the preceding **fcntl** fragment:

```
/* set the close-on-exec flag for fd1 */
ioctl(fd1,FIOCLEX,0);
/* clear the close-on-exec flag for fd2 */
ioctl(fd2,FIONCLEX,0);
```

The third parameter to the **ioctl** subroutine is not used for the **FIOCLEX** and **FIONCLEX** ioctl commands.

FIONBIO Enables nonblocking I/O. The effect is similar to setting the **O_NONBLOCK** flag with the **fcntl** subroutine. The third parameter to the **ioctl** subroutine for this command is a pointer to an integer that indicates whether nonblocking I/O is being enabled or disabled. A value of 0 disables non–blocking I/O. Any nonzero value enables nonblocking I/O. A sample code fragment follows:

```
int flag;
/* enable NBIO for fd1 */
flag = 1;
ioctl(fd1,FIONBIO,&flag);
/* disable NBIO for fd2 */
flag = 0;
ioctl(fd2,FIONBIO,&flag);
```

FIONREAD Determines the number of bytes that are immediately available to be read on a file descriptor. The third parameter to the **ioctl** subroutine for this command is a pointer to an integer variable where the byte count is to be returned. The following sample code illustrates the proper use of the **FIONREAD** ioctl command:

```
int nbytes;

ioctl(fd,FIONREAD,&nbytes);
```

FIOASYNC Enables a simple form of asynchronous I/O notification. This command causes the kernel to send **SIGIO** signal to a process or a process group when I/O is possible. Only sockets, ttys, and pseudo–ttys implement this functionality.

The third parameter of the **ioctl** subroutine for this command is a pointer to an integer variable that indicates whether the asynchronous I/O notification should be enabled or disabled. A value of 0 disables I/O notification; any nonzero value enables I/O notification. A sample code segment follows:

```
int flag;
/* enable ASYNC on fd1 */
flag = 1;
ioctl(fd, FIOASYNC,&flag);
/* disable ASYNC on fd2 */
flag = 0;
ioctl(fd,FIOASYNC,&flag);
```

FIOSETOWN Sets the recipient of the **SIGIO** signals when asynchronous I/O notification (**FIOASYNC**) is enabled. The third parameter to the **ioctl** subroutine for this command is a pointer to an integer that contains the recipient identifier. If the value of the integer pointed to by the third parameter is negative, the value is assumed to be a process group identifier. If the value is positive, it is assumed to be a process identifier.

Sockets support both process groups and individual process recipients, while ttys and psuedo–ttys support only process groups. Attempts to specify an individual process as the recipient will be converted to the process group to which the process belongs. The following code example illustrates how to set the recipient identifier:

```
int owner;
owner = -getpgrp();
ioctl(fd,FIOSETOWN,&owner);
```

Note: In this example, the asynchronous I/O signals are being enabled on a process group basis. Therefore, the value passed through the owner parameter must be a negative number.

The following code sample illustrates enabling asynchronous I/O signals to an individual process:

```
int owner;
owner = getpid();
ioctl(fd,FIOSETOWN,&owner);
```

FIOGETOWN Determines the current recipient of the asynchronous I/O signals of an object that has asynchronous I/O notification (**FIOASYNC**) enabled. The third parameter to the **ioctl** subroutine for this command is a pointer to an integer used to return the owner ID. For example:

```
int owner;
ioctl(fd,FIOGETOWN,&owner);
```

If the owner of the asynchronous I/O capability is a process group, the value returned in the reference parameter is negative. If the owner is an individual process, the value is positive.

Return Values

If the **ioctl** subroutine fails, a value of –1 is returned. The **errno** global variable is set to indicate the error.

The **ioctl** subroutine fails if one or more of the following are true:

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

EFAULT The *Argument* or *Ext* parameter is used to point to data outside of the process address space.

EINTR A signal was caught during the **ioctl** or **ioctlx** subroutine and the process had not enabled re–startable subroutines for the signal.

EINTR A signal was caught during the **ioctl** , **ioctlx** , **ioctl32** , or **ioct132x** subroutine and the process had not enabled re–startable subroutines for the signal.

EINVAL The *Command* or *Argument* parameter is not valid for the specified object.

ENOTTY The *FileDescriptor* parameter is not associated with an object that accepts control functions.

ENODEV The *FileDescriptor* parameter is associated with a valid character or block special file, but the supporting device driver does not support the ioctl function.

ENXIO The *FileDescriptor* parameter is associated with a valid character or block special file, but the supporting device driver is not in the configured state.

Object–specific error codes are defined in the documentation for associated objects.

Implementation Specifics

This subroutine is part of Base Operating System (BOS) Runtime.

Related Information

The **ddioctl** device driver entry point and the **fp_ioctl** kernel service in *AIX Technical Reference: Kernel and Subsystems*.

The Special Files Overview in *AIX Files Reference*.

The Input and Output Handling Programmer's Overview, the tty Subsystem Overview, in *AIX General Programming Concepts : Writing and Debugging Programs*.

The Sockets Overview and Understanding Socket Data Transfer in *AIX Communications Programming Concepts*.

isendwin Subroutine

Purpose

Determines whether the **endwin** subroutine was called without any subsequent refresh calls.

Library

Curses Library (**libcurses.a**)

Syntax

```
#include <curses.h>

isendwin()
```

Description

The **isendwin** subroutine determines whether the **endwin** subroutine was called without any subsequent refresh calls. If the **endwin** was called without any subsequent calls to the **wrefresh** or **doupdate** subroutines, the **isendwin** subroutine returns TRUE.

Return Values

TRUE	Indicates the **endwin** subroutine was called without any subsequent calls to the **wrefresh** or **doupdate** subroutines.
FALSE	Indicates subsequest calls to the refresh subroutines.

Implementation Specifics

This subroutine is part of Base Operating System (BOS) Runtime.

Related Information

The **doupdate** subroutine, **endwin** subroutine, **wrefresh** subroutine.

Curses Overview for Programming, Initializing Curses, List of Curses Subroutines in *AIX General Programming Concepts : Writing and Debugging Programs*.

iswalnum, iswalpha, iswcntrl, iswdigit, iswgraph, iswlower, iswprint, iswpunct, iswspace, iswupper, or iswxdigit Subroutine

Purpose

Tests a wide character for membership in a specific character class.

Library

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

Syntax

```
#include <wchar.h>

int iswalnum (WC)
wint_t WC;

int iswalpha (WC)
wint_t WC;

int iswcntrl (WC)
wint_t WC;

int iswdigit (WC)
wint_t WC;

int iswgraph (WC)
wint_t WC;

int iswlower (WC)
wint_t WC;

int iswprint (WC)
wint_t WC;

int iswpunct (WC)
wint_t WC;

int iswspace (WC)
wint_t WC;

int iswupper (WC)
wint_t WC;

int iswxdigit (WC)
wint_t WC;
```

Description

The **isw** subroutines check the character class status of the wide character code specified by the *WC* parameter. Each subroutine tests to see if a wide character is part of a different character class. If the wide character is part of the character class, the **isw** subroutine returns true; otherwise, it returns false.

Each subroutine is named by adding the **isw** prefix to the name of the character class that the subroutine tests. For example, the **iswalpha** subroutine tests whether the wide character specified by the *WC* parameter is an alphabetic character. The character classes are defined as follows:

alnum Alphanumeric character.

alpha Alphabetic character.

cntrl Control character. No characters in the **alpha** or **print** classes are included.

digit	Numeric digit character.
graph	Graphic character for printing, not including the space character or **cntrl** characters. Includes all characters in the **digit** and **punct** classes.
lower	Lowercase character. No characters in **cntrl**, **digit**, **punct**, or **space** are included.
print	Print character. All characters in the **graph** class are included, but no characters in **cntrl** are included.
punct	Punctuation character. No characters in the **alpha**, **digit**, or **cntrl** classes, or the space character are included.
space	Space characters.
upper	Uppercase character.
xdigit	Hexadecimal character.

Parameters

WC	Specifies a wide character for testing.

Return Values

If the wide character tested is part of the particular character class, the **isw** subroutine returns a nonzero value; otherwise it returns a value of 0.

Implementation Specifics

These subroutines are part of Base Operating System (BOS) Runtime.

Related Information

The **iswctype** subroutine, **setlocale** subroutine, **towlower** subroutine, **towupper** subroutine **wctype** subroutine.

National Language Support Overview for Programming, Subroutines Overview, Understanding Wide Character Classification Subroutines in *AIX General Programming Concepts : Writing and Debugging Programs*.

iswctype or is_wctype Subroutine

Purpose

Determines properties of a wide character.

Library

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

Syntax

```
#include  <wchar.h>

int iswctype  (WC, Property)
wint_t WC;
wctype_t Property;

int is_wctype (WC, Property)
wint_t WC;
wctype_t Property;
```

Description

The **iswctype** subroutine tests the wide character specified by the *WC* parameter to determine if it has the property specified by the *Property* parameter. The **iswctype** subroutine is defined for the wide–character null value and for values in the character range of the current code set, defined in the current locale. The **is_wctype** subroutine is identical to the **iswctype** subroutine.

Parameters

WC	Specifies the wide character to be tested.
Property	Specifies the property for which to test.

Return Values

If the *WC* parameter has the property specified by the *Property* parameter, the **iswctype** subroutine returns a nonzero value. If the value specified by the *WC* parameter does not have the property specified by the *Property* parameter, the **iswctype** subroutine returns a value of zero. If the value specified by the *WC* parameter is not in the subroutine's domain, the result is undefined. If the value specified by the *Property* parameter is not valid (that is, not obtained by a call to the **wctype** subroutine, or the *Property* parameter has been invalidated by a subsequent call to the **setlocale** subroutine that has affected the **LC_CTYPE** category), the result is undefined.

Implementation Specifics

These subroutines are part of Base Operating System (BOS) Runtime.

The **iswctype** subroutine adheres to *Systems Interface and Headers, Issue 4* of X/Open.

Related Information

The **iswalnum** subroutine, **iswalpha** subroutine, **iswcntrl** subroutine, **iswdigit** subroutine, **iswgraph** subroutine, **iswlower** subroutine, **iswprint** subroutine, **iswpunct** subroutine, **iswspace** subroutine, **iswupper** subroutine, **iswxdigit** subroutine, **setlocale** subroutine, **towlower** subroutine, **towupper** subroutine, **wctype** subroutine.

National Language Support Overview for Programming, Subroutines Overview, Understanding Wide Character Classification Subroutines in *AIX General Programming Concepts : Writing and Debugging Programs*.

jcode Subroutines

Purpose

Perform string conversion on 8–bit processing codes.

Library

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

Syntax

```
#include <jcode.h>

char *jistosj(String1, String2)
char *String1, *String2;

char *jistouj(String1, String2)
char *String1, *String2;

char *sjtojis(String1, String2)
char *String1, *String2;

char *sjtouj(String1, String2)
char *String1, *String2;

char *ujtojis(String1, String2)
char *String1, *String2;

char *ujtosj(String1, String2)
char *String1, *String2;

char *cjistosj(String1, String2)
char *String1, *String2;

char *cjistouj(String1, String2)
char *String1, *String2;

char *csjtojis(String1, String2)
char *String1, *String2;

char *csjtouj(String1, String2)
char *String1, *String2;

char *cujtojis(String1, String2)
char *String1, *String2;

char *cujtosj(String1, String2)
char *String1, *String2;
```

Description

The **jistosj**, **jistouj**, **sjtojis**, **sjtouj**, **ujtojis**, and **ujtosj** subroutines perform string conversion on 8–bit processing codes. The *String2* parameter is converted and the converted string is stored in the *String1* parameter. The overflow of the *String1* parameter is not checked. Also, the *String2* parameter must be a valid string. Code validation is not permitted.

The **jistosj** subroutine converts JIS to SJIS. The **jistouj** subroutine converts JIS to UJIS. The **sjtojis** subroutine converts SJIS to JIS. The **sjtouj** subroutine converts SJIS to UJIS. The **ujtojis** subroutine converts UJIS to JIS. The **ujtosj** subroutine converts UJIS to SJIS.

The **cjistosj**, **cjistouj**, **csjtojis**, **csjtouj**, **cujtojis**, and **cujtosj** macros perform code conversion on 8–bit processing JIS Kanji characters. A character is removed from the *String2* parameter, and its code is converted and stored in the *String1* parameter. The *String1* parameter is returned. The validity of the *String2* parameter is not checked.

The **cjistosj** macro converts from JIS to SJIS. The **cjistouj** macro converts from JIS to UJIS. The **csjtojis** macro converts from SJIS to JIS. The **csjtouj** macro converts from SJIS to UJIS. The **cujtojis** macro converts from UJIS to JIS. The **cujtosj** macro converts from UJIS to SJIS.

Parameters

String1	Stores converted string or code.
String2	Stores string or code to be converted.

Implementation Specifics

These subroutines are part of Base Operating System (BOS) Runtime.

Related Information

The Japanese **conv** subroutines, Japanese **ctype** subroutines.

List of String Manipulation Services in *AIX General Programming Concepts : Writing and Debugging Programs*.

National Language Support Overview for Programming in *AIX General Programming Concepts : Writing and Debugging Programs*.

Subroutines Overview in *AIX General Programming Concepts : Writing and Debugging Programs*.

Japanese conv Subroutines

Purpose

Translates predefined Japanese character classes.

Library

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

Syntax

```
#include <ctype.h>
int atojis (Character)
int Character;

int jistoa (Character)
int Character;

int _atojis (Character)
int Character;

int _jistoa (Character)
int Character;

int tojupper (Character)
int Character;

int tojlower (Character)
int Character;

int _tojupper (Character)
int Character;

int _tojlower (Character)
int Character;

int toujis (Character)
int Character;

int kutentojis (Character)
int Character;

int tojhira (Character)
int Character;

int tojkata (Character)
int Character;
```

Description

When running AIX with Japanese Language Support on your system, the legal value of the *Character* parameter is in the range from 0 to **NLCOLMAX**.

The **jistoa** subroutine converts an SJIS ASCII equivalent to the corresponding ASCII equivalent. The **atojis** subroutine converts an ASCII character to the corresponding SJIS equivalent. Other values are returned unchanged.

The **_jistoa** and **_atojis** routines are macros that function like the **jistoa** and **atojis** subroutines, but are faster and have no error checking function.

The **tojlower** subroutine converts a SJIS uppercase letter to the corresponding SJIS lowercase letter. The **tojupper** subroutine converts an SJIS lowercase letter to the corresponding SJIS uppercase letter. All other values are returned unchanged.

The **_tojlower** and **_tojupper** routines are macros that function like the **tojlower** and **tojupper** subroutines, but are faster and have no error–checking function.

The **toujis** subroutine sets all parameter bits that are not 16–bit SJIS code to 0.

The **kutentojis** subroutine converts a kuten code to the corresponding SJIS code. The **kutentojis** routine returns 0 if the given kuten code is invalid.

The **tojhira** subroutine converts an SJIS katakana character to its SJIS hiragana equivalent. Any value that is not an SJIS katakana character is returned unchanged.

The **tojkata** subroutine converts an SJIS hiragana character to its SJIS katakana equivalent. Any value that is not an SJIS hiragana character is returned unchanged.

The **_tojhira** and **_tojkata** subroutines attempt the same conversions without checking for valid input.

For all functions except the **toujis** subroutine, the out–of–range parameter values are returned without conversion.

Parameters

Character	Character to be converted.
Pointer	Pointer to the escape sequence.
CharacterPointer	Pointer to a single **NLchar** data type.

Implementation Specifics

These subroutines are part of Base Operating System (BOS) Runtime.

Related Information

The **ctype** subroutine, **conv** subroutine, **getc**, **getchar**, **fgetc**, or **getw** subroutine, **getwc**, **fgetwc**, or **getwchar** subroutine, **setlocale** subroutine.

List of Character Manipulation Services, National Language Support Overview for Programming, Subroutines Overview in *AIX General Programming Concepts : Writing and Debugging Programs*.

Japanese ctype Subroutines

Purpose

Classify characters.

Library

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

Syntax

```
#include <ctype.h>

int isjalpha (Character)
int Character;

int isjupper (Character)
int Character;

int isjlower (Character)
int Character;

int isjlbytekana (Character)
int Character;

int isjdigit (Character)
int Character;

int isjxdigit (Character)
int Character;

int isjalnum (Character)
int Character;

int isjspace (Character)
int Character;

int isjpunct (Character)
int Character;

int isjparen (Character)
int Character;

int isparent (Character)
intCharacter;

int isjprint (Character)
int Character;

int isjgraph (Character)
int Character;

int isjis (Character)
int Character;

int isjhira (wc)
wchar_t wc;

int isjkanji (wc)
wchar_wc;

int isjkata (wc)
wchar_t wc;
```

Description

The **Japanese ctype** subroutines classify character–coded integer values specified in a table. Each of these subroutines returns a nonzero value for True and 0 for False.

Parameters

Character	Character to be tested.

Return Values

The **isjprint** and **isjgraph** subroutines return a 0 value for user–defined characters.

Implementation Specifics

These subroutines are part of Base Operating System (BOS) Runtime.

Related Information

The **ctype** subroutines, **setlocale** subroutine.

List of Character Manipulation Services, National Language Support Overview for Programming, Subroutines Overview in *AIX General Programming Concepts : Writing and Debugging Programs*.

kill or killpg Subroutine

Purpose

Sends a signal to a process or to a group of processes.

Library

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

Syntax

```
#include <sys/types.h>
#include <signal.h>

int kill(
Process,
Signal)
pid_t Process;
int Signal;

killpg(
ProcessGroup, Signal)
int ProcessGroup, Signal;
```

Description

The **kill** subroutine sends the signal specified by the *Signal* parameter to the process or group of processes specified by the *Process* parameter.

To send a signal to another process, either the real or the effective user ID of the sending process must match the real or effective user ID of the receiving process, and the calling process must have root user authority.

The processes that have the process IDs of 0 and 1 are special processes and are sometimes referred to here as *proc0* and *proc1*, respectively.

Processes can send signals to themselves.

Note: Sending a signal does not imply that the operation is successful. All signal operations must pass the access checks prescribed by each enforced access control policy on the system.

Parameters

Process	Specifies the ID of a process or group of processes. If the *Process* parameter is greater than 0, the signal specified by the *Signal* parameter is sent to the process identified by the *Process* parameter. If the *Process* parameter is 0, the signal specified by the *Signal* parameter is sent to all processes, excluding *proc0* and *proc1*, whose process group ID matches the process group ID of the sender. If the value of the *Process* parameter is a negative value other than –1 and if the calling process passes the access checks for the process to be signaled, the signal specified by the *Signal* parameter is sent to all the processes, excluding *proc0* and *proc1*. If the user ID of the calling process has root user authority, all processes, excluding *proc0* and *proc1*, are signaled. If the value of the *Process* parameter is a negative value other than –1, the signal specified by the *Signal* parameter is sent to all processes having a process group ID equal to the absolute value of the *Process* parameter. If the value of the *Process* parameter is –1, the signal specified by the *Signal* parameter is sent to all processes which the process has permission to send that signal. If pid is –1, sig will be sent to all processes (excluding an unspecified set of system processes) for which the process has permission to send that signal.
Signal	Specifies the signal. If the Signal parameter is a null value, error checking is performed but no signal is sent. This parameter is used to check the validity of the *Process* parameter.
ProcessGroup	Specifies the process group.

Return Values

Upon successful completion, the **kill** subroutine returns a value of 0. Otherwise, a value of –1 is returned and the **errno** global variable is set to indicate the error.

Error Codes

The **kill** subroutine is unsuccessful and no signal is sent if one or more of the following are true:

EINVAL	The *Signal* parameter is not a valid signal number.
EINVAL	The *Signal* parameter specifies the **SIGKILL**, **SIGSTOP**, **SIGTSTP**, or **SIGCONT** signal, and the *Process* parameter is 1 (*proc1*).
ESRCH	No process can be found corresponding to that specified by the *Process* parameter.
EPERM	The real or effective user ID does not match the real or effective user ID of the receiving process, or else the calling process does not have root user authority.

Implementation Specifics

These subroutines are part of Base Operating System (BOS) Runtime.

The following interface is provided for BSD Compatibility:

```
killpg(ProcessGroup, Signal)
int ProcessGroup; Signal;
```

This interface is equivalent to:

```
if (ProcessGroup < 0)
{
  errno = ESRCH;
  return (-1);
}
return (kill(-ProcessGroup, Signal));
```

Related Information

The **getpid**, **getpgrp**, or **getppid** subroutine, **setpgid** or **setpgrp** subroutine, **sigaction**, **sigvec**, or **signal** subroutine.

The **kill** command.

Signal Management in *AIX General Programming Concepts : Writing and Debugging Programs* provides more information about signal management in multi–threaded processes.

kleenup Subroutine

Purpose

Cleans up the run–time environment of a process.

Library

Syntax

```
int kleenup(FileDescriptor, SigIgn, SigKeep)
int FileDescriptor;
int SigIgn[ ];
int SigKeep[ ];
```

Description

The **kleenup** subroutine cleans up the run–time environment for a trusted process by:

- Closing unnecessary file descriptors.
- Resetting the alarm time.
- Resetting signal handlers.
- Clearing the value of the **real directory read** flag described in the **ulimit** subroutine.
- Resetting the **ulimit** value, if it is less than a reasonable value (8192).

Parameters

FileDescriptor	Specifies a file descriptor. The **kleenup** subroutine closes all file descriptors greater than or equal to the *FileDescriptor* parameter.
SigIgn	Points to a list of signal numbers. If these are nonnull values, this list is terminated by 0s. Any signals specified by the *SigIgn* parameter are set to **SIG_IGN**. The handling of all signals not specified by either this list or the *SigKeep* list are set to **SIG_DFL**. Some signals cannot be reset and are left unchanged.
SigKeep	Points to a list of signal numbers. If these are nonnull values, this list is terminated by 0s. The handling of any signals specified by the *SigKeep* parameter is left unchanged. The handling of all signals not specified by either this list or the *SigIgn* list are set to **SIG_DFL**. Some signals cannot be reset and are left unchanged.

Return Values

The **kleenup** subroutine is always successful and returns a value of 0. Errors in closing files are not reported. It is not an error to attempt to modify a signal that the process is not allowed to handle.

Implementation Specifics

This subroutine is part of Base Operating System (BOS) Runtime.

Related Information

The **ulimit** subroutine.

List of Security and Auditing Subroutines and Subroutines Overview in *AIX General Programming Concepts : Writing and Debugging Programs*.

knlist Subroutine

Purpose

Translates names to addresses in the running system.

Syntax

```
#include <nlist.h>

int knlist(NList, NumberOfElements, Size)
struct nlist *NList;
int NumberOfElements;
int Size;
```

Description

The **knlist** subroutine allows a program to examine the list of symbols exported by kernel routines to other kernel modules.

The first field in the **nlist** structure is an input parameter to the **knlist** subroutine. The **n_value** field is modified by the **knlist** subroutine, and all the others remain unchanged. The **nlist** structure consists of the following fields:

`char *n_name`	Specifies the name of the symbol whose attributes are to be retrieved.
`long n_value`	Indicates the virtual address of the object. This will also be the offset when using segment descriptor 0 as the extension parameter of the **readx** or **writex** subroutines against the **/dev/mem** file.

If the name is not found, all fields, other than `n_name`, are set to 0.

The **nlist.h** file is automatically included by the **a.out.h** file for compatibility. However, do not include the **a.out.h** file if you only need the information necessary to use the **knlist** subroutine. If you do include the **a.out.h** file, follow the **#include** statement with the line:

```
#undef n_name
```

Notes:

1. If both the **nlist.h** and **netdb.h** files are to be included, the **netdb.h** file should be included before the **nlist.h** file in order to avoid a conflict with the **n_name** structure member. Likewise, if both the **a.out.h** and **netdb.h** files are to be included, the **netdb.h** file should be included before the **a.out.h** file to avoid a conflict with the **n_name** structure.
2. If the **netdb.h** file and either the **nlist.h** or **syms.h** file are included, the `n_name` field will be defined as `_n._n_name`. This definition allows you to access the `n_name` field in the **nlist** or **syment** structure. If you need to access the `n_name` field in the **netent** structure, undefine the `n_name` field by entering:

   ```
   #undef n_name
   ```

   before accessing the `n_name` field in the **netent** structure. If you need to access the `n_name` field in a **syment** or **nlist** structure after undefining it, redefine the `n_name` field with:

   ```
   #define n_name _n._n_name
   ```

Parameters

NList	Points to an array of **nlist** structures.
NumberOfElements	Specifies the number of structures in the array of **nlist** structures.
Size	Specifies the size of each structure.

Return Values

Upon successful completion, **knlist** returns a value of 0. Otherwise, a value of –1 is returned, and the **errno** global variable is set to indicate the error.

Error Codes

The **knlist** subroutine fails when the following is true:

EFAULT	Indicates that the *NList* parameter points outside the limit of the array of **nlist** structures.

Implementation Specifics

This subroutine is part of Base Operating System (BOS) Runtime.

Related Information

The **nlist** subroutine.

_lazySetErrorHandler Subroutine

Purpose

Installs an error handler into the lazy loading runtime system for the current process.

Library

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

Syntax

```
#include <sys/ldr.h>
#include <sys/errno.h>

typedef void (*_handler_t(
char *_module,
char *_symbol,
unsigned int _errVal ))();

handler_t *_lazySetErrorHandler(err_handler)
handler_t *err_handler;
```

Description

This function allows a process to install a custom error handler to be called when a lazy loading reference fails to find the required module or function. This function should only be used when the main program or one of its dependent modules was linked with the **–blazy** option. To call **_lazySetErrorHandler** from a module that is not linked with the **–blazy** option, you must use the **–lrtl** option. If you use **–blazy**, you do not need to specify **–lrtl**.

This function is not thread safe. The calling program should ensure that **_lazySetErrorHandler** is not called by multiple threads at the same time.

The user–supplied error handler may print its own error message, provide a substitute function to be used in place of the called function, or call the **longjmp** subroutine. To provide a substitute function that will be called instead of the originally referenced function, the error handler should return a pointer to the substitute function. This substitute function will be called by all subsequent calls to the intended function from the same module. If the value returned by the error handler appears to be invalid (for example, a NULL pointer), the default error handler will be used.

Each calling module resolves its lazy references independent of other modules. That is, if module A and B both call **foo** subroutine in module C, but module C does not export **foo** subroutine, the error handler will be called once when **foo** subroutine is called for the first time from A, and once when **foo** subroutine is called for the first time from B.

The default lazy loading error handler will print a message containing: the name of module that the program required; the name of the symbol being accessed; and the error value generated by the failure. Since the default handler considers a lazy load error to be fatal, the process will exit with a status of 1.

During execution of a program that utilizes lazy loading, there are a few conditions that may cause an error to occur. In all cases the current error handler will be called.

1. The referenced module (which is to be loaded upon function invocation) is unavailable or cannot be loaded. The *errVal* parameter will probably indicate the reason for the error if a system call failed.
2. A function is referenced, but the loaded module does not contain a definition for the function. In this case, *errVal* parameter will be **EINVAL**.

Some possibilities as to why either of these errors might occur:

1. The **LIBPATH** environment variable may contain a set of search paths that cause the application to load the wrong version of a module.
2. A module has been changed and no longer provides the same set of symbols that it did when the application was built.
3. The **load** subroutine fails due to a lack of resources available to the process.

Parameters

err_handler A pointer to the new error handler function. The new function should accept 3 arguments:

module The name of the referenced module.

symbol The name of the function being called at the time the failure occurred.

errVal The value of **errno** at the time the failure occurred, if a system call used to load the module fails. For other failures, errval may be **EINVAL** or **ENOMEM**.

Note that the value of module or symbol may be NULL if the calling module has somehow been corrupted.

If the *err_handler* parameter is NULL, the default error handler is restored.

Return Value

The function returns a pointer to the previous user–supplied error handler, or NULL if the default error handler was in effect.

Implementation Specifics

The **_lazySetErrorHandler** subroutine is part of Base Operating System (BOS) Runtime.

Related Information

The **load** subroutine.

The **ld** command.

The Shared Library Overview and Subroutines Overview in *AIX Version 4.2 General Programming Concepts*.

The Shared Library and Lazy Loading in *AIX Version 4.2 General Programming Concepts*.

l3tol or ltol3 Subroutine

Purpose

Converts between 3–byte integers and long integers.

Library

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

Syntax

```
void l3tol (LongPointer, CharacterPointer, Number)
long *LongPointer;
char *CharacterPointer;
int Number;

void ltol3 (CharacterPointer, LongPointer, Number)
char *CharacterPointer;
long *LongPointer;
int Number;
```

Description

The **l3tol** subroutine converts a list of the number of 3–byte integers specified by the *Number* parameter packed into a character string pointed to by the *CharacterPointer* parameter into a list of long integers pointed to by the *LongPointer* parameter.

The **ltol3** subroutine performs the reverse conversion, from long integers (the *LongPointer* parameter) to 3–byte integers (the *CharacterPointer* parameter).

These functions are useful for file system maintenance where the block numbers are 3 bytes long.

Parameters

LongPointer	Specifies the address of a list of long integers.
CharacterPointer	Specifies the address of a list of 3–byte integers.
Number	Specifies the number of list elements to convert.

Implementation Specifics

These subroutines are part of Base Operating System (BOS) Runtime.

Related Information

The **filsys.h** file format.

Subroutines Overview in *AIX General Programming Concepts : Writing and Debugging Programs*.

l64a_r Subroutine

Purpose

Converts base–64 long integers to strings.

Library

Thread–Safe C Library (**libc_r.a**)

Syntax

```
#include <stdlib.h>

int l64a_r (Convert, Buffer, Length)
long Convert;
char *Buffer;
int Length;
```

Description

The **l64a_r** subroutine converts a given long integer into a base–64 string.

For base–64 characters, the following ASCII characters are used:

.	Represents 0.
/	Represents 1.
0 –9	Represents the numbers 2–11.
A–Z	Represents the numbers 12–37.
a–z	Represents the numbers 38–63.

The **l64a_r** subroutine places the converted base–64 string in the buffer pointed to by the *Buffer* parameter.

Parameters

Convert	Specifies the long integer that is to be converted into a base–64 ASCII string.
Buffer	Specifies a working buffer to hold the converted long integer.
Length	Specifies the length of the *Buffer* parameter.

Return Values

0	Indicates that the subroutine was successful.
–1	Indicates that the subroutine was not successful. If the **l64a_r** subroutine is not successful, the **errno** global variable is set to indicate the error.

Error Codes

If the **l64a_r** subroutine is not successful, it returns the following error code:

EINVAL	The *Buffer* parameter value is invalid or too small to hold the resulting ASCII string.

Implementation Specifics

These subroutines are part of Base Operating System (BOS) Runtime.

Programs using this subroutine must link to the **libpthreads.a** library.

Related Information

Subroutines Overview in *AIX General Programming Concepts : Writing and Debugging Programs*.

List of Multithread Subroutines in *AIX General Programming Concepts : Writing and Debugging Programs*.

layout_object_create Subroutine

Purpose

Initializes a layout context.

Library

Layout Library (**libi18n.a**)

Syntax

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

int layout_object_create (locale_name, layout_object)
const char *locale_name;
LayoutObject *layout_object;
```

Description

The **layout_object_create** subroutine creates the **LayoutObject** structure associated with the locale specified by the *locale_name* parameter. The **LayoutObject** structure is a symbolic link containing all the data and methods necessary to perform the layout operations on context dependent and bidirectional characters of the locale specified.

When the **layout_object_create** subroutine completes without errors, the *layout_object* parameter points to a valid **LayoutObject** structure that can be used by other BIDI subroutines. The returned **LayoutObject** structure is initialized to an initial state that defines the behavior of the BIDI subroutines. This initial state is locale dependent and is described by the layout values returned by the **layout_ object_getvalue** subroutine. You can change the layout values of the **LayoutObject** structure using the **layout_object_setvalue** subroutine. Any state maintained by the **LayoutObject** structure is independent of the current global locale set with the **setlocale** subroutine.

Note: If you are developing internationalized applications that may support multibyte locales, please see Use of the libcur Package in *AIX General Programming Concepts : Writing and Debugging Programs*

Parameters

locale_name	Specifies a locale. It is recommended that you use the **LC_CTYPE** category by calling the **setlocale** (**LC_CTYPE**,NULL) subroutine.
layout_object	Points to a valid **LayoutObject** structure that can be used by other layout subroutines. This parameter is used only when the **layout_object_create** subroutine completes without errors. The *layout_object* parameter is not set and a non–zero value is returned if a valid **LayoutObject** structure cannot be created.

Return Values

Upon successful completion, the **layout_object_create** subroutine returns a value of 0. The *layout_object* parameter points to a valid handle.

Error Codes

If the **layout_object_create** subroutine fails, it returns the following error codes:

LAYOUT_EINVAL	The locale specified by the *locale_name* parameter is not available.
LAYOUT_EMFILE	The OPEN_MAX value of files descriptors are currently open in the calling process.
LAYOUT_ENOMEM	Insufficient storage space is available.

Implementation Specifics

This subroutine is part of Base Operating System (BOS) Runtime.

Related Information

The **layout_object_editshape** subroutine, **layout_object_free** subroutine, **layout_object_getvalue** subroutine, **layout_object_setvalue** subroutine, **layout_object_shapeboxchars** subroutine, **layout_object_transform** subroutine.

Bidirectionality and Arabic Character Shaping Overview, and National Language Support Overview for Programming in *AIX General Programming Concepts : Writing and Debugging Programs*.

layout_object_editshape or wcslayout_object_editshape Subroutine

Purpose

Edits the shape of the context text.

Library

Layout library (**libi18n.a**)

Syntax

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

int layout_editshape (layout_object, EditType, index, InpBuf,
Inpsize, OutBuf, OutSize)
LayoutObject layout_object;
BooleanValue EditType;
size_t *index;
const char *InpBuf;
size_t *Inpsize;
void *OutBuf;
size_t *OutSize;

int wcslayout_object_editshape(layout_object, EditType, index,
InpBuf, Inpsize, OutBuf, OutSize)
LayoutObject layout_object;
BooleanValue EditType;
size_t *index;
const wchar t *InpBuf;
size_t *InpSize;
void *OutBuf;
size_t *OutSize;
```

Description

The **layout_object_editshape** and **wcslayout_object_editshape** subroutines provide the shapes of the context text. The shapes are defined by the code element specified by the *index* parameter and any surrounding code elements specified by the ShapeContextSize layout value of the **LayoutObject** structure. The *layout_object* parameter specifies this **LayoutObject** structure.

Use the **layout_object_editshape** subroutine when editing code elements of one byte. Use the **wcslayout_object_editshape** subroutine when editing single code elements of multibytes. These subroutines do not affect any state maintained by the **layout_object_transform** or **wcslayout_object_transform** subroutine.

Note: If you are developing internationalized applications that may support multibyte locales, please see Use of the libcur Package in *AIX General Programming Concepts : Writing and Debugging Programs*

Parameters

layout_object
: Specifies the **LayoutObject** structure created by the **layout_object_create** subroutine.

EditType
: Specifies the type of edit shaping. When the *EditType* parameter stipulates the `EditInput` field, the subroutine reads the current code element defined by the *index* parameter and any preceding code elements defined by ShapeContextSize layout value of the **LayoutObject** structure. When the *EditType* parameter stipulates the `EditReplace` field, the subroutine reads the current code element defined by the *index* parameter and any surrounding code elements defined by ShapeContextSize layout value of the **LayoutObject** structure.

Note: The editing direction defined by the Orientation and TEXT_VISUAL of the TypeOfText layout values of the **LayoutObject** structure determines which code elements are preceding and succeeding.

When the ActiveShapeEditing layout value of the **LayoutObject** structure is set to True, the **LayoutObject** structure maintains the state of the `EditInput` field that may affect subsequent calls to these subroutines with the `EditInput` field defined by the *EditType* parameter. The state of the `EditInput` field of **LayoutObject** structure is not affected when the *EditType* parameter is set to the `EditReplace` field. To reset the state of the `EditInput` field to its initial state, call these subroutines with the *InpBuf* parameter set to NULL. The state of the `EditInput` field is not affected if errors occur within the subroutines.

index
: Specifies an offset (in bytes) to the start of a code element in the *InpBuf* parameter on input. The *InpBuf* parameter provides the base text to be edited. In addition, the context of the surrounding code elements is considered where the minimum set of code elements needed for the specific context dependent script(s) is identified by the ShapeContextSize layout value.

If the set of surrounding code elements defined by the *index*, *InpBuf*, and *InpSize* parameters is less than the size of front and back of the ShapeContextSize layout value, these subroutines assume there is no additional context available. The caller must provide the minimum context if it is available. The *index* parameter is in units associated with the type of the *InpBuf* parameter.

On return, the *index* parameter is modified to indicate the offset to the first code element of the *InpBuf* parameter that required shaping. The number of code elements that required shaping is indicated on return by the *InpSize* parameter.

InpBuf
: Specifies the source to be processed. A Null value with the `EditInput` field in the *EditType* parameter indicates a request to reset the state of the `EditInput` field to its initial state.

Any portion of the *InpBuf* parameter indicates the necessity for redrawing or shaping.

InpSize Specifies the number of code elements to be processed in units on input. These units are associated with the types for these subroutines. A value of –1 indicates that the input is delimited by a Null code element.

On return, the value is modified to the actual number of code elements needed by the *InpBuf* parameter. A value of 0 when the value of the *EditType* parameter is the `EditInput` field indicates that the state of the `EditInput` field is reset to its initial state. If the *OutBuf* parameter is not NULL, the respective shaped code elements are written into the *OutBuf* parameter.

OutBuf Contains the shaped output text. You can specify this parameter as a Null pointer to indicate that no transformed text is required. If Null, the subroutines return the *index* and *InpSize* parameters, which specify the amount of text required, to be redrawn.

The encoding of the *OutBuf* parameter depends on the ShapeCharset layout value defined in *layout_object* parameter. If the ActiveShapeEditing layout value is set to False, the encoding of the *OutBuf* parameter is to be the same as the code set of the locale associated with the specified **LayoutObject** structure.

OutSize Specifies the size of the output buffer on input in number of bytes. Only the code elements required to be shaped are written into the *OutBuf* parameter.

The output buffer should be large enough to contain the shaped result; otherwise, only partial shaping is performed. If the ActiveShapeEditing layout value is set to True, the *OutBuf* parameter should be allocated to contain at least the number of code elements in the *InpBuf* parameter multiplied by the value of the ShapeCharsetSize layout value.

On return, the *OutSize* parameter is modified to the actual number of bytes placed in the output buffer.

When the *OutSize* parameter is specified as 0, the subroutines calculate the size of an output buffer large enough to contain the transformed text from the input buffer. The result will be returned in this field. The content of the buffers specifies by the *InpBuf* and *OutBuf* parameters, and the value of the *InpSize* parameter, remain unchanged.

Return Values

Upon successful completion, these subroutines return a value of 0. The *index* and *InpSize* parameters return the minimum set of code elements required to be redrawn.

Error Codes

If these subroutines fail, they return the following error codes:

LAYOUT_EILSEQ
Shaping stopped due to an input code element that cannot be shaped. The *index* parameter indicates the code element that caused the error. This code element is either a valid code element that cannot be shaped according to the ShapeCharset layout value or an invalid code element not defined by the code set defined in the **LayoutObject** structure. Use the **mbtowc** or **wctomb** subroutine in the same locale as the **LayoutObject** structure to determine if the code element is valid.

LAYOUT_E2BIG
The output buffer is too small and the source text was not processed. The *index* and *InpSize* parameters are not guaranteed on return.

LAYOUT_EINVAL
Shaping stopped due to an incomplete code element or shift sequence at the end of input buffer. The *InpSize* parameter indicates the number of code elements successfully transformed.

Note: You can use this error code to determine the code element causing the error.

LAYOUT_ERANGE
Either the *index* parameter is outside the range as defined by the *InpSize* parameter, more than 15 embedding levels are in the source text, or the *InpBuf* parameter contains unbalanced Directional Format (Push/Pop).

Implementation Specifics

This subroutine is part of Base Operating System (BOS) Runtime.

Related Information

The **layout_object_create** subroutine, **layout_object_free** subroutine, **layout_object_getvalue** subroutine, **layout_object_setvalue** subroutine, **layout_object_shapeboxchars** subroutine, **layout_object_transform** subroutine.

Bidirectionality and Arabic Character Shaping Overview in *AIX General Programming Concepts : Writing and Debugging Programs*.

layout_object_free Subroutine

Purpose

Frees a **LayoutObject** structure.

Library

Layout library (**libi18n.a**)

Syntax

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

int layout_object_free(layout_object)
LayoutObject layout_object;
```

Description

The **layout_object_free** subroutine releases all the resources of the **LayoutObject** structure created by the **layout_object_create** subroutine. The *layout_object* parameter specifies this **LayoutObject** structure.

Note: If you are developing internationalized applications that may support multibyte locales, please see Use of the libcur Package in *AIX General Programming Concepts : Writing and Debugging Programs*

Parameters

layout_object	Specifies a **LayoutObject** structure returned by the **layout_object_create** subroutine.

Return Values

Upon successful completion, the **layout_object_free** subroutine returns a value of 0. All resources associated with the *layout_object* parameter are successfully deallocated.

Error Codes

If the **layout_object_free** subroutine fails, it returns the following error code:

LAYOUT_EFAULT	Errors occurred while processing the request.

Implementation Specifics

This subroutine is part of Base Operating System (BOS) Runtime.

Related Information

The **layout_object_create** subroutine, **layout_object_editshape** subroutine, **layout_object_getvalue** subroutine, **layout_object_setvalue** subroutine, **layout_object_shapeboxchars** subroutine, **layout_object_transform** subroutine.

Bidirectionality and Arabic Character Shaping Overview in *AIX General Programming Concepts : Writing and Debugging Programs*.

layout_object_getvalue Subroutine

Purpose

Queries the current layout values of a **LayoutObject** structure.

Library

Layout Library (**libi18n.a**)

Syntax

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

int layout_object_getvalue(layout_object, values, index)
LayoutObject layout_object;
LayoutValues values;
int *index;
```

Description

The **layout_object_getvalue** subroutine queries the current setting of layout values within the **LayoutObject** structure. The *layout_object* parameter specifies the **LayoutObject** structure created by the **layout_object_create** subroutine.

The name field of the LayoutValues structure contains the name of the layout value to be queried. The value field is a pointer to where the layout value is stored. The values are queried from the **LayoutObject** structure and represent its current state.

For example, if the layout value to be queried is of type T, the *value* parameter must be of type T*. If T itself is a pointer, the **layout_object_getvalue** subroutine allocates space to store the actual data. The caller must free this data by calling the **free(**T**)** subroutine with the returned pointer.

When setting the value field, an extra level of indirection is present that is not present using the **layout_object_setvalue** parameter. When you set a layout value of type T, the value field contains T. However, when querying the same layout value, the value field contains &T.

Note: If you are developing internationalized applications that may support multibyte locales, please see Use of the libcur Package in *AIX General Programming Concepts : Writing and Debugging Programs*

Parameters

layout_object	Specifies the **LayoutObject** structure created by the **layout_object_create** subroutine.
values	Specifies an array of layout values of type LayoutValueRec that are to be queried in the **LayoutObject** structure. The end of the array is indicated by `name=0`.
index	Specifies a layout value to be queried. If the value cannot be queried, the *index* parameter causing the error is returned and the subroutine returns a non–zero value.

Return Values

Upon successful completion, the **layout_object_getvalue** subroutine returns a value of 0. All layout values were successfully queried.

Error Codes

If the **layout_object_getvalue** subroutine fails, it returns the following values:

LAYOUT_EINVAL The layout value specified by the *index* parameter is unknown or the *layout_object* parameter is invalid.

LAYOUT_EMOMEM Insufficient storage space is available.

Examples

The following example queries whether the locale is bidirectional and gets the values of the `in` and `out` orienations.

```
#include <sys/lc_layout.h>
#include <locale.h>
main()
{
LayoutObject plh;
int RC=0;
LayoutValues layout;
LayoutTextDescriptor Descr;
int index;

RC=layout_object_create(setlocale(LC_CTYPE,””),&plh); /* create
object */
if (RC) {printf(”Create error !!\n”); exit(0);}

layout=malloc(3*sizeof(LayoutValueRec));
                                          /* allocate layout array
*/
layout[0].name=ActiveBidirection;         /* set name */
layout[1].name=Orientation;               /* set name */
layout[1].value=(caddr_t)&Descr;
            /* send address of memory to be allocated by function
*/

layout[2].name=0;                      /* indicate end of array */
RC=layout_object_getvalue(plh,layout,&index);
if (RC) {printf(”Getvalue error at %d !!\n”,index); exit(0);}
printf(”ActiveBidirection = %d \n”,*(layout[0].value));
                                                     /*print
output*/
printf(”Orientation in = %x  out = %x \n”, Descr->>in,
Descr->>out);

free(layout);                         /* free layout array */
free (Descr);             /* free memory allocated by function */
RC=layout_object_free(plh);         /* free layout object */
if (RC) printf(”Free error !!\n”);
}
```

Implementation Specifics

This subroutine is part of Base Operating System (BOS) Runtime.

Related Information

The **layout_object_create** subroutine, **layout_object_editshape** subroutine, **layout_object_free** subroutine, **layout_object_setvalue** subroutine, **layout_object_shapeboxchars** subroutine, **layout_object_transform** subroutine.

Bidirectionality and Arabic Character Shaping Overview in *AIX General Programming Concepts : Writing and Debugging Programs*.

layout_object_setvalue Subroutine

Purpose

Sets the layout values of a **LayoutObject** structure.

Library

Layout Library (**libi18n.a**)

Syntax

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

int layout_object_setvalue(layout_object, values, index)
LayoutObject layout_object;
LayoutValues values;
int *index;
```

Description

The **layout_object_setvalue** subroutine changes the current layout values of the **LayoutObject** structure. The *layout_object* parameter specifies the **LayoutObject** structure created by the **layout_object_create** subroutine. The values are written into the **LayoutObject** structure and may affect the behavior of subsequent layout functions.

Note: Some layout values do alter internal states maintained by a **LayoutObject** structure.

The name field of the LayoutValueRec structure contains the name of the layout value to be set. The value field contains the actual value to be set. The value field is large enough to support all types of layout values. For more information on layout value types, see ”Layout Values for the Layout Library” in *AIX General Programming Concepts : Writing and Debugging Programs*.

Note: If you are developing internationalized applications that may support multibyte locales, please see Use of the libcur Package in *AIX General Programming Concepts : Writing and Debugging Programs*

Parameters

layout_object	Specifies the **LayoutObject** structure returned by the **layout_object_create** subroutine.
values	Specifies an array of layout values of the type LayoutValueRec that this subroutine sets. The end of the array is indicated by `name=0`.
index	Specifies a layout value to be queried. If the value cannot be queried, the index parameter causing the error is returned and the subroutine returns a non–zero value. If an error is generated, a subset of the values may have been previously set.

Return Values

Upon successful completion, the **layout_object_setvalue** subroutine returns a value of 0. All layout values were successfully set.

Error Codes

If the **layout_object_setvalue** subroutine fails, it returns the following values:

LAYOUT_EINVAL The layout value specified by the *index* parameter is unknown, its value is invalid, or the *layout_object* parameter is invalid.

LAYOUT_EMFILE The **(OPEN_MAX)** file descriptors are currently open in the calling process.

LAYOUT_ENOMEM Insufficient storage space is available.

Examples

The following example sets the TypeofText value to Implicit and the out value to Visual.

```
#include <sys/lc_layout.h>
#include <locale.h>

main()
{
LayoutObject plh;
int RC=0;
LayoutValues layout;
LayoutTextDescriptor Descr;
int index;

RC=layout_object_create(setlocale(LC_CTYPE,””),&plh); /* create
object */
if (RC) {printf(”Create error !!\n”); exit(0);}

layout=malloc(2*sizeof(LayoutValueRec)); /*allocate layout
array*/  Descr=malloc(sizeof(LayoutTextDescriptorRec)); /*
allocate text descriptor */
layout[0].name=TypeOfText;            /* set name */
layout[0].value=(caddr_t)Descr;       /* set value */
layout[1].name=0;                     /* indicate end of array */


Descr->in=TEXT_IMPLICIT;
Descr->out=TEXT_VISUAL;
RC=layout_object_setvalue(plh,layout,&index);
if (RC) printf(”SetValue error at %d!!\n”,index); /* check return
code */
free(layout);                       /* free alllocated memory */
free (Descr);
RC=layout_object_free(plh);         /* free layout object */
if (RC) printf(”Free error !!\n”);
}
```

Implementation Specifics

This subroutine is part of Base Operating System (BOS) Runtime.

Related Information

The **layout_object_create** subroutine, **layout_object_editshape** subroutine, **layout_object_free** subroutine, **layout_object_getvalue** subroutine, **layout_object_shapeboxchars** subroutine, **layout_object_transform** subroutine.

Bidirectionality and Character Shaping Overview in *AIX General Programming Concepts : Writing and Debugging Programs*.

layout_object_shapeboxchars Subroutine

Purpose

Shapes box characters.

Library

Layout Library (**libi18n.a**)

Syntax

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

int layout_object_shapeboxchars(layout_object, InpBuf,  InpSize,
OutBuf)
LayoutObject layout_object;
const char *InpBuf;
const size_t InpSize;
char *OutBuf;
```

Description

The **layout_object_shapeboxchars** subroutine shapes box characters into the VT100 box character set.

Note: If you are developing internationalized applications that may support multibyte locales, please see Use of the libcur Package in *AIX General Programming Concepts : Writing and Debugging Programs*

Parameters

layout_object	Specifies the **LayoutObject** structure created by the **layout_object_create** subroutine.
InpBuf	Specifies the source text to be processed.
InpSize	Specifies the number of code elements to be processed.
OutBuf	Contains the shaped output text.

Return Values

Upon successful completion, this subroutine returns a value of 0.

Error Codes

If this subroutine fails, it returns the following values:

LAYOUT_EILSEQ	Shaping stopped due to an input code element that cannot be mapped into the VT100 box character set.
LAYOUT_EINVAL	Shaping stopped due to an incomplete code element or shift sequence at the end of the input buffer.

Implementation Specifics

This subroutine is part of Base Operating System (BOS) Runtime.

Related Information

The **layout_object_create** subroutine, **layout_object_editshape** subroutine, **layout_object_free** subroutine, **layout_object_getvalue** subroutine, **layout_object_setvalue** subroutine, **layout_object_transform** subroutine.

Bidirectionalit y and Character Shaping Overview in *AIX General Programming Concepts : Writing and Debugging Programs*.

layout_object_transform or wcslayout_object_transform Subroutine

Purpose

Transforms text according to the current layout values of a **LayoutObject** structure.

Library

Layout Library (**libi18n.a**)

Syntax

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

int layout_object_transform(layout_object, InpBuf, InpSize,
OutBuf, OutSize, InpToOut, OutToInp, BidiLvl)
LayoutObject layout_object;
const char *InpBuf;
size_t *InpSize;
void * OutBuf;
size_t *OutSize;
size_t  *InpToOut;
size_t *OutToInp;
unsigned char *BidiLvl;

int wcslayout_object_transform (layout_object, InpBuf, InpSize,
OutBuf, OutSize, InpToOut, OutToInp, BidiLvl)
LayoutObject layout_object;
const char *InpBuf;
size_t *InpSize;
void *OutBuf;
Size_t *OutSize;
size_t *InpToOut;
size_t *OutToInp;
unsigned char *BidiLvl;
```

Description

The **layout_object_transform** and **wcslayout_object_transform** subroutines transform the text specified by the *InpBuf* parameter according to the current layout values in the **LayoutObject** structure. Any layout value whose type is LayoutTextDescriptor describes the attributes within the *InpBuf* and *OutBuf* parameters. If the attributes are the same as the *InpBuf* and *OutBuf* parameters themselves, a null transformation is done with respect to that specific layout value.

The output of these subroutines may be one or more of the following results depending on the setting of the respective parameters:

OutBuf, *OutSize*	Any transformed data is stored in the *OutBuf* parameter.
InpToOut	A cross reference from each code element of the *InpBuf* parameter to the transformed data.
OutToInp	A cross reference to each code element of the *InpBuf* parameter from the transformed data.
BidiLvl	A weighted value that represents the directional level of each code element of the *InpBuf* parameter. The level is dependent on the internal directional algorithm of the **LayoutObject** structure.

You can specify each of these output parameters as Null to indicate that no output is needed for the specific parameter. However, you should set at least one of these parameters to a nonNULL value to perform any significant work.

To perform shaping of a text string without reordering of code elements, set the TypeOfText layout value to **TEXT_VISUAL** and the in and out values of the Orientation layout value alike. These layout values are in the **LayoutObject** structure.

Note: If you are developing internationalized applications that may support multibyte locales, please see Use of the libcur Package in *AIX General Programming Concepts : Writing and Debugging Programs*

Parameters

layout_object
: Specifies the **LayoutObject** structure created by the **layout_object_create** subroutine.

InpBuf
: Specifies the source text to be processed. This parameter cannot be null.

InpSize
: Specifies the units of code elements processed associated with the bytes for the **layout_object_transform** and **wcslayout_object_transform** subroutines. A value of –1 indicates that the input is delimited by a null code element. On return, the value is modified to the actual number of code elements processed in the *InBuf* parameter. However, if the value in the *OutSize* parameter is zero, the value of the *InpSize* parameter is not changed.

OutBuf
: Contains the transformed data. You can specify this parameter as a null pointer to indicate that no transformed data is required.

  The encoding of the *OutBuf* parameter depends on the ShapeCharset layout value defined in the **LayoutObject** structure. If the ActiveShapeEditing layout value is set to True, the encoding of the *OutBuf* parameter is the same as the code set of the locale associated with the **LayoutObject** structure.

OutSize
: Specifies the size of the output buffer in number of bytes. The output buffer should be large enough to contain the transformed result; otherwise, only a partial transformation is performed. If the ActiveShapeEditing layout value is set to True, the *OutBuf* parameter should be allocated to contain at least the number of code elements multiplied by the ShapeCharsetSize layout value.

  On return, the *OutSize* parameter is modified to the actual number of bytes placed in this parameter.

  When you specify the *OutSize* parameter as 0, the subroutine calculates the size of an output buffer to be large enough to contain the transformed text. The result is returned in this field. The content of the buffers specified by the *InpBuf* and *OutBuf* parameters, and a value of the *InpSize* parameter remains unchanged.

InpToOut
: Represents an array of values with the same number of code elements as the *InpBuf* parameter if *InpToOut* parameter is not a null pointer.

  On output, the *n*th value in *InpToOut* parameter corresponds to the *n*th code element in *InpBuf* parameter. This value is the index in *OutBuf* parameter which identifies the transformed ShapeCharset element of the *n*th code element in *InpBuf* parameter. You can specify the *InpToOut* parameter as null if no index array from the *InpBuf* to *OutBuf* parameters is desired.

OutToInp Represents an array of values with the same number of code elements as contained in the *OutBuf* parameter if the *OutToInp* parameter is not a null pointer.

On output, the *n*th value in the *OutToInp* parameter corresponds to the *n*th ShapeCharset element in the *OutBuf* parameter. This value is the index in the *InpBuf* parameter which identifies the original code element of the *n*th ShapeCharset element in the *OutBuf* parameter. You can specify the *OutToInp* parameter as NULL if no index array from the *OutBuf* to *InpBuf* parameters is desired.

BidiLvl Represents an array of values with the same number of elements as the source text if the *BidiLvl* parameter is not a null pointer. The *n*th value in the *BidiLvl* parameter corresponds to the *n*th code element in the *InpBuf* parameter. This value is the level of this code element as determined by the bidirectional algorithm. You can specify the *BidiLvl* parameter as null if a levels array is not desired.

Return Values

Upon successful completion, these subroutines return a value of 0.

Error Codes

If these subroutines fail, they return the following values:

LAYOUT_EILSEQ Transformation stopped due to an input code element that cannot be shaped or is invalid. The *InpSize* parameter indicates the number of the code element successfully transformed.

Note: You can use this error code to determine the code element causing the error.

This code element is either a valid code element but cannot be shaped into the ShapeCharset layout value or is an invalid code element not defined by the code set of the locale of the **LayoutObject** structure. You can use the **mbtowc** and **wctomb** subroutines to determine if the code element is valid when used in the same locale as the **LayoutObject** structure.

LAYOUT_E2BIG The output buffer is full and the source text is not entirely processed.

LAYOUT_EINVAL Transformation stopped due to an incomplete code element or shift sequence at the end of the input buffer. The *InpSize* parameter indicates the number of the code elements successfully transformed.

Note: You can use this error code to determine the code element causing the error.

LAYOUT_ERANGE More than 15 embedding levels are in the source text or the *InpBuf* parameter contains unbalanced Directional Format (Push/Pop).

When the size of *OutBuf* parameter is not large enough to contain the entire transformed text, the input text state at the end of the **LAYOUT_E2BIG** error code is returned. To resume the transformation on the remaining text, the application calls the **layout_object_transform** subroutine with the same **LayoutObject** structure, the same *InpBuf* parameter, and *InpSize* parameter set to 0.

Examples

The following is an example of transformation of both directional re–ordering and shaping.

Notes:

1. Uppercase represent left–to–right characters; lowercase represent right–to–left characters.
2. `xyz` represent the shapes of `cde`.

```
Position:             0123456789
InpBuf:               AB cde 12Z

Position:             0123456789
OutBuf:               AB 12 zyxZ

Position:             0123456789
ToTarget:             0128765349

Position:             0123456789
ToSource:             0127865439

Position:             0123456789
BidiLevel:            0001111220
```

Implementation Specifics

This subroutine is part of Base Operating System (BOS) Runtime.

Related Information

The **layout_object_create** subroutine, **layout_object_editshape** subroutine, **layout_object_free** subroutine, **layout_object_getvalue** subroutine, **layout_object_setvalue** subroutine, **layout_object_shapeboxchars** subroutine.

Bidirectionality and Character Shaping Overview in *AIX General Programming Concepts : Writing and Debugging Programs*.

ldahread Subroutine

Purpose

Reads the archive header of a member of an archive file.

Library

Object File Access Routine Library (**libld.a**)

Syntax

```
#include <stdio.h>
#include <ar.h>
#include <ldfcn.h>

int ldahread(ldPointer, ArchiveHeader)
LDFILE *ldPointer;
ARCHDR *ArchiveHeader;
```

Description

If the **TYPE(***ldPointer***)** macro from the **ldfcn.h** file is the archive file magic number, the **ldahread** subroutine reads the archive header of the extended common object file currently associated with the *ldPointer* parameter into the area of memory beginning at the *ArchiveHeader* parameter.

Parameters

ldPointer	Points to the **LDFILE** structure that was returned as the result of a successful call to **ldopen** or **ldaopen**.
ArchiveHeader	Points to a **ARCHDR** structure.

Return Values

The **ldahread** subroutine returns a SUCCESS or FAILURE value.

Error Codes

The **ldahread** routine fails if the **TYPE(***ldPointer***)** macro does not represent an archive file, or if it cannot read the archive header.

Implementation Specifics

This subroutine is part of Base Operating System (BOS) Runtime.

Related Information

The **ldfhread** subroutine, **ldgetname** subroutine, **ldlread**, **ldlinit**, or **ldlitem** subroutine, **ldshread** or **ldnshread** subroutine, **ldtbread** subroutine.

Subroutines Overview in *AIX General Programming Concepts : Writing and Debugging Programs*.

ldclose or ldaclose Subroutine

Purpose

Closes a common object file.

Library

Object File Access Routine Library (**libld.a**)

Syntax

```
#include <stdio.h>
#include <ldfcn.h>

int ldclose(ldPointer)
LDFILE *ldPointer;

int ldaclose(ldPointer)
LDFILE *ldPointer;
```

Description

The **ldopen** and **ldclose** subroutines provide uniform access to both simple object files and object files that are members of archive files. Thus, an archive of common object files can be processed as if it were a series of simple common object files.

If the **ldfcn.h** file **TYPE(***ldPointer***)** macro is the magic number of an archive file, and if there are any more files in the archive, the **ldclose** subroutine reinitializes the **ldfcn.h** file **OFFSET(***ldPointer***)** macro to the file address of the next archive member and returns a failure value. The **ldfile** structure is prepared for a subsequent **ldopen**.

If the **TYPE(***ldPointer***)** macro does not represent an archive file, the **ldclose** subroutine closes the file and frees the memory allocated to the **ldfile** structure associated with *ldPointer.*

The **ldaclose** subroutine closes the file and frees the memory allocated to the **ldfile** structure associated with the *ldPointer* parameter regardless of the value of the **TYPE**(*ldPointer***)** macro.

Parameters

ldPointer	Pointer to the **LDFILE** structure that was returned as the result of a successful call to **ldopen** or **ldaopen**.

Return Values

The **ldclose** subroutine returns a SUCCESS or FAILURE value.

The **ldaclose** subroutine always returns a SUCCESS value and is often used in conjunction with the **ldaopen** subroutine.

Error Codes

The **ldclose** subroutine returns a failure value if there are more files to archive.

Implementation Specifics

These subroutines are part of Base Operating System (BOS) Runtime.

Related Information

The **ldaopen** or **ldopen** subroutine.

Subroutines Overview in *AIX General Programming Concepts : Writing and Debugging Programs*.

ldfhread Subroutine

Purpose

Reads the file header of an XCOFF file.

Library

Object File Access Routine Library (**libld.a**)

Syntax

```
#include <stdio.h>
#include <ldfcn.h>

int ldfhread (ldPointer, FileHeader)
LDFILE *ldPointer;
void *FileHeader;
```

Description

The **ldfhread** subroutine reads the file header of the object file currently associated with the *ldPointer* parameter into the area of memory beginning at the *FileHeader* parameter. For AIX 4.3.2 and above it is the responsibility of the calling routine to provide a pointer to a buffer large enough to contain the file header of the associated object file. Since the **ldopen** subroutine provides magic number information (via the **HEADER**(*ldPointer*)**.f_magic** macro), the calling application can always determine whether the *FileHeader* pointer should refer to a 32–bit FILHDR or 64–bit FILHDR_64 structure.

Parameters

ldPointer	Points to the **LDFILE** structure that was returned as the result of a successful call to **ldopen** or **ldaopen** subroutine.
FileHeader	Points to a buffer large enough to accommodate a **FILHDR** structure, according to the object mode of the file being read.

Return Values

The **ldfhread** subroutine returns Success or Failure.

Error Codes

The **ldfhread** subroutine fails if it cannot read the file header.

Note: In most cases, the use of **ldfhread** can be avoided by using the **HEADER** (*ldPointer*) macro defined in the **ldfcn.h** file. The information in any field or fieldname of the header file may be accessed using the **header** *(ldPointer*) **fieldname** macro.

Examples

The following is an example of code that opens an object file, determines its mode, and uses the **ldfhread** subroutine to acquire the file header. This code would be compiled with both **_XCOFF32_** and **_XCOFF64_** defined:

```
#define __XCOFF32__
#define __XCOFF64__

#include <ldfcn.h>


/* for each FileName to be processed */

if ( (ldPointer = ldopen(fileName, ldPointer)) != NULL)
{
    FILHDR    FileHead32;
    FILHDR_64 FileHead64;
    void      *FileHeader;

    if ( HEADER(ldPointer).f_magic == U802TOCMAGIC )
        FileHeader = &FileHead32;
    else if ( HEADER(ldPointer).f_magic == U803XTOCMAGIC )
        FileHeader = &FileHead64;
    else
        FileHeader = NULL;

    if ( FileHeader && (ldfhread( ldPointer, &FileHeader ) ==
SUCCESS) )
    {
        /* ...successfully read header... */
        /* ...process according to magic number... */
    }
}
```

Implementation Specifics

This subroutine is part of Base Operating System (BOS) Runtime.

Related Information

The **ldahread** subroutine, **ldgetname** subroutine, **ldlread**, **ldlinit**, or **ldlitem** subroutine, **ldopen** subroutine, **ldshread** or **ldnshread** subroutine, **ldtbread** subroutine.

Subroutines Overview in *AIX General Programming Concepts : Writing and Debugging Programs*.

ldgetname Subroutine

Purpose

Retrieves symbol name for common object file symbol table entry.

Library

Object File Access Routine Library (**libld.a)**

Syntax

```
#include <stdio.h>
#include <ldfcn.h>

char *ldgetname (ldPointer, Symbol)
LDFILE *ldPointer;
void *Symbol;
```

Description

The **ldgetname** subroutine returns a pointer to the name associated with *Symbol* as a string. The string is in a static buffer local to the **ldgetname** subroutine that is overwritten by each call to the **ldgetname** subroutine and must therefore be copied by the caller if the name is to be saved.

The common object file format handles arbitrary length symbol names with the addition of a string table. The **ldgetname** subroutine returns the symbol name associated with a symbol table entry for an XCOFF–format object file.

The calling routine to provide a pointer to a buffer large enough to contain a symbol table entry for the associated object file. Since the **ldopen** subroutine provides magic number information (via the **HEADER(***ldPointer***).f_magic** macro), the calling application can always determine whether the Symbol pointer should refer to a 32–bit SYMENT or 64–bit SYMENT_64 structure.

The maximum length of a symbol name is **BUFSIZ**, defined in the **stdio.h** file.

Parameters

ldPointer	Points to an **LDFILE** structure that was returned as the result of a successful call to the **ldopen** or **ldaopen** subroutine.
Symbol	Points to an initialized 32–bit or 64–bit **SYMENT** structure.

Error Codes

The **ldgetname** subroutine returns a null value (defined in the **stdio.h** file) for a COFF–format object file if the name cannot be retrieved. This situation can occur if one of the following is true:

- The string table cannot be found.
- The string table appears invalid (for example, if an auxiliary entry is handed to the **ldgetname** subroutine wherein the name offset lies outside the boundaries of the string table).
- The name's offset into the string table is past the end of the string table.

Typically, the **ldgetname** subroutine is called immediately after a successful call to the **ldtbread** subroutine to retrieve the name associated with the symbol table entry filled by the **ldtbread** subroutine.

Examples

The following is an example of code that determines the object file type before making a call to the **ldtbread** and **ldgetname** subroutines.

```
#define __XCOFF32__
#define __XCOFF64__

#include <ldfcn.h>


SYMENT    Symbol32;
SYMENT_64 Symbol64;
void      *Symbol;

if ( HEADER(ldPointer).f_magic == U802TOCMAGIC )
    Symbol = &Symbol32;
else if ( HEADER(ldPointer).f_magic == U803XTOCMAGIC )
    Symbol = &Symbol64;
else
    Symbol = NULL;

if ( Symbol )
    /* for each symbol in the symbol table */
    for ( symnum = 0 ; symnum < HEADER(ldPointer).f_nsyms ;
symnum++ )    {        if ( ldtbread(ldPointer,symnum,Symbol) ==
SUCCESS )        {            char *name =
ldgetname(ldPointer,Symbol)            if ( name )            {
             /* Got the name... */                 .
 .            }            /* Increment symnum by the number of
auxiliary entries */            if ( HEADER(ldPointer).f_magic ==
U802TOCMAGIC )                symnum += Symbol32.n_numaux;
    else if ( HEADER(ldPointer).f_magic == U803XTOCMAGIC )
        symnum += Symbol64.n_numaux;        }        else
{            /* Should have been a symbol...indicate the error */
              .                .        }    }
```

Implementation Specifics

This subroutine is part of Base Operating System (BOS) Runtime.

Related Information

The **ldahread** subroutine, **ldfhread** subroutine, **ldlread**, **ldlinit**, or **ldlitem** subroutine, **ldshread** or **ldnshread** subroutine, **ldtbread** subroutine.

Subroutines Overview in *AIX General Programming Concepts : Writing and Debugging Programs*.

ldlread, ldlinit, or ldlitem Subroutine

Purpose

Manipulates line number entries of a common object file function.

Library

Object File Access Routine Library (**libld.a**)

Syntax

```
#include <stdio.h>
#include <ldfcn.h>

int ldlread (ldPointer, FunctionIndex, LineNumber, LineEntry)
LDFILE *ldPointer;
int FunctionIndex;
unsigned short LineNumber;
void *LineEntry;

int ldlinit (ldPointer, FunctionIndex)
LDFILE *ldPointer;
int FunctionIndex;

int ldlitem (ldPointer, LineNumber, LineEntry)
LDFILE *ldPointer;
unsigned short LineNumber;
void *LineEntry;
```

Description

The **ldlread** subroutine searches the line number entries of the XCOFF file currently associated with the *ldPointer* parameter. The **ldlread** subroutine begins its search with the line number entry for the beginning of a function and confines its search to the line numbers associated with a single function. The function is identified by the *FunctionIndex* parameter, the index of its entry in the object file symbol table. The **ldlread** subroutine reads the entry with the smallest line number equal to or greater than the *LineNumber* parameter into the memory beginning at the *LineEntry* parameter. It is the responsibility of the calling routine to provide a pointer to a buffer large enough to contain the line number entry for the associated object file type. Since the **ldopen** subroutine provides magic number information (via the **HEADER**(*ldPointer*)**.f_magic** macro), the calling application can always determine whether the *LineEntry* pointer should refer to a 32–bit LINENO or 64–bit LINENO_64 structure.

The **ldlinit** and **ldlitem** subroutines together perform the same function as the **ldlread** subroutine. After an initial call to the **ldlread** or **ldlinit** subroutine, the **ldlitem** subroutine may be used to retrieve successive line number entries associated with a single function. The **ldlinit** subroutine simply locates the line number entries for the function identified by the *FunctionIndex* parameter. The **ldlitem** subroutine finds and reads the entry with the smallest line number equal to or greater than the *LineNumber* parameter into the memory beginning at the *LineEntry* parameter.

Parameters

ldPointer	Points to the **LDFILE** structure that was returned as the result of a successful call to the **ldopen** , **lddopen**,or **ldaopen** subroutine.
LineNumber	Specifies the index of the first *LineNumber* parameter entry to be read.
LineEntry	Points to a buffer that will be filled in with a **LINENO** structure from the object file.
FunctionIndex	Points to the symbol table index of a function.

Return Values

The **ldlread**, **ldlinit**, and **ldlitem** subroutines return a SUCCESS or FAILURE value.

Error Codes

The **ldlread** subroutine fails if there are no line number entries in the object file, if the *FunctionIndex* parameter does not index a function entry in the symbol table, or if it finds no line number equal to or greater than the *LineNumber* parameter. The **ldlinit** subroutine fails if there are no line number entries in the object file or if the *FunctionIndex* parameter does not index a function entry in the symbol table. The **ldlitem** subroutine fails if it finds no line number equal to or greater than the *LineNumber* parameter.

Implementation Specifics

These subroutines are part of Base Operating System (BOS) Runtime.

Related Information

The **ldahread** subroutine, **ldfhread** subroutine, **ldgetname** subroutine, **ldshread** or **ldnshread** subroutine, **ldtbread** subroutine.

Subroutines Overview in *AIX General Programming Concepts : Writing and Debugging Programs*.

ldlseek or ldnlseek Subroutine

Purpose

Seeks to line number entries of a section of a common object file.

Library

Object File Access Routine Library (**libld.a**)

Syntax

```
#include <stdio.h>
#include <ldfcn.h>

int ldlseek (ldPointer, SectionIndex)
LDFILE *ldPointer;
unsigned short SectionIndex;

int ldnlseek (ldPointer, SectionName)
LDFILE *ldPointer;
char *SectionName;
```

Description

The **ldlseek** subroutine seeks to the line number entries of the section specified by the *SectionIndex* parameter of the common object file currently associated with the *ldPointer* parameter. The first section has an index of 1.

The **ldnlseek** subroutine seeks to the line number entries of the section specified by the *SectionName* parameter.

Both subroutines determine the object mode of the associated file before seeking to the relocation entries of the indicated section.

Parameters

ldPointer	Points to the **LDFILE** structure that was returned as the result of a successful call to the **ldopen** or **ldaopen** subroutine.
SectionIndex	Specifies the index of the section whose line number entries are to be seeked to.
SectionName	Specifies the name of the section whose line number entries are to be seeked to.

Return Values

The **ldlseek** and **ldnlseek** subroutines return a SUCCESS or FAILURE value.

Error Codes

The **ldlseek** subroutine fails if the *SectionIndex* parameter is greater than the number of sections in the object file. The **ldnlseek** subroutine fails if there is no section name corresponding with the *SectionName* parameter. Either function fails if the specified section has no line number entries or if it cannot seek to the specified line number entries.

Implementation Specifics

These subroutines are part of Base Operating System (BOS) Runtime.

Related Information

The **ldohseek** subroutine, **ldrseek** or **ldnrseek** subroutine, **ldsseek** or **ldnsseek** subroutine, **ldtbseek** subroutine.

Subroutines Overview in *AIX General Programming Concepts : Writing and Debugging Programs*.

ldohseek Subroutine

Purpose

Seeks to the optional file header of a common object file.

Library

Object File Access Routine Library (**libld.a**)

Syntax

```
#include <stdio.h>
#include <ldfcn.h>

int ldohseek (ldPointer)
LDFILE *ldPointer;
```

Description

The **ldohseek** subroutine seeks to the optional auxiliary header of the common object file currently associated with the *ldPointer* parameter. The subroutine determines the object mode of the associated file before seeking to the end of its file header.

Parameters

ldPointer	Points to the **LDFILE** structure that was returned as the result of a successful call to **ldopen** or **ldaopen** subroutine.

Return Values

The **ldohseek** subroutine returns a SUCCESS or FAILURE value.

Error Codes

The **ldohseek** subroutine fails if the object file has no optional header, if the file is not a 32–bit or 64–bit object file, or if it cannot seek to the optional header.

Implementation Specifics

This subroutine is part of Base Operating System (BOS) Runtime.

Related Information

The **ldlseek** or **ldnlseek** subroutine, **ldrseek** or **ldnrseek** subroutine, **ldsseek** or **ldnsseek** subroutine, **ldtbseek** subroutine.

Subroutines Overview in *AIX General Programming Concepts : Writing and Debugging Programs*.

ldopen or ldaopen Subroutine

Purpose

Opens an object or archive file for reading.

Library

Object File Access Routine Library (**libld.a**)

Syntax

```
#include <stdio.h>
#include <ldfcn.h>

LDFILE *ldopen(FileName, ldPointer)
char *FileName;
LDFILE *ldPointer;

LDFILE *ldaopen(FileName, ldPointer)
char *FileName;
LDFILE *ldPointer;

LDFILE *lddopen(FileDescriptor, type, ldPointer)
int FileDescriptor;
char *type;
LDFILE *ldPointer;
```

Description

The **ldopen** and **ldclose** subroutines provide uniform access to both simple object files and object files that are members of archive files. Thus, an archive of object files can be processed as if it were a series of ordinary object files.

If the *ldPointer* is null, the **ldopen** subroutine opens the file named by the *FileName* parameter and allocates and initializes an **LDFILE** structure, and returns a pointer to the structure.

If the *ldPointer* parameter is not null and refers to an **LDFILE** for an archive, the structure is updated for reading the next archive member. In this case, and if the value of the **TYPE(***ldPointer***)** macro is the archive magic number **ARTYPE**.

The **ldopen** and **ldclose** subroutines are designed to work in concert. The **ldclose** subroutine returns failure only when the *ldPointer* refers to an archive containing additional members. Only then should the **ldopen** subroutine be called with a num–null *ldPointer* argument. In all other cases, in particular whenever a new *FileName* parameter is opened, the **ldopen** subroutine should be called with a null *ldPointer* argument.

If the value of the *ldPointer* parameter is not null, the **ldaopen** subroutine opens the *FileName* parameter again and allocates and initializes a new **LDFILE** structure, copying the **TYPE**, **OFFSET**, and **HEADER** fields from the *ldPointer* parameter. The **ldaopen** subroutine returns a pointer to the new **ldfile** structure. This new pointer is independent of the old pointer, *ldPointer*. The two pointers may be used concurrently to read separate parts of the object file. For example, one pointer may be used to step sequentially through the relocation information, while the other is used to read indexed symbol table entries.

The **lddopen** function accesses the previously opened file referenced by the *FileDescriptor* parameter. In all other respects, it functions the same as the **ldopen** subroutine.

For AIX 4.3.2 and above, the functions transparently open both 32–bit and 64–bit object files, as well as both small format and large format archive files. Once a file or archive is successfully opened, the calling application can examine the **HEADER(***ldPointer***).f_magic** field to check the magic number of the file or archive member associated with *ldPointer*. (This is necessary due to an archive potentially containing members that are not object files.) The magic numbers U802TOCMAGIC and (for AIX 4.3.2 and above) U803XTOCMAGIC are defined in the **ldfcn.h** file. If the value of **TYPE(**l*dPointer***)** is the archive magic number**ARTYPE**, the flags field can be checked for the archive type. Large format archives will have the flag bit **AR_TYPE_BIG** set in **LDFLAGS(***ldPointer***)**. Large format archives are available on AIX 4.3 and later.

Parameters

FileName	Specifies the file name of an object file or archive.
ldPointer	Points to an **LDFILE** structure.
FileDescriptor	Specifies a valid open file descriptor.
type	Points to a character string specifying the mode for the open file. The **fdopen** function is used to open the file.

Error Codes

Both the **ldopen** and **ldaopen** subroutines open the file named by the *FileName* parameter for reading. Both functions return a null value if the *FileName* parameter cannot be opened, or if memory for the **LDFILE** structure cannot be allocated.

A successful open does not ensure that the given file is a common object file or an archived object file.

Examples

The following is an example of code that uses the **ldopen** and **ldclose** subroutines:

```
/* for each FileName to be processed */

 ldPointer = NULL;
 do

 if((ldPointer = ldopen(FileName, ldPointer)) != NULL)

                     /* check magic number */
                     /* process the file */
                ″
                ″
   while(ldclose(ldPointer) == FAILURE );
```

Implementation Specifics

These subroutines are part of Base Operating System (BOS) Runtime.

Related Information

The **ldclose** or **ldaclose** subroutine.

The **fopen, fopen64, freopen, freopen64, or fdopen** subroutine.

Subroutines Overview in *AIX General Programming Concepts : Writing and Debugging Programs*.

ldrseek or ldnrseek Subroutine

Purpose

Seeks to the relocation entries of a section of an XCOFF file.

Library

Object File Access Routine Library (**libld.a**)

Syntax

```
#include <stdio.h>
#include <ldfcn.h>

int ldrseek (ldPointer, SectionIndex)
ldfile *ldPointer;
unsigned short SectionIndex;

int ldnrseek (ldPointer, SectionName)
ldfile *ldPointer;
char *SectionName;
```

Description

The **ldrseek** subroutine seeks to the relocation entries of the section specified by the *SectionIndex* parameter of the common object file currently associated with the *ldPointer* parameter.

The **ldnrseek** subroutine seeks to the relocation entries of the section specified by the *SectionName* parameter.

For AIX 4.3.2 and above, both subroutines determine the object mode of the associated file before seeking to the relocation entries of the indicated section.

Parameters

ldPointer	Points to an **LDFILE** structure that was returned as the result of a successful call to the **ldopen**, **lddopen**, or **ldaopen** subroutines.
SectionIndex	Specifies an index for the section whose relocation entries are to be sought.
SectionName	Specifies the name of the section whose relocation entries are to be sought.

Return Values

The **ldrseek** and **ldnrseek** subroutines return a SUCCESS or FAILURE value.

Error Codes

The **ldrseek** subroutine fails if the contents of the *SectionIndex* parameter are greater than the number of sections in the object file. The **ldnrseek** subroutine fails if there is no section name corresponding with the *SectionName* parameter. Either function fails if the specified section has no relocation entries or if it cannot seek to the specified relocation entries.

Note: The first section has an index of 1.

Implementation Specifics

These subroutines are part of Base Operating System (BOS) Runtime.

Related Information

The **ldohseek** subroutine, **ldlseek** or **ldnlseek** subroutine, **ldsseek** or **ldnsseek** subroutine, **ldtbseek** subroutine.

Subroutines Overview in *AIX General Programming Concepts : Writing and Debugging Programs*.

ldshread or ldnshread Subroutine

Purpose

Reads a section header of an XCOFF file.

Library

Object File Access Routine Library (**libld.a**)

Syntax

```
#include <stdio.h>
#include <ldfcn.h>

int ldshread (ldPointer, SectionIndex, SectionHead)
LDFILE *ldPointer;
unsigned short SectionIndex;
void *SectionHead;

int ldnshread (ldPointer, SectionName, SectionHead)
LDFILE *ldPointer;
char *SectionName;
void *SectionHead;
```

Description

The **ldshread** subroutine reads the section header specified by the *SectionIndex* parameter of the common object file currently associated with the *ldPointer* parameter into the area of memory beginning at the location specified by the *SectionHead* parameter.

The **ldnshread** subroutine reads the section header named by the *SectionName* argument into the area of memory beginning at the location specified by the *SectionHead* parameter. It is the responsibility of the calling routine to provide a pointer to a buffer large enough to contain the section header of the associated object file. Since the **ldopen** subroutine provides magic number information (via the **HEADER(***ldPointer* **).f_magic** macro), the calling application can always determine whether the *SectionHead* pointer should refer to a 32–bit **SCNHDR** or 64–bit **SCNHDR_64** structure.

Only the first section header named by the *SectionName* argument is returned by the **ldshread** subroutine.

Parameters

ldPointer	Points to an **LDFILE** structure that was returned as the result of a successful call to the **ldopen**, **lldopen**, or **ldaopen** subroutine.
SectionIndex	Specifies the index of the section header to be read. **Note:** The first section has an index of 1.
SectionHead	Points to a buffer large enough to accept either a 32–bit or a 64–bit **SCNHDR** structure, according to the object mode of the file being read.
SectionName	Specifies the name of the section header to be read.

Return Values

The **ldshread** and **ldnshread** subroutines return a SUCCESS or FAILURE value.

Error Codes

The **ldshread** subroutine fails if the *SectionIndex* parameter is greater than the number of sections in the object file. The **ldnshread** subroutine fails if there is no section with the name specified by the *SectionName* parameter. Either function fails if it cannot read the specified section header.

Examples

The following is an example of code that opens an object file, determines its mode, and uses the **ldnshread** subroutine to acquire the .text section header. This code would be compiled with both **__XCOFF32__** and **__XCOFF64__** defined:

```
#define __XCOFF32__
#define __XCOFF64__

#include <ldfcn.h>



/* for each FileName to be processed */

if ( (ldPointer = ldopen(FileName, ldPointer)) != NULL )
{
    SCNHDR    SectionHead32;
    SCNHDR_64 SectionHead64;
    void      *SectionHeader;

    if ( HEADER(ldPointer).f_magic == U802TOCMAGIC )
        SectionHeader = &SectionHead32;
    else if ( HEADER(ldPointer).f_magic == U803XTOCMAGIC )
        SectionHeader = &SectionHead64;
    else
        SectionHeader = NULL;

    if ( SectionHeader && (ldnshread( ldPointer, ".text",
&SectionHeader ) == SUCCESS) )
    {
        /* ...successfully read header... */
        /* ...process according to magic number... */
    }
}
```

Implementation Specifics

These subroutines are part of Base Operating System (BOS) Runtime.

Related Information

The **ldahread** subroutine, **ldfhread** subroutine, **ldgetname** subroutine, **ldlread**, **ldlinit**, or **ldlitem** subroutine, **ldtbread** subroutine.

Subroutines Overview in *AIX General Programming Concepts : Writing and Debugging Programs*.

ldsseek or ldnsseek Subroutine

Purpose

Seeks to an indexed or named section of a common object file.

Library

Object File Access Routine Library (**libld.a**)

Syntax

```
#include <stdio.h>
#include <ldfcn.h>

int ldsseek (ldPointer, SectionIndex)
LDFILE *ldPointer;
unsigned short SectionIndex;

int ldnsseek (ldPointer, SectionName)
LDFILE *ldPointer;
char *SectionName;
```

Description

The **ldsseek** subroutine seeks to the section specified by the *SectionIndex* parameter of the common object file currently associated with the *ldPointer* parameter. The subroutine determines the object mode of the associated file before seeking to the indicated section.

The **ldnsseek** subroutine seeks to the section specified by the *SectionName* parameter.

Parameters

ldPointer	Points to the **LDFILE** structure that was returned as the result of a successful call to the **ldopen** or **ldaopen** subroutine.
SectionIndex	Specifies the index of the section whose line number entries are to be seeked to.
SectionName	Specifies the name of the section whose line number entries are to be seeked to.

Return Values

The **ldsseek** and **ldnsseek** subroutines return a SUCCESS or FAILURE value.

Error Codes

The **ldsseek** subroutine fails if the *SectionIndex* parameter is greater than the number of sections in the object file. The **ldnsseek** subroutine fails if there is no section name corresponding with the *SectionName* parameter*.* Either function fails if there is no section data for the specified section or if it cannot seek to the specified section.

Note: The first section has an index of 1.

Implementation Specifics

These subroutines are part of Base Operating System (BOS) Runtime.

Related Information

The **ldlseek** or **ldnlseek** subroutine, **ldohseek** subroutine, **ldrseek** or **ldnrseek** subroutine, **ldtbseek** subroutine.

Subroutines Overview in *AIX General Programming Concepts : Writing and Debugging Programs*.

ldtbindex Subroutine

Purpose

Computes the index of a symbol table entry of a common object file.

Library

Object File Access Routine Library (**libld.a**)

Syntax

```
#include <stdio.h>
#include <ldfcn.h>

long ldtbindex (ldPointer)
LDFILE *ldPointer;
```

Description

The **ldtbindex** subroutine returns the index of the symbol table entry at the current position of the common object file associated with the *ldPointer* parameter.

The index returned by the **ldtbindex** subroutine may be used in subsequent calls to the **ldtbread** subroutine. However, since the **ldtbindex** subroutine returns the index of the symbol table entry that begins at the current position of the object file, if the **ldtbindex** subroutine is called immediately after a particular symbol table entry has been read, it returns the index of the next entry.

Parameters

ldPointer	Points to the **LDFILE** structure that was returned as a result of a successful call to the **ldopen** or **ldaopen** subroutine.

Return Values

The **ldtbindex** subroutine returns the value BADINDEX upon failure. Otherwise a value greater than or equal to zero is returned.

Error Codes

The **ldtbindex** subroutine fails if there are no symbols in the object file or if the object file is not positioned at the beginning of a symbol table entry.

Note: The first symbol in the symbol table has an index of 0.

Implementation Specifics

This subroutine is part of Base Operating System (BOS) Runtime.

Related Information

The **ldtbread** subroutine, **ldtbseek** subroutine.

Subroutines Overview in *AIX General Programming Concepts : Writing and Debugging Programs*.

ldtbread Subroutine

Purpose

Reads an indexed symbol table entry of a common object file.

Library

Object File Access Routine Library (**libld.a**)

Syntax

```
#include <stdio.h>
#include <ldfcn.h>

int ldtbread (ldPointer, SymbolIndex, Symbol)
LDFILE *ldPointer;
long SymbolIndex;
void *Symbol;
```

Description

The **ldtbread** subroutine reads the symbol table entry specified by the *SymbolIndex* parameter of the common object file currently associated with the *ldPointer* parameter into the area of memory beginning at the *Symbol* parameter. It is the responsibility of the calling routine to provide a pointer to a buffer large enough to contain the symbol table entry of the associated object file. Since the ldopen subroutine provides magic number information (via the **HEADER(***ldPointer***).f_magic** macro), the calling application can always determine whether the *Symbol* pointer should refer to a 32–bit **SYMENT** or 64–bit **SYMENT_64** structure.

Parameters

ldPointer	Points to the **LDFILE** structure that was returned as the result of a successful call to the **ldopen** or **ldaopen** subroutine.
SymbolIndex	Specifies the index of the symbol table entry to be read.
Symbol	Points to a either a 32–bit or a 64–bit **SYMENT** structure.

Return Values

The **ldtbread** subroutine returns a SUCCESS or FAILURE value.

Error Codes

The **ldtbread** subroutine fails if the *SymbolIndex* parameter is greater than or equal to the number of symbols in the object file, or if it cannot read the specified symbol table entry.

Note: The first symbol in the symbol table has an index of 0.

Implementation Specifics

This subroutine is part of Base Operating System (BOS) Runtime.

Related Information

The **ldahread** subroutine, **ldfhread** subroutine, **ldgetname** subroutine, **ldlread**, **ldlinit**, or **ldlitem** subroutine, **ldshread** or **ldnshread** subroutine.

Subroutines Overview in *AIX General Programming Concepts : Writing and Debugging Programs*.

ldtbseek Subroutine

Purpose

Seeks to the symbol table of a common object file.

Library

Object File Access Routine Library (**libld.a**)

Syntax

```
#include <stdio.h>
#include <ldfcn.h>

int ldtbseek (ldPointer)
LDFILE *ldPointer;
```

Description

The **ldtbseek** subroutine seeks to the symbol table of the common object file currently associated with the *ldPointer* parameter.

Parameters

ldPointer	Points to the **LDFILE** structure that was returned as the result of a successful call to the **ldopen** or **ldaopen** subroutine.

Return Values

The **ldtbseek** subroutine returns a SUCCESS or FAILURE value.

Error Codes

The **ldtbseek** subroutine fails if the symbol table has been stripped from the object file or if the subroutine cannot seek to the symbol table.

Implementation Specifics

This subroutine is part of Base Operating System (BOS) Runtime.

Related Information

The **ldlseek** or **ldnlseek** subroutine, **ldohseek** subroutine, **ldrseek** or **ldnrseek** subroutine, **ldsseek** or **ldnsseek** subroutine.

Subroutines Overview in *AIX General Programming Concepts : Writing and Debugging Programs*.

lgamma, lgammal, or gamma Subroutine

Purpose

Computes the natural logarithm of the gamma function.

Libraries

lgamma, **lgammal**, and **gamma**:
IEEE Math Library (**libm.a**)
or System V Math Library (**libmsaa.a**)

Syntax

```
#include <math.h>

extern int signgam;

double lgamma (x)
double x;

long double lgammal (x)
long double x;


double gamma (x)
double x;
```

Description

The subroutine names **lgamma** and **gamma** are different names for the same function. The **lgammal** subroutine provides the same function for numbers in long double format.

The **lgamma** subroutine returns the natural logarithm of the absolute value of the gamma function of the *x* parameter.

```
G(x) = integral [0 to INF] of ((e**(-t) * t**(x-1) dt)
```

The sign of **lgamma** of *x* is stored in the external integer variable **signgam**. The *x* parameter may not be a non–positive integer.

Do not use the expression:

```
g = exp(lgamma(x)) * signgam
```

to compute `g = G(x)`. Instead, use a sequence such as:

```
lg = lgamma(x);
g = exp(lg) * signgam;
```

Note: Compile any routine that uses subroutines from the **libm.a** with the **–lm** flag. To compile the **lgamma.c** file, enter:

```
cc lgamma.c -lm
```

Parameters

x For the **lgamma** and **gamma** subroutines, specifies a double–precision floating–point value. For the **lgammal** subroutine, specifies a long double–precision floating–point value.

Error Codes

- For non–positive integer arguments, the **lgamma** and **lgammal** subroutines return NaNQ and set the division–by–zero bit in the floating–point exception status.

- If the correct value overflows, the **lgamma** and **lgammal** subroutines return a **HUGE_VAL** value. If the correct value underflows, the **lgamma** and **lgammal** subroutines return 0.

When using the **libmsaa.a** library with the **–lmsaa** flag:

- For non–positive integer arguments, the **lgamma** subroutine returns a **HUGE_VAL** value and set the **errno** global variable to a **EDOM** value. A message indicating SING error is printed on the standard error output.
- If the correct value overflows, the **lgamma** and **lgammal** subroutines and **lgammal** subroutine return a **HUGE_VAL** value and sets the **errno** global variable to a **ERANGE** value.

Note: These error–handling procedures may be changed with the **matherr** subroutine when using **libmsaa.a** (**–lmsaa**).

Implementation Specifics

These subroutines are part of Base Operating System (BOS) Runtime.

Related Information

The **exp**, **expm1**, **log**, **log10**, **log1p** or **pow** subroutine, **matherr** subroutine.

Subroutines Overview in *AIX General Programming Concepts : Writing and Debugging Programs*.

128–Bit long double Floating–Point Format in *AIX General Programming Concepts : Writing and Debugging Programs*.

lineout Subroutine

Purpose

Formats a print line.

Library

None (provided by the print formatter)

Syntax

```
#include <piostruct.h>

int lineout (fileptr)
FILE *fileptr;
```

Description

The **lineout** subroutine is invoked by the formatter driver only if the **setup** subroutine returns a non–null pointer. This subroutine is invoked for each line of the document being formatted. The **lineout** subroutine reads the input data stream from the *fileptr* parameter. It then formats and outputs the print line until it recognizes a situation that causes vertical movement on the page.

The **lineout** subroutine should process all characters to be printed and all printer commands related to horizontal movement on the page.

The **lineout** subroutine should not output any printer commands that cause vertical movement on the page. Instead, it should update the **vpos** (new vertical position) variable pointed to by the **shars_vars** structure that it shares with the formatter driver to indicate the new vertical position on the page. It should also refresh the **shar_vars** variables for vertical increment and vertical decrement (reverse line–feed) commands.

When the **lineout** subroutine returns, the formatter driver sends the necessary commands to the printer to advance to the new vertical position on the page. This position is specified by the **vpos** variable. The formatter driver automatically handles top and bottom margins, new pages, initial pages to be skipped, and progress reports to the **qdaemon** daemon.

The following conditions can cause vertical movements:

- Line–feed control character or variable line–feed control sequence
- Vertical–tab control character
- Form–feed control character
- Reverse line–feed control character
- A line too long for the printer that wraps to the next line

Other conditions unique to a specific printer also cause vertical movement.

Parameters

fileptr	Specifies a file structure for the input data stream.

Return Values

Upon successful completion, the **lineout** subroutine returns the number of bytes processed from the input data stream. It excludes the end–of–file character and any control characters or escape sequences that result only in vertical movement on the page (for example, line feed or vertical tab).

If a value of 0 is returned and the value in the **vpos** variable pointed to by the **shars_vars** structure has not changed, or there are no more data bytes in the input data stream, the formatter driver assumes that printing is complete.

If the **lineout** subroutine detects an error, it uses the **piomsgout** subroutine to issue an error message. It then invokes the **pioexit** subroutine with a value of PIOEXITBAD.

Note: If either the **piocmdout** or **piogetstr** subroutine detects an error, it automatically issues its own error messages and terminates the print job.

Related Information

The **piocmdout** subroutine, **pioexit** subroutine, **piogetstr** subroutine, **piomsgout** subroutine, **setup** subroutine.

Adding a New Printer Type to Your System and Printer Addition Management Subsystem: Programming Overview in *AIX Kernel Extensions and Device Support Programming Concepts*.

Example of Print Formatter in *AIX General Programming Concepts : Writing and Debugging Programs*.

link Subroutine

Purpose

Creates an additional directory entry for an existing file.

Library

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

Syntax

```
#include <unistd.h>

int link (Path1,
Path2)
const char *Path1, *Path2;
```

Description

The **link** subroutine creates an additional hard link (directory entry) for an existing file. Both the old and the new links share equal access rights to the underlying object.

Parameters

Path1	Points to the path name of an existing file.
Path2	Points to the path name of the directory entry to be created.

Notes:

1. If Network File System (NFS) is installed on your system, these paths can cross into another node.
2. With hard links, both the *Path1* and *Path2* parameters must reside on the same file system. If *Path1* is a symbolic link, an error is returned. Creating links to directories requires root user authority.

Return Values

Upon successful completion, the **link** subroutine returns a value of 0. Otherwise, a value of –1 is returned, and the **errno** global variable is set to indicate the error.

Error Codes

The **link** subroutine is unsuccessful if one of the following is true:

EACCES	Indicates the requested link requires writing in a directory that denies write permission.
EDQUOT	Indicates the directory in which the entry for the new link is being placed cannot be extended, or disk blocks could not be allocated for the link because the user or group quota of disk blocks or i–nodes on the file system containing the directory has been exhausted.
EEXIST	Indicates the link named by the *Path2* parameter already exists.
EMLINK	Indicates the file already has the maximum number of links.
ENOENT	Indicates the file named by the *Path1* parameter does not exist.
ENOSPC	Indicates the directory in which the entry for the new link is being placed cannot be extended because there is no space left on the file system containing the directory.

EPERM Indicates the file named by the *Path1* parameter is a directory, and the calling process does not have root user authority.

EROFS Indicates the requested link requires writing in a directory on a read–only file system.

EXDEV Indicates the link named by the *Path2* parameter and the file named by the *Path1* parameter are on different file systems, or the file named by *Path1* refers to a named STREAM.

The **link** subroutine can be unsuccessful for other reasons. See ”Base Operating System Error Codes For Services That Require Path–Name Resolution” for a list of additional errors.

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

ETIMEDOUT Indicates the connection timed out.

Implementation Specifics

This subroutine is part of Base Operating System (BOS) Runtime.

Related Information

The **symlink** subroutine, **unlink** subroutine.

The **link** or **unlink** command, **ln** command, **rm** command.

Files, Directories, and File Systems for Programmers in *AIX General Programming Concepts : Writing and Debugging Programs*.

lio_listio or lio_listio64 Subroutine

Purpose

Initiates a list of asynchronous I/O requests with a single call.

Syntax

```
#include <aio.h>

int lio_listio (cmd,
list, nent, eventp)
int cmd, nent;
struct liocb *list[ ];
struct event *eventp;


int lio_listio64
(cmd, list,nent, eventp)
int cmd, nent; struct liocb64 *list;
struct event *eventp;
```

Description

The **lio_listio** subroutine allows the calling process to initiate the *nent* parameter asynchronous I/O requests. These requests are specified in the **liocb** structures pointed to by the elements of the *list* array. The call may block or return immediately depending on the *cmd* parameter. If the *cmd* parameter requests that I/O completion be asynchronously notified, a **SIGIO** signal is delivered when all I/O operations are completed.

The **lio_listio64** subroutine is similar to the **lio_listio** subroutine except that it takes an array of pointers to **liocb64** structures. This allows the **lio_listio64** subroutine to specify offsets in excess of OFF_MAX (2 gigbytes minus 1).

In the large file enabled programming environment, **lio_listio** is redefined to be **lio_listio64**.

Note: The **SIGIO** signal will be replaced by real–time signals when they are available. The pointer to the **event** structure *eventp* parameter is currently not in use but is included for future compatibility.

Parameters

cmd
The *cmd* parameter takes one of the following values:

LIO_WAIT
Queues the requests and waits until they are complete before returning.

LIO_NOWAIT
Queues the requests and returns immediately, without waiting for them to complete. The *event* parameter is ignored.

LIO_ASYNC
Queues the requests and returns immediately, without waiting for them to complete. An enhanced signal is delivered when all the operations are completed. Currently this command is not implemented.

LIO_ASIG
Queues the requests and returns immediately, without waiting for them to complete. A **SIGIO** signal is generated when all the I/O operations are completed.

list
Points to an array of pointers to **liocb** structures. The structure array contains *nent* elements:

lio_aiocb
The asynchronous I/O control block associated with this I/O request. This is an actual **aiocb** structure, not a pointer to one.

lio_fildes
Identifies the file object on which the I/O is to be performed.

lio_opcode
This field may have one of the following values defined in the **/usr/include/sys/aio.h** file:

LIO_READ
Indicates that the read I/O operation is requested.

LIO_WRITE
Indicates that the write I/O operation is requested.

LIO_NOP
Specifies that no I/O is requested (that is, this element will be ignored).

nent
Specifies the number of entries in the array of pointers to **listio** structures.

eventp
Points to an **event** structure to be used when the *cmd* parameter is set to the **LIO_ASYNC** value. This parameter is currently ignored.

Execution Environment The lio_listio and lio_listio64 subroutines can be called from the process environment only.

Return Values

When the **lio_listio** subroutine is successful, it returns a value of 0. Otherwise, it returns a value of –1 and sets the **errno** global variable to identify the error. The returned value indicates the success or failure of the **lio_listio** subroutine itself and not of the asynchronous I/O requests (except when the command is **LIO_WAIT**). The **aio_error** subroutine returns the status of each I/O request.

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

EAGAIN
Indicates that the system resources required to queue the request are not available. Specifically, the transmit queue may be full, or the maximum number of opens may have been reached.

EFAIL
Indicates that one or more I/O operations was not successful. This error can be received only if the *cmd* parameter has a **LIO_WAIT** value.

EINTR Indicates that a signal or event interrupted the **lio_listio** subroutine call.

EINVAL Indicates that the `aio_whence` field does not have a valid value or that the resulting pointer is not valid.

Implementation Specifics

The lio_listio and lio_listio64 subroutines are part of the Base Operating System (BOS) Runtime.

Related Information

The **aio_cancel** or **aio_cancel64** subroutine, **aio_error** or **aio_error64** subroutine, **aio_read** or **aio_read64** subroutine, **aio_return** or **aio_return64** subroutine, **aio_suspend** or **aio_suspend64** subroutine, **aio_write** or **aio_write64** subroutine.

The Asynchronous I/O Overview and the Communications I/O Subsystem: Programming Introduction in *AIX General Programming Concepts : Writing and Debugging Programs*.

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

load Subroutine

Purpose

Loads and binds an object module into the current process.

Syntax

```
int *load (FilePath, Flags, LibraryPath)
char *FilePath;
uint Flags;
char *LibraryPath;
```

Description

The **load** subroutine loads the specified module into the calling process's address space. A module is an object file that may be a member of an archive. Unlike the **exec** subroutine, the **load** subroutine does not replace the current program with a new one. Instead, it loads the new module into the process private segment at the current break value and the break value is updated to point past the new module.

The **exec** subroutine is similar to the **load** subroutine, except that the **exec** subroutine does not have an explicit library path parameter; it has only the **LIBPATH** environment variable. Also, the **LIBPATH** variable is ignored when the program using the **exec** subroutine has more privilege than the caller, for example, in the case of an **suid** program.

If the calling process later uses the **unload** subroutine to unload the object file, the space is unusable by the process except through another call to the **load** subroutine. If the kernel finds an unused space created by a previous unload, it reuses this space rather than loading the new module at the break value. Space for loaded programs is managed by the kernel and not by any user–level storage–management routine.

A large application can be split up into one or more module s in one of two ways that allow execution within the same process. The first way is to create each of the application's modules separately and use **load** to explicitly load a module when it is needed. The other way is to specify the relationship between the modules when they are created by defining imported and exported symbols.

Modules can import symbols from other modules. Whenever symbols are imported from one or more other modules, these modules are automatically loaded to resolve the symbol references if the required modules are not already loaded, and if the imported symbols are not specified as deferred imports .These modules can be archive members in libraries or separate object files and can have either shared or private object file characteristics that control how and where they are loaded.

Shared modules (typically members of a shared library archive) are loaded into the shared library region, when their access permissions allow sharing, that is, when they have read–other permission . Shared modules without the required permissions for sharing and private modules are loaded into the process private region.

When the loader resolves a symbol, it uses the file name recorded with that symbol to find the module that exports the symbol. If the file name contains any / (slash) characters, it is used directly and must name an appropriate object file (or archive). However, if the file name is a base name (contains no / characters), the loader searches the directories specified in the default library path for an object file (or archive) with that base name.

The *LibraryPath* is a string containing one or more directory path names separated by colon s . If the base name is not found, the search continues, using the library path specified in the object file containing the symbol being resolved (normally the library path specified to the **ld** command that created the object file). The first instance of the base name found is used. An error occurs if this module cannot be loaded or does not export a definition of the symbol being resolved.

The default library path may be specified using the *LibraryPath* parameter. If not explicitly set, the default library path may be obtained from the **LIBPATH** environment variable or from the module specified by the *FilePath* parameter. If the **L_LIBPATH_EXEC** flag is specified, then the library path used at process exec time is prepended to any other library path specified in the load call.

(This paragraph only applies to AIX 4.3.1 and previous releases.) When a process is executing under **ptrace** control, portions of the process's address space are recopied after the **load** processing completes. For a 32–bit process, the main program text (loaded in segment 1) and shared library modules (loaded in segment 13) are recopied. Any breakpoints or other modifications to these segments must be reinserted after the **load** call. For a 64–bit process, shared library modules are recopied after a **load** call. The debugger will be notified by setting the **W_SLWTED** flag in the status returned by **wait**, so that it can reinsert breakpoints.

(This paragraph only applies to AIX 4.3.2 and later releases.) When a process executing under **ptrace** control calls **load**, the debugger is notified by setting the **W_SLWTED** flag in the status returned by **wait**. Any modules newly loaded into the shared library segments will be copied to the process's private copy of these segments, so that they can be examined or modified by the debugger.

If the program calling the **load** subroutine was linked on 4.2 or a later release, the **load** subroutine will call initialization routines (**init** routines) for the new module and any of its dependents if they were not already loaded.

Modules loaded by this subroutine are automatically unloaded when the process terminates or when the **exec** subroutine is executed. They are explicitly unloaded by calling the **unload** subroutine.

Parameters

FilePath — Points to the name of the object file to be loaded. If the *FilePath* name contains no / (slash) symbols, it is treated as a base name, and should be in one of the directories listed in the library path.

The library path is either the value of the *LibraryPath* parameter if not a null value, or the value of the **LIBPATH** environment variable (if set) or the library path used at process exec time (if the **L_LIBPATH_EXEC** is set). If no library path is provided, the object file should be in the current directory.

If the *FilePath* parameter is not a base name (if it contains at least one / character), the name is used as it is, and no library path searches are performed to locate the object file. However, the library path is used to locate dependent modules.

Flags — Modifies the behavior of the **load** service as follows (see the **ldr.h** file). If no special behavior is required, set the value of the flags parameter to 0 (zero) . For compatibility, a value of 1 (one) may also be specified.

L_LIBPATH_EXEC
Specifies that the library path used at process exec time should be prepended to any library path specified in the **load** call (either as an argument or environment variable). It is recommended that this flag be specified in all calls to the **load** subroutine.

L_NOAUTODEFER
Specifies that any deferred imports must be explicitly resolved by use of the **loadbind** subroutine. This allows unresolved imports to be explicitly resolved at a later time with a specified module. If this flag is not specified, deferred imports (marked for deferred resolution) are resolved at the earliest opportunity when any module is loaded that has exported symbols matching unresolved imports.

LibraryPath — Points to a character string that specifies the default library search path.

If the *LibraryPath* parameter is a null value and the **LIBPATH** environment variable is set, the **LIBPATH** value is used as the default load path. If neither default library path option is provided, the library path specified in the loader section of the object file specified in the *FilePath* parameter is used as the default library path. If the **L_LIBPATH_EXEC** flag is specified, then the library path used at process exec time is prepended to the above specified default library path.

Note the difference between setting the *LibraryPath* parameter to null, and having the *LibraryPath* parameter point to a null string (" "). A null string is a valid library path which consists of a single directory: the current directory.

If the module is not in the *LibraryPath* parameter or the LIBPATH environmental variable (if the *LibraryPath* parameter was null), then the library path specified in the loader section of the module importing the symbol is used to locate the module exporting the required symbol. The library path in the importing module was specified when the module was link–edited (by the **ld** command).

The library path search is not performed when either a relative or an absolute path name is specified for the module exporting the symbol.

Return Values

Upon successful completion, the **load** subroutine returns the pointer to function for the entry point of the module. If the module has no entry point, the address of the data section of the module is returned.

Error Codes

If the **load** subroutine fails, a null pointer is returned, the module is not loaded, and **errno** global variable is set to indicate the error. The **load** subroutine fails if one or more of the following are true of a module to be explicitly or automatically loaded:

EACCES	Indicates the file is not an ordinary file, or the mode of the program file denies execution permission, or search permission is denied on a component of the path prefix.
EINVAL	Indicates the file or archive member has a valid magic number in its header, but the header is damaged or is incorrect for the machine on which the file is to be run.
ELOOP	Indicates too many symbolic links were encountered in translating the path name.
ENOEXEC	Indicates an error occurred when loading or resolving symbols for the specified module. This can be due to an attempt to load a module with an invalid **XCOFF** header, a failure to resolve symbols that were not defined as deferred imports or several other load time related problems. The **loadquery** subroutine can be used to return more information about the load failure. If the main program was linked on a 4.2 or later system, and if runtime linking is used, the **load** subroutine will fail if the runtime linker could not resolve some symbols. In this case, **errno** will be set to **ENOEXEC** , but the **loadquery** subroutine will not return any additional information.
ENOMEM	Indicates the program requires more memory than is allowed by the system–imposed maximum.
ETXTBSY	Indicates the file is currently open for writing by some process.
ENAMETOOLONG	Indicates a component of a path name exceeded 255 characters, or an entire path name exceeded 1023 characters.
ENOENT	Indicates a component of the path prefix does not exist, or the path name is a null value.
ENOTDIR	Indicates a component of the path prefix is not a directory.
ESTALE	Indicates the process root or current directory is located in a virtual file system that has been unmounted.

Implementation Specifics

This subroutine is part of Base Operating System (BOS) Runtime.

Related Information

The **dlopen** subroutine, **exec** subroutine, **loadbind** subroutine, **loadquery** subroutine, **ptrace** subroutine, **unload** subroutine.

The **ld** command.

The Shared Library Overview and Subroutines Overview in *AIX General Programming Concepts : Writing and Debugging Programs*.

loadbind Subroutine

Purpose

Provides specific run–time resolution of a module's deferred symbols.

Syntax

```
int loadbind(Flag, ExportPointer, ImportPointer)
int Flag;
void *ExportPointer, *ImportPointer;
```

Description

The **loadbind** subroutine controls the run–time resolution of a previously loaded object module's unresolved imported symbols.

The **loadbind** subroutine is used when two modules are loaded. Module A, an object module loaded at run time with the **load** subroutine, has designated that some of its imported symbols be resolved at a later time. Module B contains exported symbols to resolve module A's unresolved imports.

To keep module A's imported symbols from being resolved until the **loadbind** service is called, you can specify the **load** subroutine flag, **L_NOAUTODEFER**, when loading module A.

(This paragraph only applies to AIX 4.3.1 and previous releases.) When a 32–bit process is executing under **ptrace** control, portions of the process's address space are recopied after the **loadbind** processing completes. The main program text (loaded in segment 1) and shared library modules (loaded in segment 13) are recopied. Any breakpoints or other modifications to these segments must be reinserted after the **loadbind** call.

(This paragraph only applies to AIX 4.3.2 and later releases.) When a 32–bit process executing under **ptrace** control calls **loadbind**, the debugger is notified by setting the **W_SLWTED** flag in the status returned by **wait**.

When a 64–bit process under **ptrace** control calls **loadbind**, the debugger is not notified and execution of the process being debugged continues normally.

Parameters

Flag	Currently not used.
ExportPointer	Specifies the function pointer returned by the **load** subroutine when module B was loaded.
ImportPointer	Specifies the function pointer returned by the **load** subroutine when module A was loaded.

Note: The *ImportPointer* or *ExportPointer* parameter may also be set to any exported static data area symbol or function pointer contained in the associated module. This would typically be the function pointer returned from the **load** of the specified module.

Return Values

A 0 is returned if the **loadbind** subroutine is successful.

Error Codes

A –1 is returned if an error is detected, with the **errno** global variable set to an associated error code:

EINVAL	Indicates that either the *ImportPointer* or *ExportPointer* parameter is not valid (the pointer to the *ExportPointer* or *ImportPointer* parameter does not correspond to a loaded program module or library).
ENOMEM	Indicates that the program requires more memory than allowed by the system–imposed maximum.

After an error is returned by the **loadbind** subroutine, you may also use the **loadquery** subroutine to obtain additional information about the **loadbind** error.

Implementation Specifics

This subroutine is part of Base Operating System (BOS) Runtime.

Related Information

The **load** subroutine, **loadquery** subroutine, **unload** subroutine.

The **ld** command.

Subroutines Overview in *AIX General Programming Concepts : Writing and Debugging Programs*.

loadquery Subroutine

Purpose

Returns error information from the **load** or **exec** subroutine; also provides a list of object files loaded for the current process.

Syntax

```
int loadquery(Flags, Buffer, BufferLength)
int Flags;
void *Buffer;
unsigned int BufferLength;
```

Description

The **loadquery** subroutine obtains detailed information about an error reported on the last **load** or **exec** subroutine executed by a calling process. The **loadquery** subroutine may also be used to obtain a list of object file names for all object files that have been loaded for the current process, or the library path that was used at process exec time.

Parameters

Buffer Points to a *Buffer* in which to store the information.

BufferLength Specifies the number of bytes available in the *Buffer* parameter.

Flags Specifies the action of the **loadquery** subroutine as follows:

L_GETINFO Returns a list of all object files loaded for the current process, and stores the list in the *Buffer* parameter. The object file information is contained in a sequence of **LD_INFO** structures as defined in the **sys/ldr.h** file. Each structure contains the module location in virtual memory and the path name that was used to load it into memory. The file descriptor field in the **LD_INFO** structure is not filled in by this function.

L_GETMESSAGE Returns detailed error information describing the failure of a previously invoked **load** or **exec** function, and stores the error message information in *Buffer.* Upon successful return from this function the beginning of the *Buffer* contains an array of character pointers. Each character pointer points to a string in the buffer containing a loader error message. The character array ends with a null character pointer. Each error message string consists of an ASCII message number followed by zero or more characters of error–specific message data. Valid message numbers are listed in the **sys/ldr.h** file.

You can format the error messages returned by the **L_GETMESSAGE** function and write them to standard error using the standard system command /**usr**/**sbin**/**execerror** as follows:

```
char *buffer[1024];
buffer[0] = ”execerror”;
buffer[1] =”name of program that failed\  to
load”;
loadquery(L_GETMESSAGES, &buffer[2],\
  sizeof buffer -8);
execvp(”/usr/sbin/execerror”,buffer);
```

This sample code causes the application to terminate after the messages are written to standard error.

L_GETLIBPATH Returns the library path that was used at process exec time. The library path is a null terminated character string.

Return Values

Upon successful completion, **loadquery** returns the requested information in the caller's buffer specified by the *Buffer* and *BufferLength* parameters.

Error Codes

The **loadquery** subroutine returns with a return code of –1 and the **errno** global variable is set to one of the following when an error condition is detected:

ENOMEM	Indicates that the caller's buffer specified by the *Buffer* and *BufferLength* parameters is too small to return the information requested. When this occurs, the information in the buffer is undefined.
EINVAL	Indicates the function specified in the *Flags* parameter is not valid.
EFAULT	Indicates the address specified in the *Buffer* parameter is not valid.

Implementation Specifics

This subroutine is part of Base Operating System (BOS) Runtime.

Related Information

The **exec** subroutine, **load** subroutine, **loadbind** subroutine, **unload** subroutine.

The **ld** command.

Subroutines Overview in *AIX General Programming Concepts : Writing and Debugging Programs*.

localeconv Subroutine

Purpose

Sets the locale–dependent conventions of an object.

Library

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

Syntax

```
#include <locale.h>

struct lconv *localeconv ( )
```

Description

The **localeconv** subroutine sets the components of an object using the **lconv** structure. The **lconv** structure contains values appropriate for the formatting of numeric quantities (monetary and otherwise) according to the rules of the current locale.

The fields of the structure with the type **char** * are strings, any of which (except `decimal_point`) can point to a null string, which indicates that the value is not available in the current locale or is of zero length. The fields with type **char** are nonnegative numbers, any of which can be the **CHAR_MAX** value which indicates that the value is not available in the current locale. The fields of the **lconv** structure include the following:

`char *decimal_point` — The decimal–point character used to format non–monetary quantities.

`char *thousands_sep` — The character used to separate groups of digits to the left of the decimal point in formatted non–monetary quantities.

`char *grouping` — A string whose elements indicate the size of each group of digits in formatted non–monetary quantities.

The value of the `grouping` field is interpreted according to the following:

- **CHAR_MAX** — No further grouping is to be performed.
- **0** — The previous element is to be repeatedly used for the remainder of the digits.
- **other** — The value is the number of digits that comprise the current group. The next element is examined to determine the size of the next group of digits to the left of the current group.

`char *int_curr_symbol` — The international currency symbol applicable to the current locale, left–justified within a four–character space–padded field. The character sequences are in accordance with those specified in ISO 4217, ”Codes for the Representation of Currency and Funds.”

`char *currency_symbol` — The local currency symbol applicable to the current locale.

`char *mon_decimal_point` — The decimal point used to format monetary quantities.

`char *mon_thousands_sep` — The separator for groups of digits to the left of the decimal point in formatted monetary quantities.

char *mon_grouping	A string whose elements indicate the size of each group of digits in formatted monetary quantities. The value of the mon_grouping field is interpreted according to the following: **CHAR_MAX** No further grouping is to be performed. **0** The previous element is to be repeatedly used for the remainder of the digits. **other** The value is the number of digits that comprise the current group. The next element is examined to determine the size of the next group of digits to the left of the current group.
char *positive_sign	The string used to indicate a nonnegative formatted monetary quantity.
char *negative_sign	The string used to indicate a negative formatted monetary quantity.
char int_frac_digits	The number of fractional digits (those to the right of the decimal point) to be displayed in a formatted monetary quantity.
char p_cs_precedes	Set to 1 if the specified currency symbol (the currency_symbol or int_curr_symbol field) precedes the value for a nonnegative formatted monetary quantity and set to 0 if the specified currency symbol follows the value for a nonnegative formatted monetary quantity.
char p_sep_by_space	Set to 1 if the currency_symbol or int_curr_symbol field is separated by a space from the value for a nonnegative formatted monetary quantity and set to 0 if the currency_symbol or int_curr_symbol field is not separated by a space from the value for a nonnegative formatted monetary quantity.
char n_cs_precedes	Set to 1 if the currency_symbol or int_curr_symbol field precedes the value for a negative formatted monetary quantity and set to 0 if the currency_symbol or int_curr_symbol field follows the value for a negative formatted monetary quantity.
char n_sep_by_space	Set to 1 if the currency_symbol or int_curr_symbol field is separated by a space from the value for a negative formatted monetary quantity and set to 0 if the currency_symbol or int_curr_symbol field is not separated by a space from the value for a negative formatted monetary quantity. Set to 2 if the symbol and the sign string are adjacent and separated by a blank character.

`char p_sign_posn`
: Set to a value indicating the positioning of the positive sign (the `positive_sign` fields) for nonnegative formatted monetary quantity.

`char n_sign_posn`
: Set to a value indicating the positioning of the negative sign (the `negative_sign` fields) for a negative formatted monetary quantity.

The values of the `p_sign_posn` and `n_sign_posn` fields are interpreted according to the following definitions:

0 Parentheses surround the quantity and the specified currency symbol or international currency symbol.

1 The sign string precedes the quantity and the currency symbol or international currency symbol.

2 The sign string follows the quantity and currency symbol or international currency symbol.

3 The sign string immediately precedes the currency symbol or international currency symbol.

4 The sign string immediately follows the currency symbol or international currency symbol.

The following table illustrates the rules that can be used by three countries to format monetary quantities:

Country	**Positive Format**	**Negative Format**	**International Format**
Italy	L.1234	–L.1234	ITL.1234
Norway	krl.234.56	krl.234.56–	NOK 1.234.56
Switzerland	SFrs.1.234.56	SFrs.1.234.56C	CHF 1.234.56

The following table shows the values of the monetary members of the structure returned by the **localeconv** subroutine for these countries:

struct localeconv	**Italy**	**Norway**	**Switzerland**
char *in_curr_symbol	”ITL.”	”NOK”	”CHF”
char *currency_symbol	”L.”	”kr”	”SFrs.”
char *mon_decimal_point	” ”	”.”	”.”
char *mon_thousands_sep	”.”	”.”	”.”
char *mon_grouping	”\3”	”\3”	”\3”
char *positive_sign	” ”	” ”	” ”
char *negative_sign	”–”	”–”	”C”
char int_frac_digits	0	2	2
char frac_digits	0	2	2
char p_cs_precedes	1	1	1
char p_sep_by_space	0	0	0

char n_cs_precedes	1	1	1
char n_sep_by_space	0	0	0
char p_sign_posn	1	1	1
char n_sign_posn	1	2	2

Return Values

A pointer to the filled–in object is returned. In addition, calls to the **setlocale** subroutine with the **LC_ALL**, **LC_MONETARY** or **LC_NUMERIC** categories may cause subsequent calls to the **localeconv** subroutine to return different values based on the selection of the locale.

Note: The structure pointed to by the return value is not modified by the program but may be overwritten by a subsequent call to the **localeconv** subroutine.

Implementation Specifics

This subroutine is part of Base Operating System (BOS) Runtime.

Related Information

The **nl_langinfo** subroutine, **rpmatch** subroutine, **setlocale** subroutine.

National Language Support Overview for Programming, Subroutines Overview, Understanding Locale Subroutines in *AIX General Programming Concepts : Writing and Debugging Programs*.

lockfx, lockf, flock, or lockf64 Subroutine

Purpose

Locks and unlocks sections of open files.

Libraries

flock:	Berkeley Compatibility Library (**libbsd.a**) Berkeley Thread Safe Library (**libbsd_r.a**) (4.2.1 and later versions)
flock:	Berkeley Compatibility Library (**libbsd.a**)

Syntax

```
#include <fcntl.h>

int lockfx (FileDescriptor,
Command, Argument)
int FileDescriptor;
int Command;
struct flock *Argument;

#include <sys/lockf.h>
#include <unistd.h>

int lockf
(FileDescriptor, Request, Size)
int FileDescriptor;
int Request;
off_t Size;
```

Note: The **lockf64** subroutine applies to Version 4.2 and later releases.

```
int lockf64 (FileDescriptor,
Request, Size)
int FileDescriptor;
int Request;
off64_t Size;

#include <sys/file.h>

int flock (FileDescriptor, Operation)
int FileDescriptor;
int Operation;
```

Description

Note: The **lockf64** subroutine applies to Version 4.2 and later releases.

Attention: Buffered I/O does not work properly when used with file locking. Do not use the standard I/O package routines on files that are going to be locked.

The **lockfx** subroutine locks and unlocks sections of an open file. The **lockfx** subroutine provides a subset of the locking function provided by the **fcntl** subroutine.

The **lockf** subroutine also locks and unlocks sections of an open file. However, its interface is limited to setting only write (exclusive) locks.

Although the **lockfx**, **lockf**, **flock**, and **fcntl** interfaces are all different, their implementations are fully integrated. Therefore, locks obtained from one subroutine are honored and enforced by any of the lock subroutines.

The *Operation* parameter to the **lockfx** subroutine, which creates the lock, determines whether it is a read lock or a write lock.

The file descriptor on which a write lock is being placed must have been opened with write access.

lockf64 is equivalent to **lockf** except that a 64–bit lock request size can be given. For **lockf**, the largest value which can be used is **OFF_MAX**, for **lockf64**, the largest value is **LONGLONG_MAX**.

In the large file enabled programming environment, **lockf** is redefined to be **lock64**.

Parameters

Argument A pointer to a structure of type **flock**, defined in the **flock.h** file**.**

Command Specifies one of the following constants for the **lockfx** subroutine:

F_SETLK Sets or clears a file lock. The `l_type` field of the **flock** structure indicates whether to establish or remove a read or write lock. If a read or write lock cannot be set, the **lockfx** subroutine returns immediately with an error value of –1.

F_SETLKW Performs the same function as **F_SETLK** unless a read or write lock is blocked by existing locks. In that case, the process sleeps until the section of the file is free to be locked.

F_GETLK Gets the first lock that blocks the lock described in the **flock** structure. If a lock is found, the retrieved information overwrites the information in the **flock** structure. If no lock is found that would prevent this lock from being created, the structure is passed back unchanged except that the `l_type` field is set to **F_UNLCK**.

FileDescriptor A file descriptor returned by a successful **open** or **fcntl** subroutine, identifying the file to which the lock is to be applied or removed.

Operation Specifies one of the following constants for the **flock** subroutine:

LOCK_SH Apply a shared (read) lock.

LOCK_EX Apply an exclusive (write) lock.

LOCK_NB Do not block when locking. This value can be logically ORed with either **LOCK_SH** or **LOCK_EX**.

LOCK_UN Remove a lock.

Request Specifies one of the following constants for the **lockf** subroutine:

F_ULOCK Unlocks a previously locked region in the file.

F_LOCK Locks the region for exclusive (write) use. This request causes the calling process to sleep if the requested region overlaps a locked region, and to resume when granted the lock.

F_TEST Tests to see if another process has already locked a region. The **lockf** subroutine returns 0 if the region is unlocked. If the region is locked, then –1 is returned and the **errno** global variable is set to **EACCES**.

F_TLOCK Locks the region for exclusive use if another process has not already locked the region. If the region has already been locked by another process, the **lockf** subroutine returns a –1 and the **errno** global variable is set to **EACCES**.

Size The number of bytes to be locked or unlocked for the **lockf** subroutine. The region starts at the current location in the open file, and extends forward if the *Size* value is positive and backward if the *Size* value is negative. If the *Size* value is 0, the region starts at the current location and extends forward to the maximum possible file size, including the unallocated space after the end of the file.

Return Values

Upon successful completion, a value of 0 is returned. Otherwise, a value of –1 is returned and the **errno** global variable is set to indicate the error.

Error Codes

The **lockfx**, **lockf**, and **flock** subroutines fail if one of the following is true:

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

EINVAL The function argument is not one of **F_LOCK**, **F_TLOCK**, **F_TEST** or **F_ULOCK**; or *size* plus the current file offset is less than 0.

EINVAL An attempt was made to lock a fifo or pipe.

EDEADLK The lock is blocked by a lock from another process. Putting the calling process to sleep while waiting for the other lock to become free would cause a deadlock.

ENOLCK The lock table is full. Too many regions are already locked.

EINTR The command parameter was **F_SETLKW** and the process received a signal while waiting to acquire the lock.

EOVERFLOW The offset of the first, or if *size* is not 0 then the last, byte in the requested section cannot be represented correctly in an object of type *off_t*.

The **lockfx** and **lockf** subroutines fail if one of the following is true:

EACCES The *Command* parameter is **F_SETLK**, the `l_type` field is **F_RDLCK**, and the segment of the file to be locked is already write–locked by another process.

EACCES The *Command* parameter is **F_SETLK**, the `l_type` field is **F_WRLCK**, and the segment of a file to be locked is already read–locked or write–locked by another process.

The **flock** subroutine fails if the following is true:

EWOULDBLOCK The file is locked and the **LOCK_NB** option was specified.

Implementation Specifics

These subroutines are part of Base Operating System (BOS) Runtime.

The **flock** subroutine locks and unlocks entire files. This is a limited interface maintained for BSD compatibility, although its behavior differs from BSD in a few subtle ways. To apply a shared lock, the file must be opened for reading. To apply an exclusive lock, the file must be opened for writing.

Locks are not inherited. Therefore, a child process cannot unlock a file locked by the parent process.

Related Information

The **close** subroutine, **exec**: **execl**, **execv**, **execle**, **execlp**, **execvp**, or **exect** subroutine, **fcntl** subroutine, **fork** subroutine, **open**, **openx**, or **creat** subroutine.

Files, Directories, and File Systems for Programmers in *AIX General Programming Concepts : Writing and Debugging Programs*.

loginfailed Subroutine

Purpose

Records an unsuccessful login attempt.

Library

Security Library (**libc.a**)

Syntax

```
int loginfailed (User, Host, Tty)
char *User;
char *Host;
char *Tty;
```

Note: This subroutine is not thread–safe.

Description

The **loginfailed** subroutine performs the processing necessary when an unsuccessful login attempt occurs. If the specified user name is not valid, the **UNKNOWN_USER** value is substituted for the user name. This substitution prevents passwords entered as the user name from appearing on screen.

The following attributes in /**etc**/**security**/**lastlog** file are updated for the specified user, if the user name is valid:

time_last_unsuccessful_login	Contains the current time.
tty_last_unsuccessful_login	Contains the value specified by the *Tty* parameter.
host_last_unsuccessful_login	Contains the value specified by the *Host* parameter, or the local hostname if the *Host* parameter is a null value.
unsuccessful_login_count	Indicates the number of unsuccessful login attempts. The **loginfailed** subroutine increments this attribute by one for each failed attempt.

A login failure audit record is cut to indicate that an unsuccessful login attempt occurred. A **utmp** entry is appended to /**etc**/**security**/**failedlogin** file, which tracks all failed login attempts.

If the current unsuccessful login and the previously recorded unsuccessful logins constitute too many unsuccessful login attempts within too short of a time period (as specified by the **logindisable** and **logininterval** port attributes), the port is locked. When a port is locked, a PORT_Locked audit record is written to inform the system administrator that the port has been locked.

If the login retry delay is enabled (as specified by the **logindelay** port attribute), a sleep occurs before this subroutine returns. The length of the sleep (in seconds) is determined by the **logindelay** value multiplied by the number of unsuccessful login attempts that occurred in this process.

Parameters

User	Specifies the user's login name who has unsuccessfully attempted to login.
Host	Specifies the name of the host from which the user attempted to login. If the *Host* parameter is Null, the name of the local host is used.
Tty	Specifies the name of the terminal on which the user attempted to login.

Security

Access Control: The calling process must have access to the account information in the user database and the port information in the port database.

File Accessed:

Mode	File
r	/etc/security/user
rw	/etc/security/lastlog
r	/etc/security/login.cfg
rw	/etc/security/portlog
w	/etc/security/failedlogin

Auditing Events:

Event	Information
USER_Login	username
PORT_Locked	portname

Return Values

Upon successful completion, the **loginfailed** subroutine returns a value of 0. If an error occurs, a value of –1 is returned and errno is set to indicate the error.

Error Codes

The **loginfailed** subroutine fails if one or more of the following values is true:

EACCES	The current process does not have access to the user or port database.
EPERM	The current process does not have permission to write an audit record.

Implementation Specifics

This subroutine is part of Base Operating System (BOS) Runtime.

Related Information

The **authenticate** subroutine, **getpcred** subroutine, **getpenv** subroutine, **loginrestrictions** subroutine, **loginsuccess** subroutine, **setpcred** subroutine, **setpenv** subroutine.

List of Security and Auditing Services in *AIX General Programming Concepts : Writing and Debugging Programs*.

Subroutines Overview in *AIX General Programming Concepts : Writing and Debugging Programs*.

loginrestrictions Subroutine

Purpose

Determines if a user is allowed to access the system.

Library

Security Library (**libc.a**)

Syntax

```
#include <login.h>

int loginrestrictions (Name, Mode, Tty, Msg)
char *Name;
int Mode;
char *Tty;
char **Msg;
```

Note: This subroutine is not thread–safe.

Description

The **loginrestrictions** subroutine determines if the user specified by the *Name* parameter is allowed to access the system. The *Mode* parameter gives the mode of account usage and the *Tty* parameter defines the terminal used for access. The *Msg* parameter returns an informational message explaining why the **loginrestrictions** subroutine failed.

This subroutine is unsuccessful if any of the following conditions exists:

- The user's account has expired as defined by the **expires** user attribute.
- The user's account has been locked as defined by the **account_locked** user attribute.
- The user attempted too many unsuccessful logins as defined by the **loginretries** user attribute.
- The user is not allowed to access the given terminal as defined by the **ttys** user attribute.
- The user is not allowed to access the system at the present time as defined by the **logintimes** user attribute.
- The *Mode* parameter is set to the **S_LOGIN** value or the **S_RLOGIN** value, and too many users are logged in as defined by the **maxlogins** system attribute.
- The *Mode* parameter is set to the **S_LOGIN** value and the user is not allowed to log in as defined by the **login** user attribute.
- The *Mode* parameter is set to the **S_RLOGIN** value and the user is not allowed to log in from the network as defined by the **rlogin** user attribute.
- The *Mode* parameter is set to the **S_SU** value and other users are not allowed to use the **su** command as defined by the **su** user attribute, or the group ID of the current process cannot use the **su** command to switch to this user as defined by the **sugroups** user attribute.
- The *Mode* parameter is set to the **S_DAEMON** value and the user is not allowed to run processes from the **cron** or **src** subsystem as defined by the **daemon** user attribute.
- The terminal is locked as defined by the **locktime** port attribute.
- The user cannot use the terminal to access the system at the present time as defined by the **logintimes** port attribute.
- The user is not the root user and the /**etc**/**nologin** file exists.

Note: The **loginrestrictions** subroutine is not safe in a multi–threaded environment. To use **loginrestrictions** in a threaded application, the application must keep the integrity of each thread.

Parameters

Name Specifies the user's login name whose account is to be validated.

Mode Specifies the mode of usage. Valid values as defined in the **login.h** file are listed below. The *Mode* parameter has a value of 0 or one of the following values:

S_LOGIN Verifies that local logins are permitted for this account.

S_SU Verifies that the **su** command is permitted and the current process has a group ID that can invoke the **su** command to switch to the account.

S_DAEMON Verifies the account can invoke daemon or batch programs through the **src** or **cron** subsystems.

S_RLOGIN Verifies the account can be used for remote logins through the **rlogind** or **telnetd** programs.

Tty Specifies the terminal of the originating activity. If this parameter is a null pointer or a null string, no tty origin checking is done.

Msg Returns an informative message indicating why the **loginrestrictions** subroutine failed. Upon return, the value is either a pointer to a valid string within memory allocated storage or a null value. If a message is displayed, it is provided based on the user interface.

Security

Access Control:The calling process must have access to the account information in the user database and the port information in the port database.

File Accessed:

Mode	Files
r	/etc/security/user
r	/etc/security/login.cfg
r	/etc/security/portlog
r	/etc/passwd

Return Values

If the account is valid for the specified usage, the **loginrestrictions** subroutine returns a value of 0. Otherwise, a value of –1 is returned, the **errno** global value is set to the appropriate error code, and the *Msg* parameter returns an informative message explaining why the specified account usage is invalid.

Error Codes

The **loginrestrictions** subroutine fails if one or more of the following values is true:

ENOENT The user specified does not have an account.

ESTALE The user's account is expired.

EPERM The user's account is locked, the specified terminal is locked, the user has had too many unsuccessful login attempts, or the user cannot log in because the /**etc**/**nologin** file exists.

EACCES One of the following conditions exists:

- The specified terminal does not have access to the specified account.
- The *Mode* parameter is the **S_SU** value and the current process is not permitted to use the **su** command to access the specified user.
- Access to the account is not permitted in the specified mode.
- Access to the account is not permitted at the current time.
- Access to the system with the specified terminal is not permitted at the current time.

EAGAIN The *Mode* parameter is neither the **S_LOGIN** value nor the **S_RLOGIN** value, and all the user licenses are in use.

EINVAL The *Mode* parameter has a value other than **S_LOGIN**, **S_SU**, **S_DAEMON**, **S_RLOGIN**, or **0**.

Implementation Specifics

This subroutine is part of Base Operating System (BOS) Runtime.

Related Information

The **authenticate** subroutine, **getpcred** subroutine, **getpenv** subroutine, **loginfailed** subroutine, **loginsuccess** subroutine, **setpcred** subroutine, **setpenv** subroutine.

The **cron** daemon.

The **login** command, **rlogin** command, **telnet, tn,** or **tn3270** command, **su** command.

List of Security and Auditing Services in *AIX General Programming Concepts : Writing and Debugging Programs*.

Subroutines Overview in *AIX General Programming Concepts : Writing and Debugging Programs*.

loginsuccess Subroutine

Purpose

Records a successful log in.

Library

Security Library (**libc.a**)

Syntax

```
int loginsuccess (User, Host, Tty, Msg)
char *User;
char *Host;
char *Tty;
char **Msg;
```

Note: This subroutine is not thread–safe.

Description

The **loginsuccess** subroutine performs the processing necessary when a user successfully logs into the system. This subroutine updates the following attributes in the **/etc/security/lastlog** file for the specified user:

time_last_login	Contains the current time.
tty_last_login	Contains the value specified by the *Tty* parameter.
host_last_login	Contains the value specified by the *Host* parameter or the local host name if the *Host* parameter is a null value.
unsuccessful_login _count	Indicates the number of unsuccessful login attempts. The **loginsuccess** subroutine resets this attribute to a value of 0.

Additionally, a login success audit record is cut to indicate in the audit trail that this user has successfully logged in.

A message is returned in the *Msg* parameter that indicates the time, host, and port of the last successful and unsuccessful login. The number of unsuccessful login attempts since the last successful login is also provided to the user.

Parameters

User	Specifies the login name of the user who has successfully logged in.
Host	Specifies the name of the host from which the user logged in. If the *Host* parameter is a null value, the name of the local host is used.
Tty	Specifies the name of the terminal which the user used to log in.
Msg	Returns a message indicating the delete time, host, and port of the last successful and unsuccessful logins. The number of unsuccessful login attempts since the last successful login is also provided. Upon return, the value is either a pointer to a valid string within memory allocated storage or a null pointer. It is the responsibility of the calling program to **free( )** the returned storage.

Security

Access Control: The calling process must have access to the account information in the user database.

File Accessed:

Mode	File
rw	/etc/security/lastlog

Auditing Events:

Event	Information
USER_Login	username

Return Values

Upon successful completion, the **loginsuccess** subroutine returns a value of 0. Otherwise, a value of –1 is returned and the **errno** global value is set to indicate the error.

Error Codes

The **loginsuccess** subroutine fails if one or more of the following values is true:

ENOENT	The specified user does not exist.
EACCES	The current process does not have write access to the user database.
EPERM	The current process does not have permission to write an audit record.

Implementation Specifics

This subroutine is part of Base Operating System (BOS) Runtime.

Related Information

The **authenticate** subroutine, **getpcred** subroutine, **getpenv** subroutine, **loginfailed** subroutine, **loginrestrictions** subroutine, **setpcred** subroutine, **setpenv** subroutine.

List of Security and Auditing Services in *AIX General Programming Concepts : Writing and Debugging Programs*.

Subroutines Overview in *AIX General Programming Concepts : Writing and Debugging Programs*.

lsearch or lfind Subroutine

Purpose

Performs a linear search and update.

Library

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

Syntax

```
void *lsearch (Key, Base, NumberOfElementsPointer, Width,
ComparisonPointer)
const void *Key;
void *Base;
size_t Width, *NumberOfElementsPointer;
int (*ComparisonPointer) (cont void*, const void*);

void *lfind (Key, Base, NumberOfElementsPointer, Width,
ComparisonPointer)
const void *Key, Base;
size_t Width, *NumberOfElementsPointer;
int (*ComparisonPointer) (cont void*, const void*);
```

Description

Warning: Undefined results can occur if there is not enough room in the table for the **lsearch** subroutine to add a new item.

The **lsearch** subroutine performs a linear search.

The algorithm returns a pointer to a table where data can be found. If the data is not in the table, the program adds it at the end of the table.

The **lfind** subroutine is identical to the **lsearch** subroutine, except that if the data is not found, it is not added to the table. In this case, a NULL pointer is returned.

The pointers to the *Key* parameter and the element at the base of the table should be of type pointer–to–element and cast to type pointer–to–character. The value returned should be cast into type pointer–to–element.

The comparison function need not compare every byte; therefore, the elements can contain arbitrary data in addition to the values being compared.

Parameters

Base	Points to the first element in the table.
ComparisonPointer	Specifies the name (that you supply) of the comparison function (**strcmp**, for example). It is called with two parameters that point to the elements being compared.
Key	Specifies the data to be sought in the table.
NumberOfElementsPointer	Points to an integer containing the current number of elements in the table. This integer is incremented if the data is added to the table.
Width	Specifies the size of an element in bytes.

The comparison function compares its parameters and returns a value as follows:

- If the first parameter equals the second parameter, the *ComparisonPointer* parameter returns a value of 0.

- If the first parameter does not equal the second parameter, the *ComparisonPointer* parameter returns a value of 1.

Return Values

If the sought entry is found, both the **lsearch** and **lfind** subroutines return a pointer to it. Otherwise, the **lfind** subroutine returns a null pointer and the **lsearch** subroutine returns a pointer to the newly added element.

Implementation Specifics

These subroutines are part of Base Operating System (BOS) Runtime.

Related Information

The **bsearch** subroutine, **hsearch** subroutine, **qsort** subroutine, **tsearch** subroutine.

Donald E. Knuth. *The Art of Computer Programming*, Volume 3, 6.1, Algorithm S. Reading, Massachusetts: Addison–Wesley, 1981.

Searching and Sorting Example Program and Subroutines Overview in *AIX General Programming Concepts : Writing and Debugging Programs*.

lseek, llseek or lseek64 Subroutine

Purpose

Moves the read–write file pointer.

Library

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

Syntax

```
off_t lseek (FileDescriptor, Offset, Whence)
int FileDescriptor, Whence;
off_t Offset;
```

```
offset_t llseek (FileDescriptor, Offset, Whence)
int FileDescriptor, Whence;
offset_t Offset;
```

Note: The **lseek64** subroutine applies to Version 4.2 and later releases.

```
off64_t lseek64 (FileDescriptor, Offset, Whence)
int FileDescriptor, Whence;
off64_t Offset;
```

Description

Note: The **lseek64** subroutine applies to Version 4.2 and later releases.

The **lseek**, **llseek**, and **lseek64** subroutines set the read–write file pointer for the open file specified by the *FileDescriptor* parameter. The **lseek** subroutine limits the *Offset* to **OFF_MAX**.

In AIX Version 4.1, the **llseek** subroutine limits the *Offset* to **OFF_MAX** if the file associated with *FileDescriptor* is a regular file or a directory and to **DEV_OFF_MAX** if the file associated with *FileDescriptor* is a block special or character special file.

In Version 4.2, both the **llseek** subroutine and the **lseek64** subroutine limit the *Offset* to the maximum file size for the file size for the file associated with *FileDescriptor* and to **DEV_OFF_MAX** if the file associated with *FileDescriptor* is a block special or character special file.

In the large file enabled programming environment, **lseek** subroutine is redefined to **lseek64**.

Parameters

FileDescriptor	Specifies a file descriptor obtained from a successful **open** or **fcntl** subroutine.
Offset	Specifies a value, in bytes, that is used in conjunction with the *Whence* parameter to set the file pointer. A negative value causes seeking in the reverse direction.
Whence	Specifies how to interpret the *Offset* parameter by setting the file pointer associated with the *FileDescriptor* parameter to one of the following variables:

SEEK_SET	Sets the file pointer to the value of the *Offset* parameter.
SEEK_CUR	Sets the file pointer to its current location plus the value of the *Offset* parameter.
SEEK_END	Sets the file pointer to the size of the file plus the value of the *Offset* parameter.

Return Values

Upon successful completion, the resulting pointer location, measured in bytes from the beginning of the file, is returned. If either the **lseek** or **llseek** subroutines are unsuccessful, a value of –1 is returned and the **errno** global variable is set to indicate the error.

Error Codes

The **lseek** or **llseek** subroutines are unsuccessful and the file pointer remains unchanged if any of the following are true:

EBADF	The *FileDescriptor* parameter is not an open file descriptor.
ESPIPE	The *FileDescriptor* parameter is associated with a pipe (FIFO) or a socket.
EINVAL	The resulting offset would be greater than the maximum offset allowed for the file or device associated with *FileDescriptor*.
EOVERFLOW	The resulting offset is larger than can be returned properly.

Implementation Specifics

This subroutine is part of Base Operating System (BOS) Runtime.

Files

/usr/include/unistd.h	Defines standard macros, data types and subroutines.

Related Information

The **fcntl** subroutine, **fseek**, **rewind**, **ftell**, **fgetpos**, or **fsetpos** subroutine, **open**, **openx**, or **creat** subroutine, **read**, **readx**, **readv**, or **readvx** subroutine, **write**, **writex**, **writev**, or **writevx** subroutine.

Files, Directories, and File Systems for Programmers in *AIX General Programming Concepts : Writing and Debugging Programs*.

lvm_changelv Subroutine

Purpose

Changes the attributes of a logical volume.

Library

Logical Volume Manager Library (**liblvm.a**)

Syntax

```
#include <lvm.h>

int lvm_changelv (ChangeLV)
struct changelv *ChangeLV;
```

Description

Note: You must have root user authority to use this subroutine.

The **lvm_changelv** subroutine changes the attributes of an existing logical volume.

The **changelv** structure pointed to by the *ChangeLV* parameter is defined in the **lvm.h** file and contains the following fields:

```
struct changelv{
   struct lv_id lv_id;
   char *lvname;
   long maxsize;
   long permissions;
   long bb_relocation;
   long mirror_policy;
   long write_verify;
   long mirwrt_consist;
 }
struct lv_id{
   struct unique_id vg_id;
   long    minor_num;}
```

Field	Definition
lv_id	Specifies the logical volume to be changed.
lvname	Specifies either the full path name of the logical volume or a single file name that must reside in the /**dev** directory, for example **rhd1**. This field must be a null–terminated string that ranges from 1 to **LVM_NAMESIZ** bytes, including the null byte, and must be the name of a raw or character device. If a raw or character device is not specified for the lvname field, the Logical Volume Manager (LVM) adds an **r** to the file name to have a raw device name. If there is no raw device entry for this name, the LVM returns the **LVM_NOTCHARDEV** error code.
maxsize	Specifies the new maximum size of the logical volume in number of logical partitions (1 – **LVM_MAXLPS**). A change in the maxsize field does not change the existing size of the logical volume.
permissions	Specifies that the permission assigned to the logical volume is either read–only or read/write.
bb_relocation	Specifies if bad block relocation is desired.
mirror_policy	Specifies how the copies of the logical partition should be written. The values for this field can be either **LVM_SEQUENTIAL** or **LVM_PARALLEL**.

Field	Definition
`write_verify`	Specifies if writes to the logical volume should be checked for successful completion. The value for this field is either **LVM_VERIFY** or **LVM_NOVERIFY**. Any other fields in the parameter list that are not to be changed should either contain a 0 or be set to null if they are pointers.
`mirwrt_consist`	Tells whether mirror–write consistency recovery will be performed for this logical volume. The LVM always insures data consistency among mirrored copies of a logical volume during normal I/O processing. For every write to a logical volume, the LVM generates a write request for every mirror copy. A problem arises if the system crashes in the middle of processing a mirrored write before all copies are written. If mirror–write consistency recovery is requested for a logical volume, the LVM keeps additional information to allow recovery of these inconsistent mirrors. Mirror–write consistency recovery should be performed for most mirrored logical volumes. Logical volumes, such as page space, that do not use the existing data when the volume group is re–varied on do not need this protection.

The logical volume must not be open when trying to change the `permissions`, `bb_relocation`, `write_verify`, `mirror_policy`, or `mirwrt_consist` fields. If the volume group that contains the logical volume to be changed is not on–line, an error will be returned.

Parameters

ChangeLV	Points to the **changelv** structure.

Return Values

Upon successful completion, a value of 0 is returned.

Error Codes

If the **changelv** subroutine does not complete successfully it returns one of the following values:

LVM_ALLOCERR	A memory allocation error occurred.
LVM_DALVOPN	The volume group reserved logical volume could not be opened.
LVM_FORCEOFF	The volume group has been forcefully varied off due to a loss of quorum.
LVM_INVALID_MIN_NUM	The minor number received was not valid.
LVM_INVALID_PARAM	A field in the **changelv** structure is not valid, or the pointer to the **changelv** structure is not valid.
LVM_INVCONFIG	An error occurred while attempting to configure this volume group into the kernel. This error will normally result if the module ID is invalid, if the major number given is already in use, or if the volume group device could not be opened.
LVM_INV_DEVENT	The logical volume device entry is not valid and cannot be checked to determine if it is raw.
LVM_LVEXIST	A logical volume already exists with the name passed into the routine.

LVM_LVOPEN	The logical volume was open. It must be closed to change the `permissions`, `bb_relocation`, `write_verify`, `mirror_policy`, or `mirwrt_consist` field.
LVM_MAPFBSY	The volume group is currently locked because system management on the volume group is being done by another process.
LVM_MAPFOPN	The mapped file, which contains a copy of the volume group descriptor area used for making changes to the volume group, could not be opened.
LVM_MAPFRDWR	An error occurred while trying to read or write the mapped file.
LVM_NOTCHARDEV	The device is not a raw or character device.
LVM_OFFLINE	A routine that requires a volume group to be online has encountered an offline volume group.

Implementation Specifics

This subroutine is part of Base Operating System (BOS) Runtime.

Related Information

The **lvm_querylv** subroutine, **lvm_varyonvg** subroutine.

List of Logical Volume Subroutines and Logical Volume Programming Overview in *AIX General Programming Concepts : Writing and Debugging Programs*.

lvm_changepv Subroutine

Purpose

Changes the attributes of a physical volume in a volume group.

Library

Logical Volume Manager Library (**liblvm.a**)

Syntax

```
#include <lvm.h>

int lvm_changepv (ChangePV)
struct changepv *ChangePV;
```

Description

Note: You must have root user authority to use this subroutine.

The **lvm_changepv** subroutine changes the state of the specified physical volume.

The **changepv** structure pointed to by the *ChangePV* parameter is defined in the **lvm.h** file and contains the following fields:

```
struct changepv{
   struct unique_id vg_id;
   struct unique_id pv_id;
   long rem_ret;
   long allocation;}
```

Field	Definition
`pv_id`	Specifies the state of the physical volume to be changed
`rem_ret`	Should be set to either **LVM_REMOVEPV** or **LVM_RETURNPV** value. The **LVM_REMOVEPV** value temporarily removes the physical volume from the volume group. The **LVM_RETURNPV** returns the physical volume to the volume group.

When a physical volume is temporarily removed from the volume group, there will be no access to that physical volume through the Logical Volume Manager (LVM) while that physical volume is in the removed state. Also, when a physical volume is removed from the volume group, any copies of the volume group descriptor area which are contained on that physical volume are removed from the volume group. Therefore, copies of the volume group descriptor area will not be counted in the quorum count of descriptor area copies which are needed for a volume group to be varied on.

The **allocation** field should be set to **LVM_NOALLOCPV** to disallow the allocation of physical partitions to the physical volume, or **LVM_ALLOCPV** to allow the allocation of physical partitions to the physical volume. It is not necessary to change both state fields; for example, the allocation field could be set to **LVM_NOALLOCPV** and the `rem_ret` field could simply be set to 0 to indicate no change is desired. The `vg_id` field identifies the volume group that contains the physical volume to be changed. The volume group must be online, or an error is returned.

Parameters

ChangePV	Specifies a pointer to the **changepv** structure.

Return Values

Upon successful completion, the **lvm_changepv** subroutine returns one of the following positive values:

LVM_REMRET_INCOMP	The physical volume was removed or returned in the volume group descriptor area but not in the kernel. The change will take effect at the next varyon.
LVM_SUCCESS	The physical volume was changed successfully.

Error Codes

If the **lvm_changepv** subroutine fails, it returns one of the following values:

LVM_ALLOCERR	A memory allocation error occurred.
LVM_BELOW_QRMCNT	The physical volume cannot be removed because there would not be a quorum of available physical volumes.
LVM_DALVOPN	The logical volume reserved by the volume group could not be opened.
LVM_FORCEOFF	The volume group has been forcefully varied off due to a loss of quorum.
LVM_INVALID_PARAM	A field in the **changepv** structure is invalid, or the pointer to the **changepv** structure is invalid.
LVM_INVCONFIG	An error occurred while attempting to configure this volume group into the kernel. This error will normally result if the module ID is invalid, if the major number given is already in use, or if the volume group device could not be opened.
LVM_INV_DEVENT	The device entry for the physical volume is invalid and cannot be checked to determine if it is raw.
LVM_MAPFBSY	The volume group is currently locked because system management on the volume group is being done by another process.
LVM_MAPFOPN	The mapped file, which contains a copy of the volume group descriptor area used for making changes to the volume group, could not be opened.
LVM_MAPFRDWR	An error occurred while trying to read or write the mapped file.
LVM_NOTCHARDEV	The device specified is not a raw or character device.
LVM_OFFLINE	The volume group containing the physical volume to be changed is offline and should be online.

Implementation Specifics

This subroutine is part of Base Operating System (BOS) Runtime.

Related Information

The **lvm_querypv** subroutine.

List of Logical Volume Subroutines and Logical Volume Programming Overview in *AIX General Programming Concepts : Writing and Debugging Programs*.

lvm_createlv Subroutine

Purpose

Creates an empty logical volume in a specified volume group.

Library

Logical Volume Manager Library (**liblvm.a**)

Syntax

```
#include <lvm.h>

int lvm_createlv
(CreateLV)
struct createlv *CreateLV;
```

Description

Note: You must have root user authority to use this subroutine.

The **lvm_createlv** subroutine creates an empty logical volume in an existing volume group with the information supplied. The **lvm_extendlv** subroutine should be called to allocate partitions once the logical volume is created.

The **createlv** structure pointed to by the *CreateLV* parameter is defined in the **lvm.h** file and contains the following fields:

```
struct createlv {
  char *lvname;
  struct unique_id vg_id;
  long minor_num;
  long maxsize;
  long mirror_policy;
  long permissions;
  long bb_relocation;
  long write_verify;
  long mirwrt_consist;
}
struct unique_id{
#ifndef_64BIT_
    unsigned long word1;
    unsigned long word2;
    unsigned long word3;
    unsigned long word4;
#else
    unsigned int word1;
    unsigned int word2;
    unsigned int word3;
    unsigned int word4;
#endif
};
```

Field	Definition
`lvname`	Specifies the special file name of the logical volume, and can be either the full path name or a single file name that must reside in the **/dev** directory (for example, **rhd1**). All name fields must be null–terminated strings of from 1 to **LVM_NAMESIZ** bytes, including the null byte. If a raw or character device is not specified for the `lvname` field, the Logical Volume Manager (LVM) will add an **r** to the file name in order to have a raw device name. If there is no raw device entry for this name, the LVM will return the **LVM_NOTCHARDEV** error code.
`vg_id`	Specifies the unique ID of the volume group that will contain the logical volume.
`minor_num`	Must be in the range from 1 to the `maxlvs` value. The `maxlvs` field is set when a volume group is created and is returned by the **lvm_queryvg** subroutine.
`maxsize`	Indicates the maximum size in logical partitions for the logical volume and must be in the range of 1 to **LVM_MAXLPS**.
`mirror_policy`	Specifies how the physical copies will be written. The **mirror_policy** field should be either **LVM_SEQUENTIAL** or **LVM_PARALLEL** to indicate how the physical copies of a logical partition are to be written when there is more than one copy.
`permissions`	Indicates read/write or read only permission for the logical volume.
`bb_relocation`	Indicates that bad block relocation is desired.
`write_verify`	Indicates that writes to the logical volume are to be verified as successful.
`mirwrt_consist`	Indicates whether mirror write consistency recovery will be performed for this logical volume.

The LVM always ensures data consistency among mirrored copies of a logical volume during normal I/O processing. For every write to a logical volume, the LVM generates a write request for every mirror copy. A problem arises if the system crashes in the middle of processing a mirrored write (before all copies are written). If mirror–write consistency recovery is requested for a logical volume, the LVM keeps additional information to allow recovery of these inconsistent mirrors. Mirror write consistency recovery should be performed for most mirrored logical volumes. Logical volumes, such as the page space, that do not use the existing data when the volume group is re–varied on do not need this protection.

All fields in the **createlv** structure must have a valid value in them, or an error will be returned.

The **lvm_createlv** subroutine uses the **createlv** structure to build an information area for the logical volume. If the volume group that is to contain this logical volume is not varied on–line, the **LVM_OFFLINE** error code is returned.

Possible values for the `mirror_policy` field are:

LVM_SEQUENTIAL	For this logical volume, use a sequential method of writing the physical copies (if more than one) of a logical partition.
LVM_PARALLEL	For this logical volume, use a parallel method of writing the physical copies (if more than one) of a logical partition.

Possible values for the `permissions` field are:

LVM_RDONLY	Create the logical volume with read only permission.
LVM_RDWR	Create the logical volume with read/write permission.

Possible values for the `bb_relocation` field are:

LVM_RELOC Bad block relocation is desired.

LVM_NORELOC Bad block relocation is not desired.

Possible values for the `write_verify` field are:

LVM_VERIFY Write verification is desired.

LVM_NOVERIFY Write verification is not desired.

Possible values for the `mirwrt_consist` field are:

LVM_CONSIST Mirror write consistency recovery will be done for this logical volume.

LVM_NOCONSIST Mirror write consistency recovery will not be done for this logical volume.

Parameters

CreateLV Points to the **createlv** structure.

Return Values

The **lvm_createlv** subroutine returns a value of 0 upon successful completion.

Error Codes

If the **lvm_createlv** subroutine fails, it returns one of the following values:

LVM_ALLOCERR A memory allocation error has occurred.

LVM_DALVOPN The descriptor area logical volume could not be opened.

LVM_FORCEOFF The volume group has been forcefully varied off due to a loss of quorum.

LVM_INVALID_MIN_NUM A minor number passed into the routine is invalid.

LVM_INVALID_PARAM A field in the **createlv** structure is invalid, or the pointer to the **createlv** structure is invalid.

LVM_INV_DEVENT The logical volume device entry is invalid and cannot be checked to determine if it is raw.

LVM_LVEXIST A logical volume already exists with the name passed into the routine.

LVM_MAPFBSY The volume group is currently locked because system management on the volume group is being done by another process.

LVM_MAPFOPN The mapped file, which contains a copy of the volume group descriptor area used for making changes to the volume group, could not be opened.

LVM_MAPFRDWR An error occurred while trying to read or write the mapped file.

LVM_NOTCHARDEV The **lvname** name given does not represent a raw or character device.

LVM_OFFLINE A routine that requires a volume group to be online has encountered one that is offline.

LVM_VGFULL The volume group that the logical volume was requested to be a member of already has the maximum number of logical volumes.

Implementation Specifics

This subroutine is part of Base Operating System (BOS) Runtime.

Related Information

The **lvm_extendlv** subroutine, **lvm_querylv** subroutine, **lvm_queryvg** subroutine, **lvm_varyonvg** subroutine.

List of Logical Volume Subroutines and Logical Volume Programming Overview in *AIX General Programming Concepts : Writing and Debugging Programs*.

lvm_createvg Subroutine

Purpose

Creates a new volume group and installs the first physical volume.

Library

Logical Volume Manager Library (**liblvm.a**)

Syntax

```
#include <lvm.h>

int lvm_createvg (CreateVG)
struct createvg *CreateVG;
```

Description

Note: You must have root user authority to use this subroutine.

The **lvm_createvg** subroutine creates a new volume group and installs its first physical volume. The physical volume must not exist in another volume group.

The **createvg** structure pointed to by the *CreateVG* parameter is found in the **lvm.h** file and defined as follows:

```
struct createvg
        {
        mid_t kmid;
        char *vgname;
        long vg_major;
        char *pvname;
        long maxlvs;
        long ppsize;
        long vgda_size;
        short int override;
        struct unique_id vg_id;
        };
```

Field	Definition
`kmid`	Specifies the module ID that identifies the entry point of the logical volume device driver module. The module ID is returned when the logical volume device driver is loaded into the kernel.
`vgname`	Specifies the character special file name that is either the full path name or a file name that resides in the /**dev** directory (for example, **rvg13**) of the volume group device. This device is actually a logical volume with the minor number 0, which is reserved for use by the Logical Volume Manager (LVM).
`vg_major`	Specifies the major number for the volume group that is to be created.
`pvname`	Specifies the character special file name, which is either the full path name or a single file name that resides in the /**dev** directory (for example, **rhdisk0**) of the physical volume being installed in the new volume group.
`maxlvs`	Specifies the maximum number of logical volumes allowed in the volume group. Minor number 0 is reserved for the LVM. User logical volumes can range from minor number 1 through **LVM_MAXLVS** – 1.

Field	Definition
`ppsize`	Specifies the size of the physical partitions in the volume group. The range is **LVM_MINPPSIZ** to **LVM_MAXPPSIZ**. The size in bytes of every physical partition in the volume group is 2 to the power of the `ppsize` field.
`vgda_size`	Indicates the number of 512–byte blocks which are to be reserved for one copy of the volume group descriptor area. The range is from **LVM_MINVGDASIZ** to **LVM_MAXVGDASIZ**. Twice this amount of space is reserved on each physical volume in the volume group so that two copies of the volume group descriptor area can be saved when needed.
`override`	Specifies whether or not the **LVM_VGMEMBER** error code should be ignored. If the `override` field is TRUE, the LVM creates the volume group with the specified physical volume even if it appears to belong to another volume group, as long as that volume group is not varied on. If the volume group is varied on, the **LVM_MEMACTVVG** error code is returned. If the `override` field is FALSE, the LVM returns the **LVM_VGMEMBER** error code, if the specified physical volume is a member of another volume group whether that volume group is varied on or off. If the **LVM_MEMACTVVG** or **LVM_VGMEMBER** error code is returned, the `vg_id` field contains the ID of the volume group of which the specified physical volume is a member.

The `vg_id` field is an output field in which the ID of the newly created volume group will be returned upon successful completion.

The physical volume installed into the new volume group contains two copies of the volume group descriptor area in the reserved area at the beginning of the physical volume, since this is the first physical volume installed. The volume group descriptor area contains information about the physical and logical volumes in the volume group. This descriptor area is used by the LVM to manage the logical volumes and physical volumes in the volume group.

Parameters

CreateVG	Points to the **createvg** structure.

Return Values

The **lvm_createvg** subroutine returns a value of 0 upon successful completion.

Error Codes

If the **lvm_createvg** subroutine fails, it returns one of the following values:

LVM_ALLOCERR	A memory allocation error occurred.
LVM_BADBBDIR	The physical volume could not be installed into the volume group because the bad–block directory could not be read from and or written to.
LVM_DALVOPN	The logical volume reserved by the volume group could not be opened.
LVM_INVALID_PARAM	A field in the **createvg** structure is not valid.
LVM_INV_DEVENT	A device entry is invalid and cannot be checked to determine if it is raw.
LVM_LVMRECERR	The LVM record, which contains information about the volume group descriptor area, could not be read or written.

LVM_MAJINUSE	The specified major number is already being used by another device.
LVM_MEMACTVVG	The physical volume specified is a member of another volume group that is varied on. This value is returned only when the `override` field is set to TRUE.
LVM_NOTCHARDEV	The device specified is not a raw or character device.
LVM_PVOPNERR	The physical volume device could not be opened.
LVM_RDPVID	The record that contains the physical volume ID could not be read.
LVM_VGDASPACE	The physical volume cannot be installed into the specified volume group because there is not enough space in the volume group descriptor area to add a description of the physical volume and its partitions.
LVM_VGMEMBER	The physical volume cannot be installed into the specified volume group because its LVM record indicates it is already a member of another volume group. If the caller feels that the information in the LVM record is incorrect, the `override` field can be set to TRUE in order to override this error. This error is only returned when the `override` field is set to FALSE.
LVM_WRTDAERR	An error occurred while trying to initialize either the volume group descriptor area, the volume group status area, or the mirror write consistency cache area on the physical volume.

Implementation Specifics

This subroutine is part of Base Operating System (BOS) Runtime.

Related Information

The **lvm_varyonvg** subroutine.

List of Logical Volume Subroutines and Logical Volume Programming Overview in *AIX General Programming Concepts : Writing and Debugging Programs*.

lvm_deletelv Subroutine

Purpose

Deletes a logical volume from its volume group.

Library

Logical Volume Manager Library (**liblvm.a**)

Syntax

```
#include <lvm.h>

int lvm_deletelv (LV_ID)
struct lv_id *LV_ID;
```

Description

The **lvm_deletelv** subroutine deletes the logical volume specified by the *LV_ID* parameter from its volume group. The logical volume must not be opened, and the volume group must be online, or an error is returned. Also, all logical partitions belonging to this logical volume must be removed using the **lvm_reducelv** subroutine before the logical volume can be deleted.

Note: You must have root user authority to use this subroutine.

Parameters

LV_ID	Specifies the logical volume to be deleted.

Return Values

The **lvm_deletelv** subroutine returns a value of 0 upon successful completion.

Error Codes

If the **lvm_deletelv** subroutine fails, it returns one of the following values:

LVM_ALLOCERR	A memory allocation error occurred.
LVM_DALVOPN	The logical volume reserved by the volume group could not be opened.
LVM_FORCEOFF	The volume group has been forcefully varied off due to a loss of quorum.
LVM_INVALID_MIN_NUM	An invalid minor number was received.
LVM_INVALID_PARAM	The logical volume ID passed in is not a valid logical volume, or the pointer to the logical volume is invalid.
LVM_INVCONFIG	An error occurred while attempting to configure this volume group into the kernel. This error will normally result if the major number in the mapped file is invalid.
LVM_INV_DEVENT	The device entry for the logical volume is invalid and cannot be checked to determine if it is raw.
LVM_LVOPEN	An open logical volume was encountered when it should be closed.
LVM_MAPFBSY	The volume group is currently locked because system management on the volume group is being done by another process.

LVM_MAPFOPN	The mapped file, which contains a copy of the volume group descriptor area used for making changes to the volume group, could not be opened.
LVM_MAPFRDWR	An error occurred while trying to read or write the mapped file.
LVM_NODELLV	The logical volume cannot be deleted because there are existing logical partitions.
LVM_NOTCHARDEV	The device specified is not a raw or character device.
LVM_OFFLINE	A routine that requires a volume group to be online has encountered one that is offline.

Implementation Specifics

This subroutine is part of Base Operating System (BOS) Runtime.

Related Information

The **lvm_varyonvg** subroutine.

List of Logical Volume Subroutines and Logical Volume Programming Overview in *AIX General Programming Concepts : Writing and Debugging Programs*.

lvm_deletepv Subroutine

Purpose

Deletes a physical volume from a volume group.

Library

Logical Volume Manager Library (**liblvm.a**)

Syntax

```
#include <lvm.h>

int lvm_deletepv (PV_ID, VG_ID)
struct unique_id *VG_ID;
struct unique_id *PV_ID;
```

Description

The **lvm_deletepv** subroutine deletes the physical volume specified by the *PV_ID* parameter from its volume group. The *VG_ID* parameter indicates the volume group that contains the physical volume to be deleted. The physical volume must not contain any partitions of a logical volume, or the **LVM_PARTFND** error code is returned. In this case, the user must delete logical volumes or relocate the partitions that reside on the physical volume. The volume group containing the physical volume to be deleted must be varied on or an error is returned.

Note: You must have root user authority to use this subroutine.

Parameters

PV_ID	Specifies the physical volume to be deleted.
VG_ID	Specifies the volume group that contains the physical volume to be deleted.

Return Values

The **lvm_deletepv** subroutine returns one of the following values upon successful completion:

LVM_SUCCESS	The physical volume was successfully deleted.
LVM_VGDELETED	The physical volume was successfully deleted, and the volume group was also deleted because that physical volume was the last one in the volume group.

Error Codes

If the **lvm_deletepv** subroutine does not complete successfully, it returns one of the following values:

LVM_ALLOCERR	A memory allocation error occurred.
LVM_BELOW_QRMCNT	The physical volume could not be removed or deleted because there would no longer be a quorum of available physical volumes.
LVM_DALVOPN	The descriptor area logical volume could not be opened.
LVM_FORCEOFF	The volume group has been forcefully varied off due to a loss of quorum.

LVM_INVALID_PARAM	An invalid parameter was passed into the routine.
LVM_INVCONFIG	An error occurred while attempting to configure this volume group into the kernel. This error will normally result if the module ID is invalid, if the major number given is already in use, or if the volume group device could not be opened.
LVM_INV_DEVENT	The physical volume specified has an invalid device entry and cannot be checked to determine if it is raw.
LVM_LVMRECERR	The Logical Volume Manager record could not be read or written.
LVM_MAPFBSY	The volume group is currently locked because system management on the volume group is being done by another process.
LVM_MAPFOPN	The mapped file, which contains a copy of the volume group descriptor area used for making changes to the volume group, could not be opened.
LVM_MAPFRDWR	An error occurred while trying to read or write the mapped file.
LVM_NOTCHARDEV	The physical volume to be deleted does not have a raw device entry.
LVM_OFFLINE	The volume group which contains the physical volume to be deleted is off–line and should be on–line.
LVM_PARTFND	This routine cannot delete the specified physical volume because it contains physical partitions allocated to a logical volume.

Implementation Specifics

This subroutine is part of Base Operating System (BOS) Runtime.

Related Information

The **lvm_deletelv** subroutine, **lvm_migratepp** subroutine, **lvm_queryvg** subroutine, **lvm_reducelv** subroutine, **lvm_varyonvg** subroutine.

List of Logical Volume Subroutines and Logical Volume Programming Overview in *AIX General Programming Concepts : Writing and Debugging Programs*.

lvm_extendlv Subroutine

Purpose

Extends a logical volume by a specified number of partitions.

Library

Logical Volume Manager Library (**liblvm.a**)

Syntax

```
#include <lvm.h>

int lvm_extendlv (LV_ID, ExtendLV)
struct Lv_id *LV_ID;
struct ext_redlv *ExtendLV;
```

Description

Note: You must have root user authority to use this subroutine.

The **lvm_extendlv** subroutine extends a logical volume specified by the *LV_ID* parameter by adding a completely new logical partition or by adding another copy to an existing logical partition.

The **ext_redlv** structure pointed to by the *ExtendLV* parameter is defined in the **lvm.h** file and contains the following fields:

```
struct ext_redlv{
     long      size;
     struct pp *parts;
   }
struct pp {
     struct unique_id pv_id;
     long             lp_num;
     long             pp_num;
   }
```

Field	Description
`parts`	Points to an array of **pp** structures. The `parts` array should have one entry for each physical partition being allocated. The `parts` field is in the **ext_redlv** structure.
`size`	Specifies the number of entries in the array pointed to by the `parts` variable. The `parts` array should have one entry for each physical partition being allocated, and the `size` field should reflect a total of these entries. The `size` field should never be 0; if it is, an error will be returned. The `size` field is in the **ext_redlv** structure.
`lp_num`	Indicates the number of the logical partition that you are extending. The `lp_numb` value must range from 1 to the maximum number of logical partitions allowed in the logical volume being extended. The maximum number of logical partitions allowed on the logical volume is the `maxsize` field returned from a query of the logical volume, and must range from 1 to **LVM_MAXLPS**. The `lp_num` field is in the **pp** structure.

Field	Description
`pv_id`	Contains the valid ID of a physical volume that is a member of the same volume group as the logical volume being extended. The volume group should be varied on, or an error is returned. The `pp_id` field is in the **pp** structure.
`pp_num`	Specifies the number of the physical partition to be allocated as a copy of the logical partition. This number must range from 1 to the number of physical partitions allowed on the physical volume specified by the `pv_id` field. (The `pp_count` field returned from a query of the physical volume. This field ranges from 1 to **LVM_MAXPPS**). The physical partition specified by the `pp_num` should have a state of **LVM_PPFREE** (that is, should not be allocated). The `pp_num` field is in the **pp** structure.

An example of a correct **parts** array and **size** value follows:

```
size = 4 (The size field is set to 4 because there are 4 struct
            pp entries.)
    parts:
     entry1   pv_id = 4321
              lp_num = 2
              pp_num = 1
     entry2   pv_id = 1234
              lp_num = 2
              pp_num = 3
     entry3   pv_id = 5432
              lp_num = 3
              pp_num = 5
     entry4   pv_id = 4242
              lp_num = 2
              pp_num = 12
```

Up to three copies (physical partitions) can be allocated to the same logical partition. An error is returned if an attempt is made to add more. It is also possible to have entries with a valid `lp_num` field and zeroes for the `pv_id` and `pp_num` fields; this type of entry specifies that this logical partition should be ignored (nothing will be allocated for the logical partition). Another way to have a logical partition ignored is simply to skip an entry for it.

```
EXAMPLE 1
  size = 2
  parts:
         entry1    pv_id  = 0  (Entry 1 would indicate that lp 3
                   lp_num = 3    should be ignored.)
                   pp_num = 0
         entry2    pv_id = 4467
                   lp_num = 5
                   pp_num = 3
EXAMPLE 2
  size = 3
  parts:
         entry1    pv_id = 5347
                   lp_num = 1
                   pp_num = 1
         entry2    pv_id = 8790
                   lp_num = 3
                   pp_num = 3
         entry3    pv_id = 2938
                   lp_num = 6
                   pp_num = 6
```

Logical partition numbers 2, 4, and 5 are ignored since there were no entries for them in the array.

Parameters

ExtendLV	Points to the **ext_redlv** structure.
LV_ID	Points to the **lv_id** structure, which specifies the logical volume to extend.

Return Values

The **lvm_extendlv** subroutine returns a value of 0 upon successful completion.

Error Codes

If the **lvm_extendlv** subroutine fails, it returns one of the following values:

LVM_ALLOCERR	A memory allocation error occurred.
LVM_DALVOPN	The logical volume reserved by the volume group could not be opened.
LVM_FORCEOFF	The volume group has been forcefully varied off due to a loss of quorum.
LVM_INRESYNC	The logical partition to be extended is being resynced and cannot be extended while the resync is in progress.
LVM_INVALID_MIN_NUM	An invalid minor number was received.
LVM_INVALID_PARAM	One or both of the *ExtendLV* or *LV_ID* parameters are invalid, or the *LV_ID* parameter is not a valid logical volume. This could also mean that one of the fields in the **ext_redlv** structure is not valid.
LVM_INVCONFIG	An error occurred while attempting to configure this volume group into the kernel. This error will normally result if the major number in the mapped file is not valid.
LVM_INV_DEVENT	The device entry for the physical volume is not valid and cannot be checked to determine if it is raw.
LVM_LPNUM_INVAL	A logical partition number passed in is not valid.
LVM_MAPFBSY	The volume group is currently locked because system management on the volume group is being done by another process.
LVM_MAPFOPN	The mapped file, which contains a copy of the volume group descriptor area used for making changes to the volume group, could not be opened.
LVM_MAPFRDWR	An error occurred while trying to read or write the mapped file.
LVM_NOALLOCLP	The specified logical partition already has three copies.
LVM_NOTCHARDEV	The specified device is not a raw or character device.
LVM_OFFLINE	The volume group is offline and should be online.
LVM_PPNUM_INVAL	A physical partition number passed in is not valid.
LVM_PVSTATE_INVAL	A physical volume ID sent in specifies a physical volume with a state of **LVM_PVNOALLOC**.

Implementation Specifics

This subroutine is part of Base Operating System (BOS) Runtime.

Related Information

The **lvm_changelv** subroutine, **lvm_createlv** subroutine, **lvm_reducelv** subroutine, **lvm_varyonvg** subroutine.

List of Logical Volume Subroutines and Logical Volume Programming Overview in *AIX General Programming Concepts : Writing and Debugging Programs*.

lvm_installpv Subroutine

Purpose

Installs a physical volume into a volume group.

Library

Logical Volume Manager Library (**liblvm.a**)

Syntax

```
#include <lvm.h>

int lvm_installpv (InstallPV)
struct installpv *InstallPV;
```

Description

Note: You must have root user authority to use this subroutine.

The **lvm_installpv** subroutine installs a physical volume into a specified volume group. The physical volume must not exist in another volume group.

The **installpv** structure pointed to by the *InstallPV* parameter is found in the **lvm.h** file and is defined as follows:

```
struct installpv
{
  char *pvname;
  struct unique_id vg_id;
  short int override;
  struct unique_id out_vg_id;
};
```

Field	Description
`pvname`	Specifies the character special file name, which can be either a full path name or a single file name that resides in the /**dev** directory (for example, **rhdisk0**) of the physical volume being installed into the volume group specified by the `vg_id` field. The `pvname` field must be a null–terminated string that ranges from 1 to **LVM_NAMESIZ** bytes, including the null byte, and must be the name of a raw character device. If a raw device is not specified for the `pvname` field, the Logical Volume Manager (LVM) will add an `r` to the file name in order to have a raw device name. If there is no raw device entry for this name, the LVM returns an **LVM_NOTCHARDEV** error code.
`override`	Specifies whether or not the **LVM_VGMEMBER** error code should be ignored. If the `override` field is TRUE, the LVM installs the physical volume into the specified volume group even if the physical volume is a member of another volume group. This is done only if the other volume group is not varied on. If it is varied on, an **LVM_MEMACTVVG** error code is returned. If the `override` field is FALSE, an **LVM_VGMEMBER** error code is returned if the physical volume belongs to another volume group, whether that volume group is varied on or varied off. The **LVM_ALRDYMEM** error code is returned if the physical volume is already a member of the specified volume group. This error is returned regardless of the setting of the `override` field.
`out_vg_id`	Contains the ID of the volume group that the physical volume is a member of. If either the **LVM_MEMACTVVG** or **LVM_VGMEMBER** error code is returned.

Each physical volume installed into a volume group contains a volume group descriptor area in the reserved area at the beginning of the physical volume. The volume group descriptor area contains information about the physical and logical volumes in the volume group. This descriptor area is used by the LVM to manage the logical and physical volumes in the volume group.

Parameters

InstallPV	Points to the **installpv** structure.

Return Values

The **lvm_installpv** subroutine returns a value of 0 upon successful completion.

Error Codes

If the **lvm_installpv** subroutine fails, it returns one of the following values:

LVM_ALLOCERR	A memory allocation error occurred.
LVM_ALRDYMEM	The physical volume is already a member of the specified volume group.
LVM_BADBBDIR	The physical volume could not be installed into the volume group because the bad block directory could not be read from or written to.
LVM_DALVOPN	The logical volume reserved by the volume group could not be opened.
LVM_INVALID_PARAM	An invalid parameter was passed into the routine.
LVM_INV_DEVENT	A device entry is invalid and cannot be checked to determine if it is raw.
LVM_LVMRECERR	The LVM record, which contains information about the volume group descriptor area, could not be read or written.
LVM_MAPFOPN	The mapped file, which contains a copy of the volume group descriptor area used for making changes to the volume group, could not be opened.
LVM_MAPFRDWR	An error occurred while trying to write to the mapped file.
LVM_MEMACTVVG	The physical volume specified is a member of another volume group that is varied on. This error is returned when the `override` field is TRUE.
LVM_NOTCHARDEV	The device specified is not a raw or character device.
LVM_OFFLINE	The volume group specified is offline. It must be varied on to perform this operation.
LVM_PVMAXERR	The physical volume cannot be installed into the specified volume group because the maximum allowed number of physical volumes are already installed in the volume group. The maximum number of physical volumes is **LVM_MAXPVS**.
LVM_PVOPNERR	The physical volume device could not be opened.
LVM_RDPVID	The record which contains the physical volume ID could not be read.
LVM_VGDASPACE	The physical volume cannot be installed into the specified volume group because there is not enough space in the volume group descriptor area to add a description of the physical volume and its partitions.

LVM_VGMEMBER	The physical volume cannot be installed into the specified volume group because its LVM record indicates it is already a member of another volume group. If the caller feels that the information in the LVM record is incorrect, the `override` field can be set to TRUE in order to override this error. This error is only returned when the `override` field is set to FALSE.
LVM_WRTDAERR	An error occurred while trying to initialize either the Volume Group Descriptor Area, the Volume Group Status Area, or the Mirror–Write Consistency Cache Area on the physical volume.

Implementation Specifics

This subroutine is part of Base Operating System (BOS) Runtime.

Related Information

The **lvm_varyonvg** subroutine.

List of Logical Volume Subroutines and Logical Volume Programming Overview in *AIX General Programming Concepts : Writing and Debugging Programs*.

lvm_migratepp Subroutine

Purpose

Moves a physical partition to a specified physical volume.

Library

Logical Volume Manager Library (**liblvm.a**)

Syntax

```
#include <lvm.h>

int lvm_migratepp (MigratePP)
struct migratepp *MigratePP;
```

Description

Note: You must have root user authority to use this subroutine.

The **lvm_migratepp** subroutine moves the physical partition specified by the `oldpp_num` field from the physical volume specified by the `oldpv_id` field to the physical partition, the `newpp_num` field, located on the physical volume given in the `newpv_id` field. The `vg_id` field specifies the volume group that contains both the old physical volume and the new physical volume. This volume group should be varied on, or an error is returned.

The **migratepp** structure pointed to by the *MigratePP* parameter is defined in the **lvm.h** file and contains the following fields:

```
struct migratepp{
       struct unique_id vg_id;
       long    oldpp_num;
       long    newpp_num;
       struct unique_id oldpv_id;
       struct unique_id newpv_id;
   }
```

Migration with Two Physical Copies

If the logical partition to which the old physical partition is allocated has two physical copies, the migration takes place in the following sequence:

1. Extend the logical partition to add the new physical partition copy.
2. Resynchronize the logical partition in an attempt to make the new physical partition non–stale.
3. Reduce the logical partition to delete the old physical partition copy.

For the migration to complete successfully, it is not necessary for the resynchronization phase to complete successfully. However, it is always necessary that each logical partition have at least one good physical copy.

If the phase 1 extension of the new physical partition fails, you will receive the error code from the extension.

In general, if the extension in phase 1 succeeds, the migration will usually be successful. The migration might not be successful even if the phase 1 extension is successful when the old physical partition being migrated from is the only good physical copy of the logical partition. If the phase 2 resynchronization fails, and the phase 3 reduction fails because the old partition is still the only good physical copy of the logical partition, an **LVM_MIGRATE_FAIL** error code is returned.

It is very unlikely for the phase 3 reduction to fail, but failure is possible if an error occurs, such as being unable to allocate memory in the kernel due to a lack of system resources.

If the phase 2 resynchronization fails, but the phase 3 reduction of the old partition is successful, you will receive the **LVM_RESYNC_FAILED** return code to indicate the migration was successful, but the resynchronization of the logical partition was not.

If the phase 2 resynchronization completes successfully, the migration is successful. The **LVM_SUCCESS** return code is returned whether or not the phase 3 reduction of the old physical partition is successful.

Migration with Three Physical Copies

If the logical partition to which the old physical partition is allocated has three physical copies, the migration will take place in the following sequence:

1. Reduce the logical partition to delete the old physical partition copy.
2. Extend the logical partition to add the new physical partition copy.
3. Resynchronize the logical partition.

If the phase 1 reduction of the old physical partition fails, you will receive the error code from the reduction. If the reduction fails because the old partition is the only good physical copy of the logical partition, an **LVM_INVLPRED** error code is returned. In this case, you should attempt to resynchronize the logical partition in question. If the resynchronization succeeds, you should attempt the migration again.

In order for the migration to be successful, both the phase 1 reduction and the phase 2 extension must be successful. If the phase 2 extension fails, an attempt will be made to extend and add back the old physical partition. If the old physical partition can be added back and the logical partition is back to its original configuration, you will receive the **LVM_MIGRATE_FAIL** error code to indicate that the migration failed. If the old partition cannot be added back, you will receive an **LVM_LOSTPP** error code to indicate that a physical partition copy has been lost and the logical partition does not have its original number of copies. It is not very likely for either of the extensions described above to fail, but it is possible to have a failure due to an error such as being unable to allocate memory in the kernel due to a lack of system resources.

If the phase 2 extension completes successfully, the migration is successful. If the phase 3 resynchronization completes successfully, you will receive a return code of **LVM_SUCCESS**. If the resynchronization is not successful, you will receive the **LVM_RESYNC_FAILED** error code to indicating that although the migration was successful, the resynchronization of the logical partition was not.

Parameters

MigratePP	Points to the **migratepp** structure.

Return Values

When successful, the **lvm_migratepp** subroutine returns the following return code:

LVM_RESYNC_FAILED	The migrate succeeded, but all physical copies of the logical partition could not be resynchronized.

Error Codes

If the **lvm_migratepp** subroutine fails, it returns one of the following values:

LVM_ALLOCERR	A memory allocation error occurred.
LVM_DALVOPN	The logical volume reserved by the volume group could not be opened.
LVM_FORCEOFF	The volume group has been forcefully varied off due to a loss of quorum.

LVM_INRESYNC	The physical partition being migrated is allocated to a logical partition that is being resynced. The migration cannot be completed while the resync is in progress.
LVM_INVALID_MIN_NUM	A minor number that is not valid was received.
LVM_INVALID_PARAM	One of the parameters passed in did not have a valid value.
LVM_INVCONFIG	An error occurred while attempting to configure this volume group into the kernel. This error will normally result if the module ID is invalid, if the major number given is already in use, or if the volume group device could not be opened.
LVM_INV_DEVENT	A device has a major number that does not correspond to the volume group being worked in.
LVM_INVLPRED	A reduction was requested that would leave a logical partition with no good copies.
LVM_LOSTPP	The migration failed and the logical partition could not be restored to its original configuration.
LVM_MAPFBSY	The volume group is currently locked because system management on the volume group is being done by another process.
LVM_MAPFOPN	The mapped file, which contains a copy of the volume group descriptor area used for making changes to the volume group, could not be opened.
LVM_MAPFRDWR	An error occurred while trying to read or write the mapped file.
LVM_MIGRATE_FAIL	The migration failed because the requested move would leave the logical partition without a good physical copy.
LVM_NOTCHARDEV	A device is not a raw or character device.
LVM_NOTSYNCED	The resync involving the physical partitions of the **migratepp** call was not complete.
LVM_OFFLINE	The volume group is offline and should be online.

Implementation Specifics

This subroutine is part of Base Operating System (BOS) Runtime.

Related Information

The **lvm_querypv** subroutine, **lvm_varyonvg** subroutine.

List of Logical Volume Subroutines and Logical Volume Programming Overview in *AIX General Programming Concepts : Writing and Debugging Programs*.

lvm_querylv Subroutine

Purposes

Queries a logical volume and returns all pertinent information.

Library

Logical Volume Manager Library (**liblvm.a**)

Syntax

```
#include <lvm.h>

int lvm_querylv (LV_ID, QueryLV, PVName)
struct lv_id *LV_ID;
struct querylv **QueryLV;
char *PVName;
```

Description

Note: You must have root user authority to use this subroutine.

The **lvm_querylv** subroutine returns information for the logical volume specified by the *LV_ID* parameter.

The **querylv** structure, found in the **lvm.h** file, is defined as follows:

```
struct querylv {
      char lvname[LVM_NAMESIZ];
      struct unique_id vg_id;
      long maxsize;
      long mirror_policy;
      long lv_state;
      long currentsize;
      long ppsize;
      long permissions;
      long bb_relocation;
      long write_verify;
      long mirwrt_consist;
      long open_close;
      struct pp *mirrors[LVM_NUMCOPIES]
}
struct pp {
      struct unique_id pv_id;
      long lp_num;
      long pp_num;
      long ppstate;
 }
```

Field	Description
`lv_state`	Specifies the current state of the logical volume and can have any of the following bit–specific values ORed together: **LVM_LVDEFINED** The logical volume is defined. **LVM_LVSTALE** The logical volume contains stale partitions.
`currentsize`	Indicates the current size in logical partitions of the logical volume. The size, in bytes, of every physical partition is 2 to the power of the `ppsize` field.
`ppsize`	Specifies the size of the physical partitions of all physical volumes in the volume group.

Field	Description
`permissions`	Specifies the permission assigned to the logical volume and can be one of the following values: **LVM_RDONLY** Access to this logical volume is read only. **LVM_RDWR** Access to this logical volume is read/write.
`bb_relocation`	Specifies if bad block relocation is desired and is one of the following values: **LVM_NORELOC** Bad blocks will not be relocated. **LVM_RELOC** Bad blocks will be relocated.
`write_verify`	Specifies if write verification for the logical volume is desired and returns one of the following values: **LVM_NOVERIFY** Write verification is not performed for this logical volume. **LVM_VERIFY** Write verification is performed on all writes to the logical volume.
`mirwrt_consist`	Indicates whether mirror–write consistency recovery will be performed for this logical volume. The LVM always insures data consistency among mirrored copies of a logical volume during normal I/O processing. For every write to a logical volume, the LVM generates a write request for every mirror copy. A problem arises if the system crashes in the middle of processing a mirrored write (before all copies are written). If mirror write consistency recovery is requested for a logical volume, the LVM keeps additional information to allow recovery of these inconsistent mirrors. Mirror write consistency recovery should be performed for most mirrored logical volumes. Logical volumes, such as page space, that do not use the existing data when the volume group is re–varied on do not need this protection. Values for the `mirwrt_consist` field are: **LVM_CONSIST** Mirror–write consistency recovery will be done for this logical volume. **LVM_NOCONSIST** Mirror–write consistency recovery will not be done for this logical volume.
`open_close`	Specifies if the logical volume is opened or closed. Values for this field are: **LVM_QLV_NOTOPEN** The logical volume is closed. **LVM_QLVOPEN** The logical volume is opened by one or more processes.
`mirrors`	Specifies an array of pointers to partition map lists (physical volume id, logical partition number, physical partition number, and physical partition state for each copy of the logical partitions for the logical volume). The `ppstate` field can be **LVM_PPFREE**, **LVM_PPALLOC**, or **LVM_PPSTALE**. If a logical partition does not contain any copies, its `pv_id`, `lp_num`, and `pp_num` fields will contain zeros.

All other fields are described in the **lvm_createlv** subroutine.

The *PVName* parameter enables the user to query from a volume group descriptor area on a specific physical volume instead of from the Logical Volume Manager's (LVM) most recent, in–memory copy of the descriptor area. This method should only be used if the volume group is varied off.

Note: The data returned is not guaranteed to be the most recent or correct, and it can reflect a back–level descriptor area.

The *PVName* parameter should specify either the full path name of the physical volume that contains the descriptor area to query, or a single file name that must reside in the /**dev** directory (for example, **rhdisk1**). This parameter must be a null–terminated string between 1 and **LVM_NAMESIZ** bytes, including the null byte, and must represent a raw device entry. If a raw or character device is not specified for the *PVName* parameter, the LVM adds an **r** to the file name to have a raw device name. If there is no raw device entry for this name, the LVM returns the **LVM_NOTCHARDEV** error code.

If a *PVName* parameter is specified, only the `minor_num` field of the *LV_ID* parameter need be supplied. The LVM fills in the `vg_id` field and returns it to the user. If the user wishes to query from the LVM's in–memory copy, the *PVName* parameter should be set to null. When using this method of query, the volume group must be varied on, or an error is returned.

Note: As long as the *PVName* parameter is not null, the LVM will attempt a query from a physical volume and *not* from its in–memory copy of data.

In addition to the *PVName* parameter, the caller passes the ID of the logical volume to be queried (*LV_ID* parameter) and the address of a pointer to the **querylv** structure, specified by the *QueryLV* parameter. The LVM separately allocates the space needed for the **querylv** structure and the struct **pp** arrays, and returns the **querylv** structure's address in the pointer variable passed in by the user. The user is responsible for freeing the space by first freeing the struct **pp** pointers in the **mirrors** array and then freeing the **querylv** structure.

Parameters

LV_ID	Points to an **lv_id** structure that specifies the logical volume to query.
QueryLV	Contains the address of a pointer to the **querylv** structure.
PVName	Names the physical volume from which to use the volume group descriptor for the query. This parameter can also be null.

Return Values

If the **lvm_querylv** subroutine is successful, it returns a value of 0.

Error Codes

If the **lvm_querylv** subroutine does not complete successfully, it returns one of the following values:

LVM_ALLOCERR	The subroutine could not allocate enough space for the complete buffer.
LVM_INVALID_MIN_NUM	The minor number of the logical volume is not valid.
LVM_INVALID_PARAM	A parameter passed into the routine is not valid.
LVM_INV_DEVENT	The device entry for the physical volume specified by the *Pvname* parameter is not valid and cannot be checked to determine if it is raw.
LVM_NOTCHARDEV	The physical volume name given does not represent a raw or character device.
LVM_OFFLINE	The volume group containing the logical volume to query was offline. If the query originates from the varied–on volume group's current volume group descriptor area, one of the following error codes is returned:

LVM_DALVOPN	The volume group reserved logical volume could not be opened.
LVM_MAPFBSY	The volume group is currently locked because system management on the volume group is being done by another process.
LVM_MAPFOPN	The mapped file, which contains a copy of the volume group descriptor area used for making changes to the volume group, could not be opened.
LVM_MAPFRDWR	The mapped file could not be read or written.

If a physical volume name has been passed, requesting that the query originate from a specific physical volume, one of the following error codes is returned:

LVM_BADBBDIR	The bad–block directory could not be read or written.
LVM_LVMRECERR	The LVM record, which contains information about the volume group descriptor area, could not be read.
LVM_NOPVVGDA	There are no volume group descriptor areas on the physical volume specified.
LVM_NOTVGMEM	The physical volume specified is not a member of a volume group.
LVM_PVDAREAD	An error occurred while trying to read the volume group descriptor area from the specified physical volume.
LVM_PVOPNERR	The physical volume device could not be opened.
LVM_VGDA_BB	A bad block was found in the volume group descriptor area located on the physical volume that was specified for the query. Therefore, a query cannot be done from the specified physical volume.

Implementation Specifics

This subroutine is part of Base Operating System (BOS) Runtime.

Related Information

The **lvm_varyonvg** subroutine, **lvm_createlv** subroutine.

List of Logical Volume Subroutines and Logical Volume Programming Overview in *AIX General Programming Concepts : Writing and Debugging Programs*.

lvm_querypv Subroutine

Purpose

Queries a physical volume and returns all pertinent information.

Library

Logical Volume Manager Library (**liblvm.a**)

Syntax

```
#include <lvm.h>

int lvm_querypv (VG_ID, PV_ID, QueryPV, PVName)
struct unique_id *VG_ID;
struct unique_id *PV_ID;
struct querypv **QueryPV;
char *PVName;
```

Description

Note: You must have root user authority to use the **lvm_querypv** subroutine.

The **lvm_querypv** subroutine returns information on the physical volume specified by the *PV_ID* parameter.

The **querypv** structure, defined in the **lvm.h** file, contains the following fields:

```
struct querypv {
      long ppsize;
      long pv_state;
      long pp_count;
      long alloc_ppcount;
      struct pp_map *pp_map;
      long pvnum_vgdas;
 }
 struct pp_map {
     long pp_state;
      struct lv_id lv_id;
      long lp_num;
      long copy;
      struct unique_id fst_alt_vol;
      long fst_alt_part;
      struct unique_id snd_alt_vol;
      long snd_alt_part;
  }
```

Field	Description
`ppsize`	Specifies the size of the physical partitions, which is the same for all partitions within a volume group. The size in bytes of a physical partition is 2 to the power of `ppsize`.
`pv_state`	Contains the current state of the physical volume.
`pp_count`	Contains the total number of physical partitions on the physical volume.
`alloc_ppcount`	Contains the number of allocated physical partitions on the physical volume.

Field	Description
`pp_map`	Points to an array that has entries for each physical partition of the physical volume. Each entry in this array will contain the `pp_state` that specifies the state of the physical partition (**LVM_PPFREE**, **LVM_PPALLOC**, or **LVM_PPSTALE**) and the `lv_id`, field, the ID of the logical volume that it is a member of. The `pp_map` array also contains the physical volume IDs (`fst_alt_vol` and `snd_alt_vol`) and the physical partition numbers (`fst_alt_part` and `snd_alt_part)` for the first and second alternate copies of the physical partition, and the logical partition number (`lp_num)` that the physical partition corresponds to. If the physical partition is free (that is, not allocated), *all* of its **pp_map** fields will be zero. `fst_alt_vol` — Contains zeros if the logical partition has only one physical copy. `fst_alt_part` — Contains zeros if the logical partition has only one physical copy. `snd_alt_vol` — Contains zeros if the logical partition has only one or two physical copies. `snd_alt_part` — Contains zeros if the logical partition has only one or two physical copies. `copy` — Specifies which copy of a logical partition this physical partition is allocated to. This field will contain one of the following values: **LVM_PRIMARY** — Primary and only copy of a logical partition **LVM_PRIMOF2** — Primary copy of a logical partition with two physical copies **LVM_PRIMOF3** — Primary copy of a logical partition with three physical copies **LVM_SCNDOF2** — Secondary copy of a logical partition with two physical copies **LVM_SCNDOF3** — Secondary copy of a logical partition with three physical copies **LVM_TERTOF3** — Tertiary copy of a logical partition with three physical copies.
`pvnum_vgdas`	Contains the number of volume group descriptor areas (0, 1, or 2) that are on the specified physical volume.

The *PVName* parameter enables the user to query from a volume group descriptor area on a specific physical volume instead of from the Logical Volume Manager's (LVM) most recent, in–memory copy of the descriptor area. This method should only be used if the volume group is varied off. The data returned is not guaranteed to be most recent or correct, and it can reflect a back level descriptor area.

The *PVname* parameter should specify either the full path name of the physical volume that contains the descriptor area to query or a single file name that must reside in the /**dev** directory (for example, **rhdisk1**). This field must be a null–terminated string of from 1 to

LVM_NAMESIZ bytes, including the null byte, and represent a raw or character device. If a raw or character device is not specified for the *PVName* parameter, the LVM will add an **r** to the file name in order to have a raw device name. If there is no raw device entry for this name, the LVM will return the **LVM_NOTCHARDEV** error code. If a *PVName* is specified, the volume group identifier, `VG_ID`, will be returned by the LVM through the *VG_ID* parameter passed in by the user. If the user wishes to query from the LVM in–memory copy, the *PVName* parameter should be set to null. When using this method of query, the volume group must be varied on, or an error will be returned.

Note: As long as the *PVName* is not null, the LVM will attempt a query from a physical volume and *not* from its in–memory copy of data.

In addition to the *PVName* parameter, the caller passes the *VG_ID* parameter, indicating the volume group that contains the physical volume to be queried, the unique ID of the physical volume to be queried, the *PV_ID* parameter, and the address of a pointer of the type *QueryPV*. The LVM will separately allocate enough space for the **querypv** structure and the struct `pp_map` array and return the address of the **querypv** structure in the *QueryPV* pointer passed in. The user is responsible for freeing the space by freeing the struct *pp_map* pointer and then freeing the *QueryPV* pointer.

Parameters

VG_ID	Points to a **unique_id** structure that specifies the volume group of which the physical volume to query is a member.
PV_ID	Points to a **unique_id** structure that specifies the physical volume to query.
QueryPV	Specifies the address of a pointer to a **querypv** structure.
PVName	Names a physical volume from which to use the volume group descriptor area for the query. This parameter can be null.

Return Values

The **lvm_querypv** subroutine returns a value of 0 upon successful completion.

Error Codes

If the **lvm_querypv** subroutine fails it returns one of the following error codes:

LVM_ALLOCERR	The routine cannot allocate enough space for a complete buffer.
LVM_INVALID_PARAM	An invalid parameter was passed into the routine.
LVM_INV_DEVENT	The device entry for the physical volume is invalid and cannot be checked to determine if it is raw.
LVM_OFFLINE	The volume group specified is offline and should be online.

If the query originates from the varied–on volume group's current volume group descriptor area, one of the following error codes may be returned:

LVM_DALVOPN	The volume group reserved logical volume could not be opened.
LVM_MAPFBSY	The volume group is currently locked because system management on the volume group is being done by another process.

LVM_MAPFOPN	The mapped file, which contains a copy of the volume group descriptor area used for making changes to the volume group, could not be opened.
LVM_MAPFRDWR	Either the mapped file could not be read, or it could not be written.

If a physical volume name has been passed, requesting that the query originate from a specific physical volume, then one of the following error codes may be returned:

LVM_BADBBDIR	The bad–block directory could not be read or written.
LVM_LVMRECERR	The LVM record, which contains information about the volume group descriptor area, could not be read.
LVM_NOPVVGDA	There are no volume group descriptor areas on this physical volume.
LVM_NOTCHARDEV	A device is not a raw or character device.
LVM_NOTVGMEM	The physical volume is not a member of a volume group.
LVM_PVDAREAD	An error occurred while trying to read the volume group descriptor area from the specified physical volume.
LVM_PVOPNERR	The physical volume device could not be opened.
LVM_VGDA_BB	A bad block was found in the volume group descriptor area located on the physical volume that was specified for the query. Therefore, a query cannot be done from the specified physical volume.

Implementation Specifics

This subroutine is part of Base Operating System (BOS) Runtime.

Related Information

The **lvm_varyonvg** subroutine.

List of Logical Volume Subroutines and Logical Volume Programming Overview in *AIX General Programming Concepts : Writing and Debugging Programs*.

lvm_queryvg Subroutine

Purpose

Queries a volume group and returns pertinent information.

Library

Logical Volume Manager Library (**liblvm.a**)

Syntax

```
#include <lvm.h>

int lvm_queryvg (VG_ID, QueryVG, PVName)
struct unique_id *VG_ID;
struct queryvg **QueryVG;
char *PVName;
```

Description

Note: You must have root user authority to use this subroutine.

The **lvm_queryvg** subroutine returns information on the volume group specified by the *VG_ID* parameter.

The **queryvg** structure , found in the **lvm.h** file, contains the following fields:

```
struct queryvg {
      long maxlvs;
      long ppsize;
      long freespace;
      long num_lvs;
      long num_pvs;
      long total_vgdas;
      struct lv_array *lvs;
      struct pv_array *pvs;
 }
 struct pv_array {
     struct unique_id pv_id;
     long pvnum_vgdas;
     char   state;
     char   res[3];
 }
 struct lv_array {
      struct lv_id   lv_id;
      char  lvname[LVM_NAMESIZ];
      char   state;
      char   res[3];
 }
```

Field	Description
maxlvs	Specifies the maximum number of logical volumes allowed in the volume group.
ppsize	Specifies the size of all physical partitions in the volume group. The size in bytes of each physical partitions is 2 to the power of the ppsize field.
freespace	Contains the number of free physical partitions in this volume group.
num_lvs	Indicates the number of logical volumes.
num_pvs	Indicates the number of physical volumes.
total_vgdas	Specifies the total number of volume group descriptor areas for the entire volume group.

Field	Description
`lvs`	Points to an array of unique IDs, names, and states of the logical volumes in the volume group.
`pvs`	Points to an array of unique IDs, states, and the number of volume group descriptor areas for each of the physical volumes in the volume group.

The *PVName* parameter enables the user to query from a descriptor area on a specific physical volume instead of from the Logical Volume Manager's (LVM) most recent, in–memory copy of the descriptor area. This method should only be used if the volume group is varied off. The data returned is *not guaranteed* to be most recent or correct, and it can reflect a back level descriptor area. The *Pvname* parameter should specify either the full path name of the physical volume that contains the descriptor area to query or a single file name that must reside in the /**dev** directory (for example, **rhdisk1**). The name must represent a raw device. If a raw or character device is not specified for the *PVName* parameter, the Logical Volume Manager will add an `r` to the file name in order to have a raw device name. If there is no raw device entry for this name, the LVM returns the **LVM_NOTCHARDEV** error code. This field must be a null–terminated string of from 1 to **LVM_NAMESIZ** bytes, including the null byte. If a *PVName* is specified, the LVM will return the *VG_ID* to the user through the *VG_ID* pointer passed in. If the user wishes to query from the LVM in–memory copy, the *PVName* parameter should be set to null. When using this method of query, the volume group must be varied on, or an error will be returned.

Note: As long as the *PVName* parameter is not null, the LVM will attempt a query from a physical volume and *not* its in–memory copy of data.

In addition to the *PVName* parameter, the caller passes the unique ID of the volume group to be queried (*VG_ID*) and the address of a pointer to a **queryvg** structure. The LVM will separately allocate enough space for the **queryvg** structure, as well as the **lv_array** and **pv_array** structures, and return the address of the completed structure in the *QueryVG* parameter passed in by the user. The user is responsible for freeing the space by freeing the `lv` and `pv` pointers and then freeing the *QueryVG* pointer.

Parameters

VG_ID	Points to a **unique_id** structure that specifies the volume group to be queried.
QueryVG	Specifies the address of a pointer to the **queryvg** structure.
PVName	Specifies the name of the physical volume that contains the descriptor area to query and must be the name of a raw device.

Return Values

The **lvm_queryvg**n subroutine returns a value of 0 upon successful completion.

Error Codes

If the **lvm_queryvg** subroutine fails it returns one of the following error codes:

LVM_ALLOCERR	The subroutine cannot allocate enough space for a complete buffer.
LVM_FORCEOFF	The volume group has been forcefully varied off due to a loss of quorum.
LVM_INVALID_PARAM	An invalid parameter was passed into the routine.
LVM_OFFLINE	The volume group is offline and should be online.

If the query originates from the varied–on volume group's current volume group descriptor area, one of the following error codes may be returned:

LVM_DALVOPN	The volume group reserved logical volume could not be opened.
LVM_INV_DEVENT	The device entry for the physical volume specified by the *PVName* parameter is invalid and cannot be checked to determine if it is raw.
LVM_MAPFBSY	The volume group is currently locked because system management on the volume group is being done by another process.
LVM_MAPFOPN	The mapped file, which contains a copy of the volume group descriptor area used for making changes to the volume group, could not be opened.
LVM_MAPFRDWR	Either the mapped file could not be read, or it could not be written.
LVM_NOTCHARDEV	A device is not a raw or character device.

If a physical volume name has been passed, requesting that the query originate from a specific physical volume, one of the following error codes may be returned:

LVM_BADBBDIR	The bad–block directory could not be read or written.
LVM_LVMRECERR	The LVM record, which contains information about the volume group descriptor area, could not be read.
LVM_NOPVVGDA	There are no volume group descriptor areas on this physical volume.
LVM_NOTVGMEM	The physical volume is not a member of a volume group.
LVM_PVDAREAD	An error occurred while trying to read the volume group descriptor area from the specified physical volume.
LVM_PVOPNERR	The physical volume device could not be opened.
LVM_VGDA_BB	A bad block was found in the volume group descriptor area located on the physical volume that was specified for the query. Therefore, a query cannot be done from this physical volume.

Implementation Specifics

This subroutine is part of Base Operating System (BOS) Runtime.

Related Information

The **lvm_varyonvg** subroutine.

List of Logical Volume Subroutines and Logical Volume Programming Overview in *AIX General Programming Concepts : Writing and Debugging Programs*.

lvm_queryvgs Subroutine

Purpose

Queries volume groups and returns information to online volume groups.

Library

Logical Volume Manager Library (**liblvm.a**)

Syntax

```
#include <lvm.h>

int lvm_queryvgs (QueryVGS, Kmid)
struct queryvgs **QueryVGS;
mid_t Kmid;
```

Description

Note: You must have root user authority to use this subroutine.

The **lvm_queryvgs** subroutine returns the volume group IDs and major numbers for all volume groups in the system that are online.

The caller passes the address of a pointer to a **queryvgs** structure, and the Logical Volume Manager (LVM) allocates enough space for the structure and returns the address of the structure in the pointer passed in by the user. The caller also passes in a *Kmid* parameter, which identifies the entry point of the logical device driver module:

```
struct queryvgs {
        long num_vgs;
        struct {
        long major_num
        struct unique_id vg_id;
        } vgs  [LVM_MAXVGS];
 }
```

Field	Description
`num_vgs`	Contains the number of online volume groups on the system. The `vgs` is an array of the volume group IDs and major numbers of all online volume groups in the system.

Parameters

QueryVGS	Points to the **queryvgs** structure.
Kmid	Identifies the address of the entry point of the logical volume device driver module.

Return Values

The **lvm_queryvgs** subroutine returns a value of 0 upon successful completion.

Error Codes

If the **lvm_queryvgs** subroutine fails, it returns one of the following error codes:

LVM_ALLOCERR	The routine cannot allocate enough space for the complete buffer.
LVM_INVALID_PARAM	An invalid parameter was passed into the routine.
LVM_INVCONFIG	An error occurred while attempting to configure this volume group into the kernel. This error will normally result if the module ID is invalid, if the major number given is already in use, or if the volume group device could not be opened.

Implementation Specifics

This subroutine is part of Base Operating System (BOS) Runtime.

Related Information

The **lvm_varyonvg** subroutine.

List of Logical Volume Subroutines and Logical Volume Programming Overview in *AIX General Programming Concepts : Writing and Debugging Programs*.

lvm_reducelv Subroutine

Purpose

Reduces the size of a logical volume by a specified number of partitions.

Library

Logical Volume Manager Library (**liblvm.a**)

Syntax

```
#include <lvm.h>

int lvm_reducelv (LV_ID, ReduceLV)
struct lv_id *LV_ID;
struct ext_redlv *ReduceLV;
```

Description

Note: You must have root user authority to use this subroutine.

The **lvm_reducelv** subroutine reduces a logical volume specified by the *LV_ID* parameter. This logical volume should be closed and should be a member of an online volume group. On partial reductions of a logical volume, all remaining logical partitions must have one good (non–stale) copy allocated to them. The Logical Volume Manager (LVM) does not reduce the last good (non–stale) copy of a logical partition on partial reductions to a logical volume. If a reduction is refused for this reason, the resync routines can be used to make all stale copies of a logical partition good so that a reduction can then be performed.

The **ext_redlv** structure, pointed to by the *ReduceLV* parameter, is found in the **lvm.h** file and is defined as follows:

```
struct ext_redlv{
      long       size;
      struct pp *parts;
}
struct pp {
      struct unique_id pv_id;
      long        lp_num;
      long        pp_num;
}
```

Following is an example of a correct **parts** array and **size** value:

```
size = 4   (The size field is set to 4 because
      there are 4 struct pp entries.)
parts:
 entry1   pv_id = 4321
          lp_num = 2
          pp_num = 1
 entry2   pv_id = 1234
          lp_num = 2
          pp_num = 3
 entry3   pv_id = 5432
          lp_num = 3
          pp_num = 5
 entry4   pv_id = 4242
          lp_num = 2
          pp_num = 12
```

The *ReduceLV* parameter is a pointer to an **ext_redlv** structure. Within this structure is the `parts` field, which is a pointer to an array of **pp** structures. Also in the **ext_redlv** structure is the `size` field, which is the number of entries in the array that is pointed to by the `parts` field. The parts array should have one entry for each physical partition being deallocated, and the `size` field should reflect a total of these entries. Also, the `size` field should never be 0; if it is, an error code is returned.

Within the **pp** structure is a `lp_num` field which is the number of the logical partitions that you are reducing. This number should be between 1 and the value of the `maxsize` field. The `maxsize` field is returned from the **lvm_querylv** subroutine and is the maximum number of logical partitions allowed for a logical volume. Also in the **pp** structure are the `pp_num` and `pv_id` fields. The `pp_num` field is the number of the physical partition to be deallocated as a copy of the logical partition. This number must range from 1 to the value of the `pp_count` field. The `pp_count` field is returned from the **lvm_querypv** subroutine and is the maximum number of physical partitions allowed on a physical volume. Also, the physical partition specified by the `pp_num` field should have a state of **LVM_PPALLOC** (that is, should be allocated). The `pv_id` field should contain the valid ID of a physical volume that is a member of the same volume group as the logical volume being reduced.

Parameters

LV_ID	Specifies the logical volume to be reduced.
ReduceLV	Points to the **ext_redlv** structure.

Return Values

Upon successful completion, a value of 0 is returned.

Error Codes

If the **lvm_reducelv** subroutine does not complete successfully, it returns one of the following error codes:

LVM_ALLOCERR	A memory allocation error occurred.
LVM_DALVOPN	The volume group reserved logical volume could not be opened.
LVM_FORCEOFF	The volume group has been forcefully varied off due to a loss of quorum.
LVM_INVALID_MIN_NUM	A minor number received was not valid.
LVM_INVALID_PARAM	One of the parameters passed in is not valid, or one of the fields in the structures pointed to by one of the parameters is not valid.
LVM_INVCONFIG	An error occurred while attempting to configure this volume group into the kernel. This error will normally result if the module ID is not valid, the major number given is already in use, or the volume group device could not be opened.
LVM_INV_DEVENT	The device entry for the physical volume is not valid and cannot be checked to determine if it is raw.
LVM_INVLPRED	The reduction cannot be completed because a logical partition would exist with only stale copies remaining.
LVM_LPNUM_INVAL	A logical partition number passed in is not valid.
LVM_MAPFBSY	The volume group is currently locked because system management on the volume group is being done by another process.
LVM_MAPFOPN	The mapped file, which contains a copy of the volume group descriptor area used for making changes to the volume group, could not be opened.
LVM_MAPFRDWR	An error occurred while trying to read or write the mapped file.

LVM_NOTCHARDEV	The device specified is not a raw or character device.
LVM_OFFLINE	The volume group is offline and should be online.
LVM_PPNUM_INVAL	A physical partition number passed in is not valid.

Implementation Specifics

This subroutine is part of Base Operating System (BOS) Runtime.

Related Information

The **lvm_createlv** subroutine, **lvm_deletelv** subroutine, **lvm_extendlv** subroutine, **lvm_resynclp** subroutine, **lvm_resynclv** subroutine, **lvm_resyncpv** subroutine.

List of Logical Volume Subroutines and Logical Volume Programming Overview in *AIX General Programming Concepts : Writing and Debugging Programs*.

lvm_resynclp Subroutine

Purpose

Synchronizes all physical partitions for a logical partition.

Library

Logical Volume Manager Library (**liblvm.a**)

Syntax

```
#include <lvm.h>

int lvm_resynclp (LV_ID, LP_Num, Force)
struct Lv_id *LV_ID;
long LP_Num;
int Force;
```

Description

Note: You must have root user authority to use this subroutine.

The **lvm_resynclp** subroutine initiates resynchronization for all the existing physical partition copies of the specified logical partition, if required.

The *LV_ID* parameter specifies the logical volume that contains the logical partition needing resynchronization. The *LP_Num* parameter is the logical partition number within the logical volume to be resynchronized. The volume group must be varied on, or an error is returned.

The *Force* parameter is used to specify whether all physical copies or only stale physical copies of a logical partition are to be resynchronized. When the *Force* parameter is False, a good physical copy is propagated only to the stale physical copies. This is sufficient for most logical volumes.

If the *Force* parameter is True, a good physical copy is chosen and propagated to all other copies of the logical partition whether or not they are stale. Setting the *Force* parameter to True is sometimes necessary in cases where mirror–write consistency recovery was not specified for the logical volume. This is especially important after a crash occurs while writing to the logical volume. It is recommended that mirror write consistency be selected for most mirrored logical volumes. For more information on mirror write consistency, see the **lvm_createlv** and **lvm_changelv** subroutines.

Parameters

LP_Num	Specifies the logical partition number within the logical volume to be resynchronized.
LV_ID	Specifies the logical volume that contains the logical partition needing resynchronization.
Force	Specifies whether all physical copies or only stale physical copies of a logical partition are to be resynchronized.

Return Values

Upon successful completion, the **lvm_resynclp** subroutine returns a value of 0.

Error Codes

If the **lvm_resynclp** subroutine fails, it returns one of the following error codes:

LVM_ALLOCERR	A memory allocation error occurred.
LVM_DALVOPN	The logical volume reserved by the volume group could not be opened.
LVM_FORCEOFF	The volume group has been forcefully varied off due to a loss of quorum.
LVM_INVALID_PARAM	One of the fields passed in did not have a valid value.
LVM_INV_DEVENT	A device has a major number that does not correspond to the volume group being worked in.
LVM_INVALID_MIN_NUM	An invalid minor number was received.
LVM_MAPFBSY	The volume group is currently locked because system management on the volume group is being done by another process.
LVM_MAPFOPN	The mapped file, which contains a copy of the volume group descriptor area used for making changes to the volume group, could not be opened.
LVM_MAPFRDWR	An error occurred while trying to read or write the mapped file.
LVM_NOTCHARDEV	A device is not a raw or character device.
LVM_NOTSYNCED	The logical partition was not completely resynced.
LVM_OFFLINE	The volume group is offline and should be online.

Implementation Specifics

This subroutine is part of Base Operating System (BOS) Runtime.

Related Information

The **lvm_changelv** subroutine, **lvm_createlv** subroutine, **lvm_extendlv** subroutine, **lvm_resynclv** subroutine, **lvm_resyncpv** subroutine, **lvm_varyonvg** subroutine.

List of Logical Volume Subroutines and Logical Volume Programming Overview in *AIX General Programming Concepts : Writing and Debugging Programs*.

lvm_resynclv Subroutine

Purpose

Synchronizes all physical copies of all of the logical partitions for a logical volume.

Library

Logical Volume Manager Library (**liblvm.a**)

Syntax

```
#include <lvm.h>

int lvm_resynclv (LV_ID, Force)
struct Lv_id *LV_ID;
int Force;
```

Description

Note: You must have root user authority to use this subroutine.

The **lvm_resynclv** subroutine synchronizes all physical copies of a logical partition for each logical partition of the logical volume specified by the *LV_ID* parameter. The volume group must be varied on or an error is returned.

The *Force* parameter is used to specify whether all physical copies or only stale physical copies of a logical partition are to be resynchronized. When the *Force* parameter is False, a good physical copy is propagated only to the stale physical copies. This is sufficient for most logical volumes.

If the *Force* parameter is True, a good physical copy is chosen and propagated to all other copies of the logical partition whether or not they are stale. Setting the *Force* parameter to True is sometimes necessary in cases in which mirror–write consistency recovery was not specified for the logical volume. This is especially important after a crash occurs while writing to the logical volume. It is recommended that mirror write consistency be selected for most mirrored logical volumes. For more information on mirror write consistency, see the **lvm_createlv** and **lvm_changelv** subroutines.

Parameters

LV_ID	Specifies the logical volume name.
Force	Specifies which physical copies of a logical partition will be resynchronized.

Return Values

Upon successful completion, the **lvm_resynclv** subroutine returns a value of 0.

Error Codes

If the **lvm_resynclv** subroutine fails, it returns one of the following error codes:

LVM_ALLOCERR	A memory allocation error occurred.
LVM_DALVOPN	The logical volume reserved by the volume group could not be opened.
LVM_FORCEOFF	The volume group has been forcefully varied off due to a loss of quorum.
LVM_INVALID_MIN_NUM	An invalid minor number was received.
LVM_INVALID_PARAM	One of the fields passed in did not have a valid value.

LVM_INV_DEVENT	A device has a major number that does not correspond to the volume group being worked in.
LVM_MAPFBSY	The volume group is currently locked because system management on the volume group is being done by another process.
LVM_MAPFOPN	The mapped file, which contains a copy of the volume group descriptor area used for making changes to the volume group, could not be opened.
LVM_MAPFRDWR	An error occurred while trying to read or write the mapped file.
LVM_NOTCHARDEV	A device is not a raw or character device.
LVM_NOTSYNCED	The logical volume could not be completely resynced.
LVM_OFFLINE	The volume group is offline and should be online.
LVM_WRTDAERR	An error occurred while trying to initialize either the Volume Group Descriptor Area, the Volume Group Status Area, or the Mirror–Write Consistency Cache Area on the physical volume.

Implementation Specifics

This subroutine is part of Base Operating System (BOS) Runtime.

Related Information

The **lvm_changelv** subroutine, **lvm_createlv** subroutine, **lvm_resynclp** subroutine, **lvm_resyncpv** subroutine, **lvm_varyonvg** subroutine.

List of Logical Volume Subroutines and Logical Volume Programming Overview in *AIX General Programming Concepts : Writing and Debugging Programs*.

lvm_resyncpv Subroutine

Purpose

Synchronizes all physical partitions on a physical volume with the related copies of the logical partition to which they correspond.

Library

Logical Volume Manager Library (**liblvm.a**)

Syntax

```
#include <lvm.h>

int lvm_resyncpv (VG_ID, PV_ID, Force)
struct unique_id *VG_ID;
struct unique_id *PV_ID;
int Force;
```

Description

The **lvm_resyncpv** subroutine synchronizes all copies of the corresponding logical partition for each physical partition on the physical volume specified by the *PV_ID* parameter. The *VG_ID* parameter specifies the volume group that contains the physical volume to be resynced. The volume group must be varied on, or the **LVM_OFFLINE** error code is returned.

The *Force* parameter is used to specify whether all physical copies or only stale physical copies of a logical partition are to be resynchronized. When the *Force* parameter is False, a good physical copy is propagated only to the stale physical copies. This is sufficient for most logical volumes.

If the *Force* parameter is True, a good physical copy is chosen and propagated to all other copies of the logical partition regardless of whether they are stale. Setting the *Force* parameter to True is sometimes necessary in cases where mirror write consistency recovery was not specified for the logical volume. This is especially important after a crash occurs while writing to the logical volume. It is recommended that mirror write consistency be selected for most mirrored logical volumes. For more information on mirror–write consistency, see the **lvm_createlv** and **lvm_changelv** subroutines.

Notes:

1. The resync of the physical volume is done by resyncing entire logical partitions to which any stale physical partitions belong on the physical volume. Because a complete logical partition is resynced, other physical volumes other than the one specified may be partially or completely resynced.
2. You must have root user authority to use this subroutine.

Parameters

VG_ID	Specifies the volume group that contains the physical volume to be resynced.
PV_ID	Specifies the physical volume.
Force	Specifies the physical copies of a logical partition to be synchronized.

Return Values

The **lvm_resyncpv** subroutine returns a value of 0 upon successful completion.

Error Codes

If the **lvm_resyncpv** subroutine fails, it returns one of the following error codes:

LVM_ALLOCERR	A memory allocation error occurred.
LVM_DALVOPN	The logical volume reserved by the volume group could not be opened.
LVM_FORCEOFF	The volume group has been forcefully varied off due to a loss of quorum.
LVM_INVALID_PARAM	One of the fields passed in did not have a valid value.
LVM_INV_DEVENT	A device has a major number that does not correspond to the volume group being worked in.
LVM_MAPFBSY	The volume group is currently locked because system management on the volume group is being done by another process.
LVM_MAPFOPN	The mapped file, which contains a copy of the volume group descriptor area used for making changes to the volume group, could not be opened.
LVM_MAPFRDWR	An error occurred while trying to read or write the mapped file.
LVM_NOTCHARDEV	A device is not a raw or character device.
LVM_NOTSYNCED	The physical volume could not be completely resynced.
LVM_OFFLINE	The volume group is offline and should be online.
LVM_WRTDAERR	An error occurred while trying to initialize either the volume group descriptor area, the volume group status area, or the mirror write consistency cache area on the physical volume.

Implementation Specifics

This subroutine is part of Base Operating System (BOS) Runtime.

Related Information

The **lvm_changelv** subroutine, **lvm_createlv** subroutine, **lvm_resynclv** subroutine, **lvm_resynclp** subroutine, **lvm_varyonvg** subroutine.

List of Logical Volume Subroutines and Logical Volume Programming Overview in *AIX General Programming Concepts : Writing and Debugging Programs*.

lvm_varyoffvg Subroutine

Purpose

Varies off a volume group.

Library

Logical Volume Manager Library (**liblvm.a**)

Syntax

```
#include <lvm.h>

int lvm_varyoffvg (VaryOffVG)
struct varyoffvg *VaryOffVG;
```

Description

Note: You must have root user authority to use this subroutine.

The **lvm_varyoffvg** subroutine varies off a specified volume group. All logical volumes in the volume group to be varied off must be closed.

The **varyoffvg** structure pointed to by the *VaryOffVG* parameter is found in the **lvm.h** file and defined as follows:

```
struct varyoffvg
        {
         struct unique_id vg_id;
         long lvs_only;
        }  * Varyoffvg;
```

Field	Description
`lvs_only`	Indicates whether the volume group is to be varied off entirely or whether system management commands, which act on the volume group, are still permitted. If the `lvs_only` field is True, then all logical volumes in the volume group will be varied off, but the volume group is still available for system management commands that act on the volume group. If the `lvs_only` field is False, then the entire volume group is varied off, and system management commands cannot be performed on the volume group. The normal value for this flag is False.
`vg_id`	Specifies the volume group to be varied off.

Parameters

VaryOfFVG	Points to the **varyoffvg** structure.

Return Values

Upon successful completion, the **lvm_varyoffvg** subroutine returns a value of 0.

Error Codes

If the **lvm_varyoffvg** subroutine fails, it returns one of the following error codes:

LVM_ALLOCERR	A memory allocation error occurred.
LVM_INVALID_PARAM	An invalid parameter was passed into the routine.
LVM_INV_DEVENT	A device entry is invalid and cannot be checked to determine if it is raw.

LVM_LVOPEN	An open logical volume was encountered when it should be closed.
LVM_MAPFOPN	The mapped file, which contains a copy of the volume group descriptor area used for making changes to the volume group, could not be opened.
LVM_MAPFRDWR	An error occurred while trying to write to the mapped file.
LVM_NOTCHARDEV	The device specified is not a raw or character device.
LVM_OFFLINE	The volume group specified is offline. It must be varied on to perform this operation.

Implementation Specifics

This subroutine is part of Base Operating System (BOS) Runtime.

Related Information

The **lvm_varyonvg** subroutine.

List of Logical Volume Subroutines and Logical Volume Programming Overview in *AIX General Programming Concepts : Writing and Debugging Programs*.

lvm_varyonvg Subroutine

Purpose

Varies a volume group on–line.

Library

Logical Volume Manager Library (**liblvm.a**)

Syntax

```
#include <lvm.h>

int lvm_varyonvg (VaryOnVG)
struct varyonvg *VaryOnVG;
```

Description

Note: You must have root user authority to use this subroutine.

The **lvm_varyonvg** subroutine varies on the specified volume group. The **lvm_varyonvg** subroutine contacts the physical volumes in the volume group and recovers the volume group descriptor area if necessary.

The **varyonvg** structure pointed to by the *VaryOnVG* parameter is found in the **lvm.h** file and is defined as follows:

```
struct varyonvg
        {
          mid_t kmid;
          char *vgname;
          long vg_major;
          struct unique_id vg_id;
          long noopen_lvs;
          long reserved;
          long auto_resync;
          long misspv_von;
          long missname_von;
          short int override;
          struct {
                   long num_pvs;
                   struct {
                           struct unique_id pv_id;
                            char *pvname;
                          } pv [LVM_MAXPVS];
                 } vvg_in;
          struct {
                   long num_pvs;
                   struct {
                          struct unique_id pv_id;
                           char *pvname;
                          long pv_status;
                        } pv [2 * LVM_MAXPVS];
                 } vvg_out;
          };
```

Field	Description
`kmid`	Specifies the module ID that identifies the entry point of the logical volume device driver module.
`vgname`	Specifies the character special file name, which is either the full path name or a file name that resides in the /**dev** directory (for example **rvg13**) of the volume group device. This device is actually a logical volume with a minor number reserved for use by the Logical Volume Manager (LVM).
`vg_major`	Specifies the major number of the volume group to be varied on.
`noopen_lvs`	Contains either a True or False value. If this field is False, the **lvm_varyonvg** subroutine builds and sends data structures describing all logical volumes in the volume group to the logical volume device driver. This enables those logical volumes to be opened and accessed. If the `noopen_lvs` flag is True, then queries to the volume group and other system management functions can be performed, but opens to the logical volumes in the volume group are not allowed. Resynchronization and migrate commands cannot be used because they require the presence of the logical volumes.
`auto_resync`	Contains either a True or False value. If this field is False**,** then resynchronization of physical and logical volumes containing stale partitions will not be performed and should be initiated by the caller at some other time. The LVM subroutines **lvm_resyncpv** and **lvm_resynclv** are provided to perform resynchronization of physical and logical volumes, respectively. The recommended value for the `auto_resync` field is True.
`pvname`	Contains the character special file name, which is either the full path name or a single file name that resides in the /**dev** directory (for example, **rhdisk0**) of the physical volume being installed in the new volume group.

The **vvg_in** structure contains input from the caller to the **lvm_varyonvg** subroutine which describes the physical volumes in the volume group. The `num_pvs` field is the number of entries in the **pv** array of structures. Each entry in the **pv** array contains the ID (`pv_id`) and name (`pvname`) of a physical volume in the volume group. Unless the volume group is already varied on, this array should contain an entry for each physical volume in the volume group.

The **vvg_out** structure contains output from the **lvm_varyonvg** subroutine to the user. This subroutine describes the status of the physical volumes in the caller's input list and any additional physical volumes in the volume group, but not included in the input list. The `num_pvs` field is the number of entries in the **pv** array of structures. Each entry in the **pv** array contains the ID (`pv_id`), the name (`pvname`), and the status (`pv_status`) of a physical volume contained in the input list or the volume group.

`The pv_status` field contains one of the following values for each physical volume in the **vvg_out** structure if either the volume group is varied on successfully or an **LVM_MISSPVNAME** or **LVM_MISSINGPV** error is returned:

LVM_PVACTIVE	This physical volume is currently an active member of the volume group.
LVM_PVMISSING	This physical volume is currently unavailable and missing from the volume group.
LVM_PVREMOVED	This physical volume has been temporarily removed from the volume group by user request or by virtue of its being missing at the time of a forced vary–on.

LVM_INVPVID	This physical volume is not a member of the specified volume group.
LVM_NONAME	This physical volume is a member of the volume group, but its name was not passed in the input list.
LVM_DUPPVID	A physical volume with the same pv_id field value as this physical volume has already appeared earlier in the input list.
LVM_LVMRECNMTCH	This physical volume needs to be deleted from the volume group because it has invalid or non matching data in its LVM record. This may mean that the physical volume has been installed into another volume group.
LVM_NAMIDNMTCH	The pv_id for this physical volume was passed in the input list, but it does not match the pv_id of the specified physical volume device name.

For physical volumes in the input list that are found to be members of the specified volume group, the pv_status field contains the physical volume state of either **LVM_PVACTIVE**, **LVM_PVMISSING**, or **LVM_PVREMOVED**. If a physical volume with the same pv_id has appeared previously in the input list, the pv_status field contains **LVM_DUPPVID**. For physical volumes in the list which are not members of the volume group, the pv_status field will be **LVM_INVPVID**.

In some cases, a physical volume that is a member of the volume group might have a pv_status field value of **LVM_LVMRECNMTCH**. This means that the LVM record on the physical volume has either invalid or nonmatching data and that the physical volume cannot be brought on line. If this happens, it is most likely because the physical volume has been installed into another volume group without first deleting it from this one. The user should now delete this physical volume from this volume group, since it can no longer be accessed as a member of this volume group.

For physical volumes that are members of the volume group but were not in the input list, the pv_status field value will be **LVM_NONAME** or **LVM_NAMIDNMTCH**. In this case the pv_id field contains the ID of the physical volume, and the pvname field contains a null pointer. An error code of **LVM_MISSPVNAME** is returned to the caller unless the subroutine was called with a value of TRUE for the missname_von field.

The pv_status field for each physical volume in the **vvg_out** structure contains one of the following values if either the **LVM_NOQUORUM** or **LVM_NOVGDAS** error is returned.

LVM_PVNOTFND	Either the physical volume device could not be opened or necessary information in the IPL or LVM record could not be read.
LVM_PVNOTINVG	The LVM record for this physical volume indicates that it is not a member of the specified volume group.
LVM_PVINVG	The LVM record for this physical volume indicates that it is a member of the specified volume group.

It is recommended that the missname_von field contain a value of FALSE for the first call to the **lvm_varyonvg** subroutine since a value of TRUE means that any physical volume for which a name was not passed in the input list is given a state of **LVM_PVMISSING**. Users of the volume group cannot have access to that physical volume until a subsequent call is made to the **lvm_varyonvg** subroutine for that volume group.

If the misspv_von field is TRUE, the volume group is varied on (provided a quorum exists) even if some of the physical volumes in the volume group have a state of **LVM_PVMISSING**. If the flag is FALSE, the volume group is varied on only if all physical volumes in the volume group that do not have a state of **LVM_PVREMOVED** are in the active state (**LVM_PVACTIVE**). The value recommended for this flag is TRUE. For any physical volume with a state of **LVM_PVMISSING** or **LVM_PVREMOVED** when the volume group is varied on, access to that physical volume is not available through LVM. If the state of a physical volume is changed from **LVM_PVREMOVED** to **LVM_PVACTIVE** through a call to the **lvm_changepv** subroutine, then that physical volume is available to LVM, provided that it is not missing at the time.

If the `override` field is TRUE, an attempt is made to vary on the volume group even if access to a quorum (or majority) of volume group descriptor area copies or a quorum of the volume group status area copies cannot be obtained. Provided that there is at least one valid copy of the descriptor area and at least one valid copy of the status area, the vary–on of the volume group will proceed with the latest available copies of the volume group descriptor area and status area.

If the volume group is forcefully varied on by overriding the absence of a quorum, the **PV** state of all missing physical volumes is changed to **LVM_PVREMOVED**. When a physical volume's state is changed to **LVM_PVREMOVED**, any copies of the volume group descriptor area and status area that it contains are removed. The physical volume no longer takes part in quorum checking until it is returned to the volume group. Also, the physical volume cannot become an active member of the volume group until it is returned. See the **lvm_changepv** subroutine for more information about removing and returning physical volumes.

The recommended value for the `override` field is FALSE. If the user chooses to override the **LVM_NOQUORUM** error and artificially force a quorum, LVM does not guarantee the data integrity of the data contained in the chosen copies of the volume group descriptor area and status area. For more information about quorums and quorum checking, see the "Logical Volume Storage Overview" in *AIX 4.3 System Management Guide: Operating System and Devices*.

If a physical volume's state is **LVM_PVMISSING** when the volume group is varied on, then access to that physical volume can be made available to the LVM only by again calling the **lvm_varyonvg** subroutine for that volume group. When the **lvm_varyonvg** subroutine is called for a volume group that is already varied on, a check is made for any physical volumes in the volume group with a state of **LVM_PVMISSING**, and an attempt will be made to open those physical volumes. Any previously missing physical volumes that are successfully opened are defined to the logical volume device driver, and access to those physical volumes will again be available through the LVM.

When the **lvm_varyonvg** subroutine is called for an already varied–on volume group for the purpose of changing previously missing physical volumes back to the active state, the caller does not need to pass an entire list of physical volumes in the **vvg_in** structure. The caller needs to pass only information for missing physical volumes that the caller is attempting to return to the **LVM_PVACTIVE** state.

Parameters

VaryOnVG	Points to the **varyonvg** structure.

Return Values

Upon successful completion, the subroutine returns one or more of the following return codes:

LVM_SUCCESS	The volume group was successfully varied on.
LVM_CHKVVGOUT	The volume group was varied on successfully, but there is information in the **vvg_out** structure that should be checked.

Error Codes

If the **lvm_varyonvg** subroutine does not complete successfully, it returns one of the following error codes:

LVM_ALLOCERR	A memory allocation error has occurred.
LVM_INVALID_PARAM	A field in the **varyongvg** structure is invalid or the pointer structure is invalid.
LVM_INVCONFIG	An error occurred while attempting to configure this volume group into the kernel.
LVM_INV_DEVENT	The device entry for a specified device is not valid and cannot be checked to determine if it is raw.
LVM_MAPFOPN	The mapped file, which contains a copy of the volume group descriptor area used for making changes to the volume group, could not be opened.
LVM_MAPFRDWR	An error occurred while trying to read or write the mapped file.
LVM_MISSINGPV	The volume group was not varied on because one of the physical volumes in the volume group has a state of **LVM_PVMISSING**. This error will be returned only if the `misspv_von` field has a value of FALSE; otherwise, the volume group is varied on if a quorum is obtained.
LVM_MISSPVNAME	The volume group was not varied on because the volume group contains a physical volume ID for which no name was passed. The **vvg_out** structure contains the `pv_id` field, a null pointer for the `pvname` field**,** and a `pv_status` field value of **LVM_NONAME** for any physical volume in the volume group for which a name was not passed in the **vvg_in** structure. This error is returned only if the `missname_von` field has a value of FALSE. Otherwise, the volume group is varied on if a quorum is obtained.
LVM_NOQUORUM	The volume group could not be varied on because access to a quorum, or majority, of all volume group descriptor areas or access to a quorum of all volume group status areas could not be obtained.
LVM_NOTCHARDEV	The device specified is not a raw or character device.
LVM_NOVGDAS	The volume group could not be varied on because access to a valid copy of the volume group descriptor area could not be obtained or access to a valid copy of the volume group status area could not be obtained.

Implementation Specifics

This subroutine is part of Base Operating System (BOS) Runtime.

Related Information

The **lvm_changepv** subroutine, **lvm_varyoffvg** subroutine.

Logical Volume Storage Overview in *AIX 4.3 System Management Guide: Operating System and Devices*.

List of Logical Volume Subroutines and Logical Volume Programming Overview in *AIX General Programming Concepts : Writing and Debugging Programs*.

madd, msub, mult, mdiv, pow, gcd, invert, rpow, msqrt, mcmp, move, min, omin, fmin, m_in, mout, omout, fmout, m_out, sdiv, or itom Subroutine

Purpose

Multiple–precision integer arithmetic.

Library

Berkeley Compatibility Library (**libbsd.a**)

Syntax

```
#include <mp.h>
#include <stdio.h>

typedef struct mint {int Length; short *Value} MINT;

madd(a,b,c)
msub(a,b,c)
mult(a,b,c)
mdiv(a,b,q,r)
pow(a,b,m,c)
gcd(a,b,c)
invert(a,b,c)
rpow(a,n,c)
msqrt(a,b,r)
mcmp(a,b)
move(a,b)
min(a)
omin(a)
fmin(a,f)
m_in(a,n,f)
mout(a)
omout(a)
fmout(a,f)
m_out(a,n,f)
MINT *a, *b, *c, *m, *q, *r;
FILE *f;
int n;

sdiv(a,n,q,r)
MINT *a, *q;
short n;
short *r;

MINT *itom(n)
```

Description

These subroutines perform arithmetic on integers of arbitrary *Length*. The integers are stored using the defined type **MINT**. Pointers to a **MINT** can be initialized using the **itom** subroutine, which sets the initial *Value* to *n*. After that, space is managed automatically by the subroutines.

The **madd** subroutine, **msub** subroutine, and **mult** subroutine assign to *c* the sum, difference, and product, respectively, of *a* and *b*.

The **mdiv** subroutine assigns to *q* and *r* the quotient and remainder obtained from dividing *a* by *b*.

The **sdiv** subroutine is like the **mdiv** subroutine except that the divisor is a short integer *n* and the remainder is placed in a short whose address is given as *r*.

The **msqrt** subroutine produces the integer square root of *a* in *b* and places the remainder in *r*.

The **rpow** subroutine calculates in c the value of *a* raised to the (regular integral) power *n*, while the **pow** subroutine calculates this with a full multiple precision exponent *b* and the result is reduced modulo *m*.

Note: The **pow** subroutine is also present in the IEEE Math Library, **libm.a**, and the System V Math Library, **libmsaa.a**. The **pow** subroutine in **libm.a** or **libmsaa.a** may be loaded in error unless the **libbsd.a** library is listed before the **libm.a** or **libmsaa.a** library on the command line.

The **gcd** subroutine returns the greatest common denominator of *a* and *b* in *c*, and the **invert** subroutine computes *c* such that *a***c* mod *b*=1, for *a* and *b* relatively prime.

The **mcmp** subroutine returns a negative, 0, or positive integer value when *a* is less than, equal to, or greater than *b*, respectively.

The **move** subroutine copies *a* to *b*. The **min** subroutine and **mout** subroutine do decimal input and output while the **omin** subroutine and **omout** subroutine do octal input and output. More generally, the **fmin** subroutine and **fmout** subroutine do decimal input and output using file *f*, and the **m_in** subroutine and **m_out** subroutine do inputs and outputs with arbitrary radix *n*. On input, records should have the form of strings of digits terminated by a new line; output records have a similar form.

Parameters

Length	Specifies the length of an integer.
Value	Specifies the initial value to be used in the routine.
a	Specifies the first operand of the multiple–precision routines.
b	Specifies the second operand of the multiple–precision routines.
c	Contains the integer result.
f	A pointer of the type **FILE** that points to input and output files used with input/output routines.
m	Indicates modulo.
n	Provides a value used to specify radix with **m_in** and **m_out**, power with **rpow**, and divisor with **sdiv**.
q	Contains the quotient obtained from **mdiv**.
r	Contains the remainder obtained from **mdiv, sdiv,** and **msqrt**.

Error Codes

Error messages and core images are displayed as a result of illegal operations and running out of memory.

Implementation Specifics

These subroutines are part of Base Operating System (BOS) Runtime.

Programs that use the multiple–precision arithmetic functions must link with the **libbsd.a** library.

Bases for input and output should be less than or equal to 10.

pow is also the name of a standard math library routine.

Files

/usr/lib/libbsd.a	Object code library.

Related Information

The **bc** command, **dc** command.

Subroutines Overview in *AIX General Programming Concepts : Writing and Debugging Programs*.

madvise Subroutine

Purpose

Advises the system of expected paging behavior.

Library

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

Syntax

```
#include <sys/types.h>
#include <sys/mman.h>

int madvise(addr, len, behav)
caddr_t addr;
size_t len;
int behav;
```

Description

The **madvise** subroutine permits a process to advise the system about its expected future behavior in referencing a mapped file region or anonymous memory region.

Parameters

addr Specifies the starting address of the memory region. Must be a multiple of the page size returned by the **sysconf** subroutine using the **_SC_PAGE_SIZE** value for the *Name* parameter.

len Specifies the length, in bytes, of the memory region. If the *len* value is not a multiple of page size as returned by the **sysconf** subroutine using the **_SC_PAGE_SIZE** value for the *Name* parameter, the length of the region will be rounded up to the next multiple of the page size.

behav Specifies the future behavior of the memory region. The following values for the *behav* parameter are defined in the **/usr/include/sys/mman.h** file:

Value	Paging Behavior Message
MADV_NORMAL	The system provides no further special treatment for the memory region.
MADV_RANDOM	The system expects random page references to that memory region.
MADV_SEQUENTIAL	The system expects sequential page references to that memory region.
MADV_WILLNEED	The system expects the process will need these pages.
MADV_DONTNEED	The system expects the process does not need these pages.
MADV_SPACEAVAIL	The system will ensure that memory resources are reserved.

Return Values

When successful, the **madvise** subroutine returns 0. Otherwise, it returns –1 and sets the **errno** global variable to indicate the error.

Error Codes

If the **madvise** subroutine is unsuccessful, the **errno** global variable can be set to one of the following values:

EINVAL	The *behav* parameter is invalid.
ENOSPC	The *behav* parameter specifies **MADV_SPACEAVAIL** and resources cannot be reserved.

Implementation Specifics

The **madvise** subroutine has no functionality and is supported for compatibility only. It is part of Base Operating System (BOS) Runtime.

Related Information

The **mmap** subroutine, **sysconf** subroutine.

List of Memory Manipulation Services and Understanding Paging Space Programming Requirements in *AIX General Programming Concepts : Writing and Debugging Programs*.

makecontext or swapcontext Subroutine

Purpose

Modifies the context specified by *ucp*.

Library

(**libc.a**)

Syntax

#include <ucontext.h>

void makecontext (ucontext_t **ucp*, **(void** **func*) **()**, **int** *argc*, **...);**
int swapcontext (uncontext_t **oucp*, **const uncontext_t** **ucp*);

Description

The **makecontext** subroutine modifies the context specified by *ucp*, which has been initialized using **getcontext** subroutine. When this context is resumed using **swapcontext** subroutine or **setcontext** subroutine, program execution continues by calling *func* parameter, passing it the arguments that follow *argc* in the **makecontext** subroutine.

Before a call is made to **makecontext** subroutine, the context being modified should have a stack allocated for it. The value of *argc* must match the number of integer argument passed to *func* parameter, otherwise the behavior is undefined.

The **uc_link** member is used to determine the context that will be resumed when the context being modified by **makecontext** subroutine returns. The **uc_link** member should be initialized prior to the call to **makecontext** subroutine.

The **swapcontext** subroutine function saves the current context in the context structure pointed to by *oucp* parameter and sets the context to the context structure pointed to by *ucp*.

Parameters

ucp	A pointer to a user structure.
oucp	A pointer to a user structure.
func	A pointer to a function to be called when *ucp* is restored.
argc	The number of arguments being passed to *func* parameter.

Return Values

On successful completion, **swapcontext** subroutine returns 0. Otherwise, a value of –1 is returned and **errno** is set to indicate the error.

–1	Not successful and the **errno** global variable is set to one of the following error codes.

Error Codes

ENOMEM	The *ucp* argument does not have enough stack left to complete the operation.

Related Information

The **exec** subroutine, **exit** subroutine, **wait** subroutine, **getcontext** subroutine, **sigaction** subroutine, and **sigprocmask** subroutine.

malloc, free, realloc, calloc, mallopt, mallinfo, alloca, or valloc Subroutine

Purpose

Provides a memory allocator.

Libraries

Berkeley Compatibility Library (**libbsd.a**)

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

Syntax

```
#include <stdlib.h>

void *malloc (Size)
size_t Size;

void free (Pointer)
void *Pointer;

void *realloc (Pointer, Size)
void *Pointer;
size_t Size;

void *calloc (NumberOfElements, ElementSize )
size_t NumberOfElements;
size_t ElementSize;

char *alloca (Size)
int Size;

void *valloc (Size)
size_t Size;

#include <malloc.h>
#include <stdlib.h>

int mallopt (Command, Value)
int Command;
int Value;

struct mallinfo mallinfo( )
```

Description

The **malloc** and **free** subroutines provide a general–purpose memory allocation package.

The **malloc** subroutine returns a pointer to a block of memory of at least the number of bytes specified by the *Size* parameter. The block is aligned so that it can be used for any type of data. Undefined results occur if the space assigned by the **malloc** subroutine is overrun.

The **free** subroutine frees a block of memory previously allocated by the **malloc** subroutine. Undefined results occur if the *Pointer* parameter is not a valid pointer. If the *Pointer* parameter is a null value, no action will occur.

The **realloc** subroutine changes the size of the block of memory pointed to by the *Pointer* parameter to the number of bytes specified by the *Size* parameter and returns a new pointer to the block. The pointer specified by the *Pointer* parameter must have been created with the **malloc**, **calloc**, or **realloc** subroutines and not been deallocated with the **free** or **realloc** subroutines. Undefined results occur if the *Pointer* parameter is not a valid pointer

The contents of the block returned by the **realloc** subroutine remain unchanged up to the lesser of the old and new sizes. If a large enough block of memory is not available, the **realloc** subroutine acquires a new area and moves the data to the new space. The **realloc** subroutine supports the old **realloc** protocol wherein the **realloc** protocol returns a pointer to a previously freed block of memory if that block satisfies the **realloc** request. The **realloc** subroutine searches a list, maintained by the **free** subroutine, of the ten most recently freed blocks of memory. If the list does not contain a memory block that satisfies the specified *Size* parameter, the **realloc** subroutine calls the **malloc** subroutine. This list is cleared by calls to the **malloc**, **calloc**, **valloc**, or **realloc** subroutines.

The **calloc** subroutine allocates space for an array with the number of elements specified by the *NumberOfElements* parameter. The *ElementSize* parameter specifies in bytes each element, and initializes space to zeros. The order and contiguity of storage allocated by successive calls to the **calloc** subroutine is unspecified. The pointer returned points to the first (lowest) byte address of the allocated space.

The **alloca** subroutine allocates the number of bytes of space specified by the *Size* parameter in the stack frame of the caller. This space is automatically freed when the subroutine that called the **alloca** subroutine returns to its caller.

The **valloc** subroutine has the same effect as **malloc**, except that the allocated memory is aligned to a multiple of the value returned by **sysconf**(_ *SC_PAGESIZE*).

The **mallopt** and **mallinfo** subroutines are provided for source–level compatibility with the System V **malloc** subroutine. Nothing done with the **mallopt** subroutine affects how memory is allocated by the system.

The **mallinfo** subroutine can be used to obtain information about the heap managed by the **malloc** subroutine. Refer to the **malloc.h** file for details of the **mallinfo** structure.

> **Note:** AIX Version 3 uses a delayed paging slot allocation technique for storage allocated to applications. When storage is allocated to an application with a subroutine such as **malloc**, no paging space is assigned to that storage until the storage is referenced. This technique is useful for applications that allocate large sparse memory segments. However, this technique may affect portability of applications that allocate very large amounts of memory. If the application expects that calls to **malloc** will fail when there is not enough backing storage to support the memory request, the application may allocate too much memory. When this memory is referenced later, the machine quickly runs out of paging space and the operating system kills processes so that the system is not completely exhausted of virtual memory.The application that allocates memory must ensure that backing storage exists for the storage being allocated.To deal with this style of allocation, sample code is provided in **/usr/lpp/bos/samples**.

Parameters

Size	Specifies a number of bytes of memory.
Pointer	Points to the block of memory that was returned by the **malloc** or **calloc** subroutines. The *Pointer* parameter points to the first (lowest) byte address of the block.

Command	Specifies a **mallopt** subroutine command. If **M_DISCLAIM** is used, then the paging space and physical memory in use by freed **malloc** space is returned to the system resource pool. If they are needed to fullfill a **malloc** request, they will be allocated to the process as needed. The address space is not altered. This will only release whole pages at a time.
Value	Specifies the value to which the **M_MXFAST**, **M_NLBLKS**, **M_GRAIN**, or **M_KEEP** label is to be set. These constants are provided only for source code compatibility. They do not affect the operation of subsequent calls to the **malloc** subroutine.
NumberOfElements	Specifies the number of elements in the array.
ElementSize	Specifies the size of each element in the array.

Return Values

Each of the allocation subroutines returns a pointer to space suitably aligned for storage of any type of object. Cast the pointer to the pointer–to–element type before using it.

The **malloc**, **realloc**, **calloc** , and **valloc** subroutines return a null pointer if there is no available memory, or if the memory arena has been corrupted by being stored outside the bounds of a block. When this happens, the block pointed to by the *Pointer* parameter may be destroyed.

If the **malloc** or **valloc** subroutine is called with a size of 0, the subroutine returns a null pointer. If the **realloc** subroutine is called with a nonnull pointer and a size of 0, the **realloc** subroutine attempts to free the pointer and return a null pointer. If the **realloc** subroutine is called with a null pointer, it calls the **malloc** subroutine for the specified size and returns the null pointer.

Error Codes

When the memory allocation subroutines are unsuccessful, the global variable **errno** may be set to the following values:

EINVAL	Indicates a call has requested 0 bytes.
ENOMEM	Indicates that not enough storage space was available.

Implementation Specifics

These subroutines are part of Base Operating System (BOS) Runtime.

The **valloc** subroutine, found in many BSD systems, is supported as a compatibility interface in the Berkeley Compatibility Library (**libbsd.a**). The **valloc** subroutine calls the **malloc** subroutine and automatically page–aligns requests that are greater than one page.

The **valloc** subroutine, found in many BSD systems, is supported as a compatibility interface in the Berkeley Compatibility Library (**libbsd.a**). The **valloc** subroutine calls the **malloc** subroutine and automatically page–aligns requests that are greater than one page. The only difference between the **valloc** subroutine in the **libbsd.a** library and the one in the standard C library (described above) is in the value returned when the size parameter is zero.

The following is the syntax for the **valloc** subroutine:

```
char *valloc (Size)
unsigned int Size;
```

Related Information

The **_end**, **_etext**, or **_edata** identifier.

The **#pragma** compiler instruction.

Subroutines Overview and Understanding System Memory Allocation in *AIX General Programming Concepts : Writing and Debugging Programs*.

MatchAllAuths, , MatchAnyAuths, or MatchAnyAuthsList Subroutine

Purpose

Compare authorizations.

Library

Security Library (**libc.a**)

Syntax

```
#include <usersec.h>

int MatchAllAuths(CommaListOfAuths)
char *CommaListOfAuths;

int MatchAllAuthsList(CommaListOfAuths, NSListOfAuths)
char *CommaListOfAuths;
char *NSListOfAuths;

int MatchAnyAuths(CommaListOfAuths)
char *CommaListOfAuths;

int MatchAnyAuthsList(CommaListOfAuths, NSListOfAuths)
char *CommaListOfAuths;
char *NSListOfAuths;
```

Description

The **MatchAllAuthsList** subroutine compares the *CommaListOfAuths* against the *NSListOfAuths*. It returns a non–zero value if all the authorizations in *CommaListOfAuths* are contained in *NSListOfAuths*. The **MatchAllAuths** subroutine calls the **MatchAllAuthsList** subroutine passing in the results of the **GetUserAuths** subroutine in place of *NSListOfAuths*. If *NSListOfAuths* contains the OFF keyword, **MatchAllAuthsList** will return a zero value. If *NSListOfAuths* contains the ALL keyword and not the OFF keyword, **MatchAllAuthsList** will return a non–zero value.

The **MatchAnyAuthsList** subroutine compares the *CommaListOfAuths* against the *NSListOfAuths*. It returns a non–zero value if one or more of the authorizations in *CommaListOfAuths* are contained in *NSListOfAuths*. The **MatchAnyAuths** subroutine calls the **MatchAnyAuthsList** subroutine passing in the results of the **GetUserAuths** subroutine in place of *NSListOfAuths*. If *NSListOfAuths* contains the OFF keyword, **MatchAnyAuthsList** will return a zero value. If *NSListOfAuths* contains the ALL keyword and not the OFF keyword, **MatchAnyAuthsList** will return a non–zero value.

Parameters

CommaListOfAuths	Specifies one or more authorizations, each separated by a comma.
NSListOfAuths	Specifies zero or more authorizations. Each authorization is null terminated. The last entry in the list must be a null string.

Return Values

The subroutines return a non–zero value if a proper match was found. Otherwise, they will return zero. If an error occurs, the subroutines will return zero and set **errno** to indicate the error. If the subroutine returns zero and no error occurred, **errno** is set to zero.

matherr Subroutine

Purpose

Math error handling function.

Library

System V Math Library (**libmsaa.a**)

Syntax

```
#include <math.h>

int matherr (x)
struct exception *x;
```

Description

The **matherr** subroutine is called by math library routines when errors are detected.

You can use **matherr** or define your own procedure for handling errors by creating a function named `matherr` in your program. Such a user–designed function must follow the same syntax as **matherr.** When an error occurs, a pointer to the exception structure will be passed to the user–supplied `matherr` function. This structure, which is defined in the **math.h** file, includes:

```
int type;
char *name;
double arg1, arg2, retval;
```

Parameters

type Specifies an integer describing the type of error that has occurred from the following list defined by the **math.h** file:

DOMAIN	Argument domain error
SING	Argument singularity
OVERFLOW	Overflow range error
UNDERFLOW	Underflow range error
TLOSS	Total loss of significance
PLOSS	Partial loss of significance.

name Points to a string containing the name of the routine that caused the error.

arg1 Points to the first argument with which the routine was invoked.

arg2 Points to the second argument with which the routine was invoked.

retval Specifies the default value that is returned by the routine unless the user's **matherr** function sets it to a different value.

Return Values

If the user's **matherr** function returns a non–zero value, no error message is printed, and the **errno** global variable will not be set.

Error Codes

If the function **matherr** is not supplied by the user, the default error–handling procedures, described with the math library routines involved, will be invoked upon error. In every case, the **errno** global variable is set to **EDOM** or **ERANGE** and the program continues.

Implementation Specifics

This subroutine is part of Base Operating System (BOS) Runtime.

Related Information

The **bessel: j0**, **j1**, **jn**, **y0**, **y1**, **yn** subroutine, **exp**, **expm1**, **log**, **log10**, **log1p**, **pow** subroutine, **lgamma** subroutine, **hypot**, **cabs** subroutine, **sin**, **cos**, **tan**, **asin**, **acos**, **atan**,**atan2** subroutine, **sinh, cosh**, **tanh** subroutine.

Subroutines Overview in *AIX General Programming Concepts : Writing and Debugging Programs*.

mblen Subroutine

Purpose

Determines the length in bytes of a multibyte character.

Library

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

Syntax

```
#include <stdlib.h>

int mblen(MbString, Number)
const char *MbString;
size_t Number;
```

Description

The **mblen** subroutine determines the length, in bytes, of a multibyte character.

Parameters

Mbstring	Points to a multibyte character string.
Number	Specifies the maximum number of bytes to consider.

Return Values

The **mblen** subroutine returns 0 if the *MbString* parameter points to a null character. It returns –1 if a character cannot be formed from the number of bytes specified by the *Number* parameter. If *MbString* is a null pointer, 0 is returned.

Implementation Specifics

This subroutine is part of Base Operating System (BOS) Runtime.

Related Information

The **mbslen** subroutine, **mbstowcs** subroutine, **mbtowc** subroutine.

National Language Support Overview for Programming, Subroutines Overview, Understanding Multibyte Code and Wide Character Code Conversion Subroutines in *AIX General Programming Concepts : Writing and Debugging Programs*.

mbrlen Subroutine

Purpose

Get number of bytes in a character (restartable).

Library

Standard Library (**libc.a**)

Syntax

```
#include <wchar.h>

size_t mbrlen (const char *s, size_t n, mbstate_t *ps )
```

Description

If *s* is not a null pointer, **mbrlen** determines the number of bytes constituting the character pointed to by *s.* It is equivalent to:

```
mbstate_t internal;
mbrtowc(NULL, s, n, ps != NULL ? ps : &internal);
```

If *ps* is a null pointer, the **mbrlen** function uses its own internal **mbstate_t** object, which is initialized at program startup to the initial conversion state. Otherwise, the mbstate_t object pointed to by *ps* is used to completely describe the current conversion state of the associated character sequence. The implementation will behave as if no function defined in this specification calls **mbrlen**.

The behavior of this function is affected by the LC_CTYPE category of the current locale.

Return Values

The **mbrlen** function returns the first of the following that applies:

0	If the next **n** or fewer bytes complete the character that corresponds to the null wide–character
positive	If the next **n** or fewer bytes complete a valid character; the value returned is the number of bytes that complete the character.
(size_t)–2	If the next **n** bytes contribute to an incomplete but potentially valid character, and all n bytes have been processed. When n has at least the value of the MB_CUR_MAX macro, this case can only occur if s points at a sequence of redundant shift sequences (for implementations with state–dependent encodings).
(size_t)–1	If an encoding error occurs, in which case the next n or fewer bytes do not contribute to a complete and valid character. In this case, EILSEQ is stored in **errno** and the conversion state is undefined.

Error Codes

The **mbrlen** function may fail if:

EINVAL	**ps** points to an object that contains an invalid conversion state.
EILSEQ	Invalid character sequence is detected.

Implementation Specifics

This subroutine is part of Base Operating System (BOS) subroutine.

Related Information

The **mbsinit** subroutine.

The **mbrtowc** subroutine.

The **wchar.h** file.

mbrtowc Subroutine

Purpose

Convert a character to a wide–character code (restartable)

Library

Standard Library (**libc.a**)

Syntax

```
#include <wchar.h>

size_t mbrtowc (wchar_t * pwc, const char * s, size_t n,
mbstate_t * ps) ;
```

Description

If *s* is a null pointer, the **mbrtowc** function is equivalent to the call:

```
mbrtowc(NULL, '''', 1, ps)
```

In this case, the values of the arguments **pwc** and **n** are ignored.

If *s* is not a null pointer, the **mbrtowc** function inspects at most *n* bytes beginning at the byte pointed to by *s* to determine the number of bytes needed to complete the next character (including any shift sequences). If the function determines that the next character is completed, it determines the value of the corresponding wide–character and then, if *pwc* is not a null pointer, stores that value in the object pointed to by *pwc.* If the corresponding wide–character is the null wide–character, the resulting state described is the initial conversion state.

If *ps* is a null pointer, the **mbrtowc** function uses its own internal **mbstate_t** object, which is initialized at program startup to the initial conversion state. Otherwise, the **mbstate_t** object pointed to by ps is used to completely describe the current conversion state of the associated character sequence. The implementation will behave as if no function defined in this specification calls **mbrtowc**.

The behavior of this function is affected by the LC_CTYPE category of the current locale.

Return Values

The **mbrtowc** function returns the first of the following that applies:

0	If the next n or fewer bytes complete the character that corresponds to the null wide–character (which is the value stored).
positive	If the next n or fewer bytes complete a valid character (which is the value stored); the value returned is the number of bytes that complete the character.
(size_t)–2	If the next n bytes contribute to an incomplete but potentially valid character, and all n bytes have been processed (no value is stored). When n has at least the value of the MB_CUR_MAX macro, this case can only occur if s points at a sequence of redundant shift sequences (for implementations with state–dependent encodings).
(size_t)–1	If an encoding error occurs, in which case the next n or fewer bytes do not contribute to a complete and valid character (no value is stored). In this case, EILSEQ is stored in errno and the conversion state is undefined.

Error Codes

The **mbrtowc** function may fail if:

EINVAL	ps points to an object that contains an invalid conversion state.
EILSEQ	Invalid character sequence is detected.

Implementation Specifics

This subroutine is part of Base Operating System (BOS) subroutine.

Related Information

The **mbsinit** subroutine.

The **wchar.h** file.

mbsadvance Subroutine

Purpose

Advances to the next multibyte character.

Note: The **mbsadvance** subroutine is specific to the manufacturer. It is not defined in the POSIX, ANSI, or X/Open standards. Use of this subroutine may affect portability.

Library

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

Syntax

```
#include <mbstr.h>

char *mbsadvance (S)
const char *S;
```

Description

The **mbsadvance** subroutine locates the next character in a multibyte character string. The **LC_CTYPE** category affects the behavior of the **mbsadvance** subroutine.

Parameters

S	Contains a multibyte character string.

Return Values

If the *S* parameter is not a null pointer, the **mbsadvance** subroutine returns a pointer to the next multibyte character in the string pointed to by the *S* parameter. The character at the head of the string pointed to by the *S* parameter is skipped. If the *S* parameter is a null pointer or points to a null string, a null pointer is returned.

Examples

To find the next character in a multibyte string, use the following:

```
#include <mbstr.h>
#include <locale.h>
#include <stdlib.h>

main()
{
   char *mbs, *pmbs;
   (void) setlocale(LC_ALL, ””);
   /*
   ** Let mbs point to the beginning of a multi-byte string.
   */
   pmbs = mbs;
   while(pmbs){
      pmbs = mbsadvance(mbs);
      /* pmbs points to the next multi-byte character
      ** in mbs */
}
```

Implementation Specifics

This subroutine is part of Base Operating System (BOS) Runtime.

Related Information

The **mbsinvalid** subroutine.

National Language Support Overview for Programming, Subroutines Overview in *AIX General Programming Concepts : Writing and Debugging Programs*.

mbscat, mbscmp, or mbscpy Subroutine

Purpose

Performs operations on multibyte character strings.

Library

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

Syntax

```
#include <mbstr.h>

char *mbscat(MbString1, MbString2)
char *MbString1, *MbString2;

int mbscmp(MbString1, MbString2)
char *MbString1, *MbString2;

char *mbscpy(MbString1, MbString2)
char *MbString1, *MbString2;
```

Description

The **mbscat**, **mbscmp**, and **mbscpy** subroutines operate on null–terminated multibyte character strings.

The **mbscat** subroutine appends multibyte characters from the *MbString2* parameter to the end of the *MbString1* parameter, appends a null character to the result, and returns *MbString1*.

The **mbscmp** subroutine compares multibyte characters based on their collation weights as specified in the **LC_COLLATE** category. The **mbscmp** subroutine compares the *MbString1* parameter to the *MbString2* parameter, and returns an integer greater than 0 if *MbString1* is greater than *MbString2*. It returns 0 if the strings are equivalent and returns an integer less than 0 if *MbString1* is less than *MbString2*.

The **mbscpy** subroutine copies multibyte characters from the *MbString2* parameter to the *MbString1* parameter and returns *MbString1*. The copy operation terminates with the copying of a null character.

Implementation Specifics

These subroutines are part of Base Operating System (BOS) Runtime.

Related Information

The **mbsncat**, **mbsncmp**, **mbsncpy** subroutine, **wcscat**, **wcscmp**, **wcscpy** subroutine.

National Language Support Overview for Programming, Subroutines Overview in *AIX General Programming Concepts : Writing and Debugging Programs*.

mbschr Subroutine

Purpose

Locates a character in a multibyte character string.

Library

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

Syntax

```
#include <mbstr.h>

char *mbschr(MbString, MbCharacter)
char *MbString;
mbchar_t MbCharacter;
```

Description

The **mbschr** subroutine locates the first occurrence of the value specified by the *MbCharacter* parameter in the string pointed to by the *MbString* parameter. The *MbCharacter* parameter specifies a multibyte character represented as an integer. The terminating null character is considered to be part of the string.

The **LC_CTYPE** category affects the behavior of the **mbschr** subroutine.

Parameters

MbString	Points to a multibyte character string.
MbCharacter	Specifies a multibyte character represented as an integer.

Return Values

The **mbschr** subroutine returns a pointer to the value specified by the *MbCharacter* parameter within the multibyte character string, or a null pointer if that value does not occur in the string.

Implementation Specifics

This subroutine is part of Base Operating System (BOS) Runtime.

Related Information

The **mbspbrk** subroutine, **mbsrchr** subroutine, **mbstomb** subroutine, **wcschr** subroutine.

National Language Support Overview for Programming, Subroutines Overview in *AIX General Programming Concepts : Writing and Debugging Programs*.

mbsinit Subroutine

Purpose

Determine conversion object status.

Library

Standard Library (**libc.a**)

Syntax

```
#include <wchar.h>

int mbsinit (const mbstate_t * p) ;
```

Description

If *ps* is not a null pointer, the **mbsinit** function determines whether the object pointed to by ps describes an initial conversion state.

Return Values

The **mbsinit** function returns non–zero if **ps** is a null pointer, or if the pointed–to object describes an initial conversion state; otherwise, it returns zero.

If an **mbstate_t** object is altered by any of the functions described as `restartable` , and is then used with a different character sequence, or in the other conversion direction, or with a different LC_CTYPE category setting than on earlier function calls, the behavior is undefined.

Implementation Specifics

The **mbstate_t** object is used to describe the current conversion state from a particular character sequence to a wide–character sequence (or vice versa) under the rules of a particular setting of the LC_CTYPE category of the current locale.

The initial conversion state corresponds, for a conversion in either direction, to the beginning of a new character sequence in the initial shift state. A zero valued **mbstate_t** object is at least one way to describe an initial conversion state. A zero valued **mbstate_t** object can be used to initiate conversion involving any character sequence, in any LC_CTYPE category setting.

Related Information

The **mbrlen** subroutine, **mbrtowc** subroutine, **wcrtomb** subroutine, **mbsrtowcs** subroutine, **wcsrtombs** subroutine.

The **wchar.h** file.

mbsinvalid Subroutine

Purpose

Validates characters of multibyte character strings.

Note: The **mbsinvalid** subroutine is specific to the manufacturer. It is not defined in the POSIX, ANSI, or X/Open standards. Use of this subroutine may affect portability.

Library

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

Syntax

```
#include <mbstr.h>

char *mbsinvalid (S)
const char *S;
```

Description

The **mbsinvalid** subroutine examines the string pointed to by the *S* parameter to determine the validity of characters. The **LC_CTYPE** category affects the behavior of the **mbsinvalid** subroutine.

Parameters

S	Contains a multibyte character string.

Return Values

The **mbsinvalid** subroutine returns a pointer to the byte following the last valid multibyte character in the *S* parameter. If all characters in the *S* parameter are valid multibyte characters, a null pointer is returned. If the *S* parameter is a null pointer, the behavior of the **mbsinvalid** subroutine is unspecified.

Implementation Specifics

This subroutine is part of Base Operating System (BOS) Runtime.

Related Information

The **mbsadvance** subroutine.

National Language Support Overview for Programming, Subroutines Overview in *AIX General Programming Concepts : Writing and Debugging Programs*.

mbslen Subroutine

Purpose

Determines the number of characters (code points) in a multibyte character string.

Note: The **mbslen** subroutine is specific to the manufacturer. It is not defined in the POSIX, ANSI, or X/Open standards. Use of this subroutine may affect portability.

Library

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

Syntax

```
#include <stdlib.h>

size_t mbslen(MbString)
char *mbs;
```

Description

The **mbslen** subroutine determines the number of characters (code points) in a multibyte character string. The **LC_CTYPE** category affects the behavior of the **mbslen** subroutine.

Parameters

MbString	Points to a multibyte character string.

Return Values

The **mbslen** subroutine returns the number of multibyte characters in a multibyte character string. It returns 0 if the *MbString* parameter points to a null character or if a character cannot be formed from the string pointed to by this parameter.

Implementation Specifics

This subroutine is part of Base Operating System (BOS) Runtime.

Related Information

The **mblen** subroutine, **mbstowcs** subroutine, **mbtowc** subroutine.

National Language Support Overview for Programming, Subroutines Overview, Understanding Multibyte Code and Wide Character Code Conversion Subroutines in *AIX General Programming Concepts : Writing and Debugging Programs*.

mbsncat, mbsncmp, or mbsncpy Subroutine

Purpose

Performs operations on a specified number of null–terminated multibyte characters.

Note: These subroutines are specific to the manufacturer. They are not defined in the POSIX, ANSI, or X/Open standards. Use of these subroutines may affect portability.

Library

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

Syntax

```
#include <mbstr.h>

char *mbsncat(MbString1, MbString2, Number)
char *MbString1, *MbString2;
size_t Number;

int mbsncmp(MbString1, MbString2, Number)
char *MbString1, *MbString2;
size_t Number;

char *mbsncpy(MbString1, MbString2, Number)
char *MbString1, *MbString2;
size_t Number;
```

Description

The **mbsncat**, **mbsncmp**, and **mbsncpy** subroutines operate on null–terminated multibyte character strings.

The **mbsncat** subroutine appends up to the specified maximum number of multibyte characters from the *MbString2* parameter to the end of the *MbString1* parameter, appends a null character to the result, and then returns the *MbString1* parameter.

The **mbsncmp** subroutine compares the collation weights of multibyte characters. The **LC_COLLATE** category specifies the collation weights for all characters in a locale. The **mbsncmp** subroutine compares up to the specified maximum number of multibyte characters from the *MbString1* parameter to the *MbString2* parameter. It then returns an integer greater than 0 if *MbString1* is greater than *MbString2*. It returns 0 if the strings are equivalent. It returns an integer less than 0 if *MbString1* is less than *MbString2*.

The **mbsncpy** subroutine copies up to the value of the *Number* parameter of multibyte characters from the *MbString2* parameter to the *MbString1* parameter and returns *MbString1*. If *MbString2* is shorter than *Number* multi–byte characters, *MbString1* is padded out to *Number* characters with null characters.

Parameters

MbString1	Contains a multibyte character string.
MbString2	Contains a multibyte character string.
Number	Specifies a maximum number of characters.

Implementation Specifics

These subroutines are part of Base Operating System (BOS) Runtime.

Related Information

The **mbscat** subroutine, **mbscmp** subroutine, **mbscpy** subroutine, **wcsncat** subroutine, **wcsncmp** subroutine, **wcsncpy** subroutine.

National Language Support Overview for Programming, Subroutines Overview in *AIX General Programming Concepts : Writing and Debugging Programs*.

mbspbrk Subroutine

Purpose

Locates the first occurrence of multibyte characters or code points in a string.

Note: The **mbspbrk** subroutine is specific to the manufacturer. It is not defined in the POSIX, ANSI, or X/Open standards. Use of this subroutine may affect portability.

Library

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

Syntax

```
#include <mbstr.h>

char *mbspbrk(MbString1, MbString2)
char *MbString1, *MbString2;
```

Description

The **mbspbrk** subroutine locates the first occurrence in the string pointed to by the *MbString1* parameter, of any character of the string pointed to by the *MbString2* parameter.

Parameters

MbString1	Points to the string being searched.
MbString2	Pointer to a set of characters in a string.

Return Values

The **mbspbrk** subroutine returns a pointer to the character. Otherwise, it returns a null character if no character from the string pointed to by the *MbString2* parameter occurs in the string pointed to by the *MbString1* parameter.

Implementation Specifics

This subroutine is part of Base Operating System (BOS) Runtime.

Related Information

The **mbschr** subroutine, **mbsrchr** subroutine, **mbstomb** subroutine, **wcspbrk** subroutine, **wcswcs** subroutine.

National Language Support Overview for Programming, Subroutines Overview in *AIX General Programming Concepts : Writing and Debugging Programs*.

mbsrchr Subroutine

Purpose

Locates a character or code point in a multibyte character string.

Library

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

Syntax

```
#include <mbstr.h>

char *mbsrchr(MbString, MbCharacter)
char *MbString;
int MbCharacter;
```

Description

The **mbschr** subroutine locates the last occurrence of the *MbCharacter* parameter in the string pointed to by the *MbString* parameter. The *MbCharacter* parameter is a multibyte character represented as an integer. The terminating null character is considered to be part of the string.

Parameters

MbString	Points to a multibyte character string.
MbCharacter	Specifies a multibyte character represented as an integer.

Return Values

The **mbsrchr** subroutine returns a pointer to the *MbCharacter* parameter within the multibyte character string. It returns a null pointer if *MbCharacter* does not occur in the string.

Implementation Specifics

This subroutine is part of Base Operating System (BOS) Runtime.

Related Information

The **mbschr** subroutine, **mbspbrk** subroutine, **mbstomb** subroutine, **wcsrchr** subroutine.

National Language Support Overview for Programming, Subroutines Overview in *AIX General Programming Concepts : Writing and Debugging Programs*.

mbsrtowcs Subroutine

Purpose

Convert a character string to a wide–character string (restartable).

Library

Standard Library (**libc.a**)

Syntax

```
#include <wchar.h>

size_t mbsrtowcs ((wchar_t * dst, const char ** src, size_t len,
mbstate_t * ps) ;
```

Description

The **mbsrtowcs** function converts a sequence of characters, beginning in the conversion state described by the object pointed to by *ps*, from the array indirectly pointed to by **src** into a sequence of corresponding wide–characters. If *dst* is not a null pointer, the converted characters are stored into the array pointed to by *dst*. Conversion continues up to and including a terminating null character, which is also stored. Conversion stops early in either of the following cases:

- When a sequence of bytes is encountered that does not form a valid character.
- When *len* codes have been stored into the array pointed to by **dst** (and *dst* is not a null pointer).

Each conversion takes place as if by a call to the **mbrtowc** function.

If *dst* is not a null pointer, the pointer object pointed to by src is assigned either a null pointer (if conversion stopped due to reaching a terminating null character) or the address just past the last character converted (if any). If conversion stopped due to reaching a terminating null character, and if *dst* is not a null pointer, the resulting state described is the initial conversion state.

If *ps* is a null pointer, the **mbsrtowcs** function uses its own internal **mbstate_t** object, which is initialised at program startup to the initial conversion state. Otherwise, the **mbstate_t** object pointed to by ps is used to completely describe the current conversion state of the associated character sequence. The implementation will behave as if no function defined in this specification calls **mbsrtowcs**.

The behavior of this function is affected by the LC_CTYPE category of the current locale.

Return Values

If the input conversion encounters a sequence of bytes that do not form a valid character, an encoding error occurs. In this case, the **mbsrtowcs** function stores the value of the macro EILSEQ in errno and returns (**size_t)–1**); the conversion state is undefined. Otherwise, it returns the number of characters successfully converted, not including the terminating null (if any).

Error Codes

The **mbsrtowcs** function may fail if:

EINVAL	ps points to an object that contains an invalid conversion state.
EILSEQ	Invalid character sequence is detected.

Implementation Specifics

This subroutine is part of Base Operating System (BOS) subroutine.

Related Information

The **mbsinit** subroutine.

The **mbrtowc** subroutine.

The **wchar.h** file.

mbstomb Subroutine

Purpose

Extracts a multibyte character from a multibyte character string.

Note: The **mbstomb** subroutine is specific to the manufacturer. It is not defined in the POSIX, ANSI, or X/Open standards. Use of this subroutine may affect portability.

Library

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

Syntax

```
#include <mbstr.h>

mbchar_t mbstomb (MbString)
const char *MbString;
```

Description

The **mbstomb** subroutine extracts the multibyte character pointed to by the *MbString* parameter from the multibyte character string. The **LC_CTYPE** category affects the behavior of the **mbstomb** subroutine.

Parameters

MbString	Contains a multibyte character string.

Return Values

The **mbstomb** subroutine returns the code point of the multibyte character as a **mbchar_t** data type. If an unusable multibyte character is encountered, a value of 0 is returned.

Implementation Specifics

This subroutine is part of Base Operating System (BOS) Runtime.

Related Information

The **mbschr** subroutine, **mbspbrk** subroutine, **mbsrchr** subroutine.

National Language Support Overview for Programming, Subroutines Overview in *AIX General Programming Concepts : Writing and Debugging Programs*.

mbstowcs Subroutine

Purpose

Converts a multibyte character string to a wide character string.

Library

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

Syntax

```
#include <stdlib.h>

size_t mbstowcs(WcString, String, Number)
wchar_t *WcString;
const char *String;
size_t Number;
```

Description

The **mbstowcs** subroutine converts the sequence of multibyte characters pointed to by the *String* parameter to wide characters and places the results in the buffer pointed to by the *WcString* parameter. The multibyte characters are converted until a null character is reached or until the number of wide characters specified by the *Number* parameter have been processed.

Parameters

WcString	Points to the area where the result of the conversion is stored.
String	Points to a multibyte character string.
Number	Specifies the maximum number of wide characters to be converted.

Return Values

The **mbstowcs** subroutine returns the number of wide characters converted, not including a null terminator, if any. If an invalid multibyte character is encountered, a value of –1 is returned. The *WcString* parameter does not include a null terminator if the value *Number* is returned.

If *WcString* is a null wide character pointer, the **mbstowcs** subroutine returns the number of elements required to store the wide character codes in an array.

Error Codes

The **mbstowcs** subroutine fails if the following occurs:

EILSEQ	Invalid byte sequence is detected.

Implementation Specifics

This subroutine is part of Base Operating System (BOS) Runtime.

Related Information

The **mblen** subroutine, **mbslen** subroutine, **mbtowc** subroutine, **wcstombs** subroutine, **wctomb** subroutine.

National Language Support Overview for Programming, Subroutines Overview, Understanding Multibyte Code and Wide Character Code Conversion Subroutines in *AIX General Programming Concepts : Writing and Debugging Programs*.

mbswidth Subroutine

Purpose

Determines the number of multibyte character string display columns.

Note: The **mbswidth** subroutine is specific to this manufacturer. It is not defined in the POSIX, ANSI, or X/Open standards. Use of this subroutine may affect portability.

Library

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

Syntax

```
#include <mbstr.h>

int mbswidth (MbString, Number)
const char *MbString;
size_t Number;
```

Description

The **mbswidth** subroutine determines the number of display columns required for a multibyte character string.

Parameters

MbString	Contains a multibyte character string.
Number	Specifies the number of bytes to read from the *s* parameter.

Return Values

The **mbswidth** subroutine returns the number of display columns that will be occupied by the *MbString* parameter if the number of bytes (specified by the *Number* parameter) read from the *MbString* parameter form valid multibyte characters. If the *MbString* parameter points to a null character, a value of 0 is returned. If the *MbString* parameter does not point to valid multibyte characters, –1 is returned. If the *MbString* parameter is a null pointer, the behavior of the **mbswidth** subroutine is unspecified.

Implementation Specifics

This subroutine is part of Base Operating System (BOS) Runtime.

Related Information

The **wcswidth** subroutine, **wcwidth** subroutine.

National Language Support Overview for Programming in *AIX General Programming Concepts : Writing and Debugging Programs*.

mbtowc Subroutine

Purpose

Converts a multibyte character to a wide character.

Library

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

Syntax

```
#include <stdlib.h>

int mbtowc (WideCharacter, String, Number)
wchar_t *WideCharacter;
const char *String;
size_t Number;
```

Description

The **mbtowc** subroutine converts a multibyte character to a wide character and returns the number of bytes of the multibyte character.

The **mbtowc** subroutine determines the number of bytes that comprise the multibyte character pointed to by the *String* parameter. It then converts the multibyte character to a corresponding wide character and, if the *WideCharacter* parameter is not a null pointer, places it in the location pointed to by the *WideCharacter* parameter. If the *WideCharacter* parameter is a null pointer, the **mbtowc** subroutine returns the number of converted bytes but does not change the *WideCharacter* parameter value. If the *WideCharacter* parameter returns a null value, the multibyte character is not converted.

Parameters

WideCharacter	Specifies the location where a wide character is to be placed.
String	Specifies a multibyte character.
Number	Specifies the maximum number of bytes of a multibyte character.

Return Values

The **mbtowc** subroutine returns a value of 0 if the *String* parameter is a null pointer. The subroutine returns a value of –1 if the bytes pointed to by the *String* parameter do not form a valid multibyte character before the number of bytes specified by the *Number* parameter (or fewer) have been processed. It then sets the **errno** global variable to indicate the error. Otherwise, the number of bytes comprising the multibyte character is returned.

Error Codes

The **mbtowc** subroutine fails if the following occurs:

EILSEQ	Invalid byte sequence is detected.

Implementation Specifics

This subroutine is part of Base Operating System (BOS) Runtime.

Related Information

The **mblen** subroutine, **mbslen** subroutine, **mbstowcs** subroutine, **wcstombs** subroutine, **wctomb** subroutine.

National Language Support Overview for Programming, Subroutines Overview, Understanding Multibyte Code, Wide Character Code Conversion Subroutines in *AIX General Programming Concepts : Writing and Debugging Programs*.

memccpy, memchr, memcmp, memcpy, memset or memmove Subroutine

Purpose

Performs memory operations.

Library

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

Syntax

```
#include <memory.h>

void *memccpy (Target, Source, C, N)
void *Target;
const void *Source;
int C;
size_t N;

void *memchr (S, C, N)
const void *S;
int C;
size_t N;

int memcmp (Target, Source, N)
const void *Target, *Source;
size_t N;

void *memcpy (Target, Source, N)
void *Target;
const void *Source;
size_t N;

void *memset (S, C, N)
void *S;
int C;
size_t N;

void *memmove (Target, Source, N)
void *Source;
const void *Target;
size_t N;
```

Description

The **memory** subroutines operate on memory areas. A memory area is an array of characters bounded by a count. The **memory** subroutines do not check for the overflow of any receiving memory area. All of the **memory** subroutines are declared in the **memory.h** file.

The **memccpy** subroutine copies characters from the memory area specified by the *Source* parameter into the memory area specified by the *Target* parameter. The **memccpy** subroutine stops after the first character specified by the *C* parameter (converted to the **unsigned char** data type) is copied, or after *N* characters are copied, whichever comes first. If copying takes place between objects that overlap, the behavior is undefined.

The **memcmp** subroutine compares the first *N* characters as the **unsigned cha**r data type in the memory area specified by the *Target* parameter to the first *N* characters as the **unsigned char** data type in the memory area specified by the *Source* parameter.

The **memcpy** subroutine copies *N* characters from the memory area specified by the *Source* parameter to the area specified by the *Target* parameter and then returns the value of the *Target* parameter.

The **memset** subroutine sets the first *N* characters in the memory area specified by the *S* parameter to the value of character *C* and then returns the value of the *S* parameter.

Like the **memcpy** subroutine, the **memmove** subroutine copies *N* characters from the memory area specified by the *Source* parameter to the area specified by the *Target* parameter. However, if the areas of the *Source* and *Target* parameters overlap, the move is performed nondestructively, proceeding from right to left.

Parameters

Target	Points to the start of a memory area.
Source	Points to the start of a memory area.
C	Specifies a character to search.
N	Specifies the number of characters to search.
S	Points to the start of a memory area.

Return Values

The **memccpy** subroutine returns a pointer to character *C* after it is copied into the area specified by the *Target* parameter, or a null pointer if the *C* character is not found in the first *N* characters of the area specified by the *Source* parameter.

The **memchr** subroutine returns a pointer to the first occurrence of the *C* character in the first *N* characters of the memory area specified by the *S* parameter, or a null pointer if the *C* character is not found.

The **memcmp** subroutine returns the following values:

Less than 0	If the value of the *Target* parameter is less than the values of the *Source* parameter.
Equal to 0	If the value of the *Target* parameter equals the value of the *Source* parameter.
Greater than 0	If the value of the *Target* parameter is greater than the value of the *Source* parameter.

Implementation Specifics

These subroutines are part of Base Operating System (BOS) Runtime.

The **memccpy** subroutine is not in the ANSI C library.

Related Information

The **swab** subroutine.

Subroutines Overview in *AIX General Programming Concepts : Writing and Debugging Programs*.

mincore Subroutine

Purpose

Determines residency of memory pages.

Library

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

Syntax

```
int mincore (addr, len, *vec)
caddr_t addr;
size_t len;
char *vec;
```

Description

The **mincore** subroutine returns the primary–memory residency status for regions created from calls made to the **mmap** subroutine. The status is returned as a character for each memory page in the range specified by the *addr* and *len* parameters. The least significant bit of each character returned is set to 1 if the referenced page is in primary memory. Otherwise, the bit is set to 0. The settings of the other bits in each character are undefined.

Parameters

addr	Specifies the starting address of the memory pages whose residency is to be determined. Must be a multiple of the page size returned by the **sysconf** subroutine using the **_SC_PAGE_SIZE** value for the *Name* parameter.
len	Specifies the length, in bytes, of the memory region whose residency is to be determined. If the *len* value is not a multiple of the page size as returned by the **sysconf** subroutine using the **_SC_PAGE_SIZE** value for the *Name* parameter, the length of the region is rounded up to the next multiple of the page size.
vec	Specifies the character array where the residency status is returned. The system assumes that the character array specified by the *vec* parameter is large enough to encompass a returned character for each page specified.

Return Values

When successful, the **mincore** subroutine returns 0. Otherwise, it returns –1 and sets the **errno** global variable to indicate the error.

Error Codes

If the **mincore** subroutine is unsuccessful, the **errno** global variable is set to one of the following values:

EFAULT A part of the buffer pointed to by the *vec* parameter is out of range or otherwise inaccessible.

EINVAL The *addr* parameter is not a multiple of the page size as returned by the **sysconf** subroutine using the **_SC_PAGE_SIZE** value for the *Name* parameter.

ENOMEM Addresses in the (*addr*, *addr* + *len*) range are invalid for the address space of the process, or specify one or more pages that are not mapped.

Implementation Specifics

This subroutine is part of Base Operating System (BOS) Runtime.

Related Information

The **mmap** subroutine, **sysconf** subroutine.

List of Memory Manipulation Services in *AIX General Programming Concepts : Writing and Debugging Programs*.

mkdir Subroutine

Purpose

Creates a directory.

Library

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

Syntax

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


int mkdir (Path, Mode)
const char *Path;
mode_t Mode;
```

Description

The **mkdir** subroutine creates a new directory.

The new directory has the following:

- The owner ID is set to the process–effective user ID.
- If the parent directory has the *SetFileGroupID* (**S_ISGID**) attribute set, the new directory inherits the group ID of the parent directory. Otherwise, the group ID of the new directory is set to the effective group ID of the calling process.
- Permission and attribute bits are set according to the value of the *Mode* parameter, with the following modifications:
  - All bits set in the process–file mode–creation mask are cleared.
  - The *SetFileUserID* and *Sticky* (**S_ISVTX**) attributes are cleared.
- If the *Path* variable names a symbolic link, the link is followed. The new directory is created where the variable pointed.

Parameters

Path	Specifies the name of the new directory. If Network File System (NFS) is installed on your system, this path can cross into another node. In this case, the new directory is created at that node. To execute the **mkdir** subroutine, a process must have search permission to get to the parent directory of the *Path* parameter as well as write permission in the parent directory itself.
Mode	Specifies the mask for the read, write, and execute flags for owner, group, and others. The *Mode* parameter specifies directory permissions and attributes. This parameter is constructed by logically ORing values described in the **sys/mode.h** file.

Return Values

Upon successful completion, the **mkdir** subroutine returns a value of 0. Otherwise, a value of –1 is returned, and the **errno** global variable is set to indicate the error.

Error Codes

The **mkdir** subroutine is unsuccessful and the directory is not created if one or more of the following are true:

EACCES	Creating the requested directory requires writing in a directory with a mode that denies write permission.
EEXIST	The named file already exists.
EROFS	The named file resides on a read–only file system.
ENOSPC	The file system does not contain enough space to hold the contents of the new directory or to extend the parent directory of the new directory.
EMLINK	The link count of the parent directory exceeds the maximum **(LINK_MAX)** number. **(LINK_MAX)** is defined in **limits.h** file.
ENAMETOOLONG	The *Path* parameter or a path component is too long and cannot be truncated.
ENOENT	A component of the path prefix does not exist or the *Path* parameter points to an empty string.
ENOTDIR	A component of the path prefix is not a directory.
EDQUOT	The directory in which the entry for the new directory is being placed cannot be extended, or an i–node or disk blocks could not be allocated for the new directory because the user's or group's quota of disk blocks or i–nodes on the file system containing the directory is exhausted.

The **mkdir** subroutine can be unsuccessful for other reasons. See "Appendix A. Base Operating System Error Codes for Services That Require Path–Name Resolution", on page A-1 for a list of additional errors.

If NFS is installed on the system, the **mkdir** subroutine is also unsuccessful if the following is true:

ETIMEDOUT	The connection timed out.

Implementation Specifics

This subroutine is part of Base Operating System (BOS) Runtime.

Related Information

The **chmod** subroutine, **mknod** subroutine, **rmdir** subroutine, **umask** subroutine.

The **chmod** command, **mkdir** command, **mknod** command.

Files, Directories, and File Systems for Programmers in *AIX General Programming Concepts : Writing and Debugging Programs*.

mknod or mkfifo Subroutine

Purpose

Creates an ordinary file, first–in–first–out (FIFO), or special file.

Library

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

Syntax

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

int mknod (const char *Path, mode_t Mode, dev_t Device)
char *Path;
int Mode;
dev_t Device;

int mkfifo (const char *Path, mode_t Mode)
const char *Path;
int Mode;
```

Description

The **mknod** subroutine creates a new regular file, special file, or FIFO file. Using the **mknod** subroutine to create file types (other than FIFO or special files) requires root user authority.

For the **mknod** subroutine to complete successfully, a process must have both search and write permission in the parent directory of the *Path* parameter.

The **mkfifo** subroutine is an interface to the **mknod** subroutine, where the new file to be created is a FIFO or special file. No special system privileges are required.

The new file has the following characteristics:

- File type is specified by the *Mode* parameter.
- Owner ID is set to the effective user ID of the process.
- Group ID of the file is set to the group ID of the parent directory if the *SetGroupID* attribute (**S_ISGID**) of the parent directory is set. Otherwise, the group ID of the file is set to the effective group ID of the calling process.
- Permission and attribute bits are set according to the value of the *Mode* parameter. All bits set in the file–mode creation mask of the process are cleared.

Upon successful completion, the **mkfifo** subroutine marks for update the `st_atime`, `st_ctime`, and `st_mtime` fields of the file. It also marks for update the `st_ctime` and `st_mtime` fields of the directory that contains the new entry.

If the new file is a character special file having the **S_IMPX** attribute (multiplexed character special file), when the file is used, additional path–name components can appear after the path name as if it were a directory. The additional part of the path name is available to the device driver of the file for interpretation. This feature provides a multiplexed interface to the device driver.

Parameters

Path	Names the new file. If Network File System (NFS) is installed on your system, this path can cross into another node.
Mode	Specifies the file type, attributes, and access permissions. This parameter is constructed by logically ORing values described in the **sys/mode.h** file.
Device	Specifies the ID of the device, which corresponds to the `st_rdev` member of the structure returned by the **statx** subroutine. This parameter is configuration–dependent and used only if the *Mode* parameter specifies a block or character special file. If the file you specify is a remote file, the value of the *Device* parameter must be meaningful on the node where the file resides.

Return Values

Upon successful completion, a value of 0 is returned. Otherwise, a value of –1 is returned and the **errno** global variable is set to indicate the error.

Error Codes

The **mknod** subroutine fails and the new file is not created if one or more of the following are true:

EEXIST	The named file exists.
EDQUOT	The directory in which the entry for the new file is being placed cannot be extended, or an i–node could not be allocated for the file because the user's or group's quota of disk blocks or i–nodes on the file system is exhausted.
EISDIR	The *Mode* parameter specifies a directory. Use the **mkdir** subroutine instead.
ENOSPC	The directory that would contain the new file cannot be extended, or the file system is out of file–allocation resources.
EPERM	The *Mode* parameter specifies a file type other than **S_IFIFO**, and the calling process does not have root user authority.
EROFS	The directory in which the file is to be created is located on a read–only file system.

The **mknod** and **mkfifo** subroutine can be unsuccessful for other reasons. See ”Appendix. A Base Operating System Error Codes for Services That Require Path–Name Resolution”, on page A-1 for a list of additional errors.

If NFS is installed on the system, the **mknod** subroutine can also fail if the following is true:

ETIMEDOUT	The connection timed out.

Implementation Specifics

These subroutines are part of Base Operating System (BOS) Runtime.

Related Information

The **chmod** subroutine, **mkdir** subroutine, **open**, **openx**, or **creat** subroutine, **statx** subroutine, **umask** subroutine.

The **chmod** command, **mkdir** command, **mknod** command.

The **mode.h** file, **types.h** file.

Files, Directories, and File Systems for Programmers in *AIX General Programming Concepts : Writing and Debugging Programs*.

mktemp or mkstemp Subroutine

Purpose

Constructs a unique file name.

Libraries

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

Berkeley Compatibility Library (**libbsd.a**)

Syntax

```
#include <stdlib.h>

char *mktemp (Template)
char *Template;

int mkstemp (Template)
char *Template;
```

Description

The **mktemp** subroutine replaces the contents of the string pointed to by the *Template* parameter with a unique file name.

Note: The **mktemp** subroutine creates a filename and checks to see if the file exist. It that file does not exist, the name is returned. If the user calls **mktemp** twice without creating a file using the name returned by the first call to **mktemp**, then the second **mktemp** call may return the same name as the first **mktemp** call since the name does not exist.

To avoid this, either create the file after calling **mktemp** or use the **mkstemp** subroutine. The **mkstemp** subroutine creates the file for you.

Parameters

Template	Points to a string to be replaced with a unique file name. The string in the *Template* parameter is a file name with up to six trailing X's. Since the system randomly generates a six–character string to replace the X's, it is recommended that six trailing X's be used.

Return Values

Upon successful completion, the **mktemp** subroutine returns the address of the string pointed to by the *Template* parameter.

If the string pointed to by the *Template* parameter contains no X's, and if it is an existing file name, the *Template* parameter is set to a null character, and a null pointer is returned; if the string does not match any existing file name, the exact string is returned.

Upon successful completion, the **mkstemp** subroutine returns an open file descriptor. If the **mkstemp** subroutine fails, it returns a value of –1.

Implementation Specifics

These subroutines are part of Base Operating System (BOS) Runtime.

To get the BSD version of this subroutine, compile with Berkeley Compatibility Library (**libbsd.a**).

The **mkstemp** subroutine performs the same substitution to the template name and also opens the file for reading and writing.

In BSD systems, the **mkstemp** subroutine was intended to avoid a race condition between generating a temporary name and creating the file. Because the name generation in the operating system is more random, this race condition is less likely. BSD returns a file name of / (slash).

Former implementations created a unique name by replacing X's with the process ID and a unique letter.

Related Information

The **getpid** subroutine, **tmpfile** subroutine, **tmpnam** or **tempnam** subroutine.

Files, Directories, and File Systems for Programmers in *AIX General Programming Concepts : Writing and Debugging Programs*.

mmap or mmap64 Subroutine

Purpose

Maps a file–system object into virtual memory.

Library

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

Syntax

```
#include <sys/types.h>
#include <sys/mman.h>

void *mmap (addr, len, prot, flags, fildes, off)
void *addr;
size_t len;
int prot, flags, fildes;
off_t off;
```

Note: The **mmap64** subroutine applies to Version 4.2 and later releases.

```
void *mmap64 (addr, len, prot, flags, fildes, off)
void *addr;
size_t len;
int prot, flags, fildes;
off64_t off;
```

Description

Note: The **mmap64** subroutine applies to Version 4.2 and later releases.

Attention: A file–system object should not be simultaneously mapped using both the **mmap** and **shmat** subroutines. Unexpected results may occur when references are made beyond the end of the object.

The **mmap** subroutine creates a new mapped file or anonymous memory region by establishing a mapping between a process–address space and a file–system object. Care needs to be taken when using the **mmap** subroutine if the program attempts to map itself. If the page containing executing instructions is currently referenced as data through an mmap mapping, the program will hang. Use the –H4096 binder option, and that will put the executable text on page boundries. Then reset the file that contains the executable material, and view via an **mmap** mapping.

A region created by the **mmap** subroutine cannot be used as the buffer for read or write operations that involve a device. Similarly, an **mmap** region cannot be used as the buffer for operations that require either a **pin** or **xmattach** operation on the buffer.

Modifications to a file–system object are seen consistently, whether accessed from a mapped file region or from the **read** or **write** subroutine.

Child processes inherit all mapped regions from the parent process when the **fork** subroutine is called. The child process also inherits the same sharing and protection attributes for these mapped regions. A successful call to any **exec** subroutine will unmap all mapped regions created with the **mmap** subroutine.

The **mmap64** subroutine is identical to the **mmap** subroutine except that the starting offset for the file mapping is specified as a 64–bit value. This permits file mappings which start beyond **OFF_MAX**.

In the large file enabled programming environment, **mmap** is redefined to be **mmap64**.

If the application has requested SPEC1170 compliant behavior then the **st_atime** field of the mapped file is marked for update upon successful completion of the **mmap** call.

If the application has requested SPEC1170 compliant behavior then the **st_ctime** and **st_mtime** fields of a file that is mapped with **MAP_SHARED** and **PROT_WRITE** are marked for update at the next call to **msync** subroutine or **munmap** subroutine if the file has been modified.

Parameters

addr
: Specifies the starting address of the memory region to be mapped. When the **MAP_FIXED** flag is specified, this address must be a multiple of the page size returned by the **sysconf** subroutine using the **_SC_PAGE_SIZE** value for the *Name* parameter. A region is never placed at address zero, or at an address where it would overlap an existing region.

len
: Specifies the length, in bytes, of the memory region to be mapped. The system performs mapping operations over whole pages only. If the *len* parameter is not a multiple of the page size, the system will include in any mapping operation the address range between the end of the region and the end of the page containing the end of the region.

prot
: Specifies the access permissions for the mapped region. The **sys/mman.h** file defines the following access options:

PROT_READ	Region can be read.
PROT_WRITE	Region can be written.
PROT_EXEC	Region can be executed.
PROT_NONE	Region cannot be accessed.

The *prot* parameter can be the **PROT_NONE** flag, or any combination of the **PROT_READ** flag, **PROT_WRITE** flag, and **PROT_EXEC** flag logically ORed together. If the **PROT_NONE** flag is not specified, access permissions may be granted to the region in addition to those explicitly requested. However, write access will not be granted unless the **PROT_WRITE** flag is specified.

Note: The operating system generates a **SIGSEGV** signal if a program attempts an access that exceeds the access permission given to a memory region. For example, if the **PROT_WRITE** flag is not specified and a program attempts a write access, a **SIGSEGV** signal results.

If the region is a mapped file that was mapped with the **MAP_SHARED** flag, the **mmap** subroutine grants read or execute access permission only if the file descriptor used to map the file was opened for reading. It grants write access permission only if the file descriptor was opened for writing.

If the region is a mapped file that was mapped with the **MAP_PRIVATE** flag, the **mmap** subroutine grants read, write, or execute access permission only if the file descriptor used to map the file was opened for reading. If the region is an anonymous memory region, the **mmap** subroutine grants all requested access permissions.

fildes Specifies the file descriptor of the file–system object to be mapped. If the **MAP_ANONYMOUS** flag is set, the *fildes* parameter must be –1. After the successful completion of the **mmap** subroutine, the file specified by the *fildes* parameter may be closed without effecting the mapped region or the contents of the mapped file. Each mapped region creates a file reference, similar to an open file descriptor, which prevents the file data from being deallocated.

Note: The **mmap** subroutine supports the mapping of regular files only. An **mmap** call that specifies a file descriptor for a special file fails, returning the **ENODEV** error. An example of a file descriptor for a special file is one that might be used for mapping either I/O or device memory.

off Specifies the file byte offset at which the mapping starts. This offset must be a multiple of the page size returned by the **sysconf** subroutine using the **_SC_PAGE_SIZE** value for the *Name* parameter.

flags Specifies attributes of the mapped region. Values for the *flags* parameter are constructed by a bitwise–inclusive ORing of values from the following list of symbolic names defined in the **sys/mman.h** file:

MAP_FILE Specifies the creation of a new mapped file region by mapping the file associated with the *fildes* file descriptor. The mapped region can extend beyond the end of the file, both at the time when the **mmap** subroutine is called and while the mapping persists. This situation could occur if a file with no contents was created just before the call to the **mmap** subroutine, or if a file was later truncated. However, references to whole pages following the end of the file result in the delivery of a **SIGBUS** signal. Only one of the **MAP_FILE** and **MAP_ANONYMOUS** flags must be specified with the **mmap** subroutine.

MAP_ANONYMOUS Specifies the creation of a new, anonymous memory region that is initialized to all zeros. This memory region can be shared only with the descendants of the current process. When using this flag, the *fildes* parameter must be –1. Only one of the **MAP_FILE** and **MAP_ANONYMOUS** flags must be specified with the **mmap** subroutine.

MAP_ VARIABLE Specifies that the system select an address for the new memory region if the new memory region cannot be mapped at the address specified by the *addr* parameter, or if the *addr* parameter is null. Only one of the **MAP_VARIABLE** and **MAP_FIXED** flags must be specified with the **mmap** subroutine.

MAP_FIXED Specifies that the mapped region be placed exactly at the address specified by the *addr* parameter. If the application has requested SPEC1170 complaint behavior and the **mmap** request is successful, the mapping replaces any previous mappings for the process' pages in the specified range. If the application has not requested SPEC1170 compliant behavior and a previous mapping exists in the range then the request fails. Only one of the **MAP_VARIABLE** and **MAP_FIXED** flags must be specified with the **mmap** subroutine.

MAP_SHARED When the **MAP_SHARED** flag is set, modifications to the mapped memory region will be visible to other processes that have mapped the same region using this flag. If the region is a mapped file region, modifications to the region will be written to the file.

Only one of the **MAP_SHARED** or **MAP_PRIVATE** flags can be specified with the **mmap** subroutine. **MAP_PRIVATE** is the default setting when neither flag is specified.

MAP_PRIVATE When the **MAP_PRIVATE** flag is specified, modifications to the mapped region by the calling process are not visible to other processes that have mapped the same region. If the region is a mapped file region, modifications to the region are not written to the file.

If this flag is specified, the initial write reference to an object page creates a private copy of that page and redirects the mapping to the copy. Until then, modifications to the page by processes that have mapped the same region with the **MAP_SHARED** flag are visible.

Only one of the **MAP_SHARED** or **MAP_PRIVATE** flags can be specified with the **mmap** subroutine. **MAP_PRIVATE** is the default setting when neither flag is specified.

Return Values

If successful, the **mmap** subroutine returns the address at which the mapping was placed. Otherwise, it returns –1 and sets the **errno** global variable to indicate the error.

Error Codes

Under the following conditions, the **mmap** subroutine fails and sets the **errno** global variable to:

EACCES The file referred to by the *fildes* parameter is not open for read access, or the file is not open for write access and the **PROT_WRITE** flag was specified for a **MAP_SHARED** mapping operation. Or, the file to be mapped has enforced locking enabled and the file is currently locked.

EBADF The *fildes* parameter is not a valid file descriptor, or the **MAP_ANONYMOUS** flag was set and the *fildes* parameter is not –1.

EFBIG The mapping requested extends beyond the maximum file size associated with *fildes*.

EINVAL The *flags* or *prot* parameter is invalid, or the *addr* parameter or *off* parameter is not a multiple of the page size returned by the **sysconf** subroutine using the **_SC_PAGE_SIZE** value for the *Name* parameter.

EINVAL The application has requested SPEC1170 compliant behavior and the value of flags is invalid (neither **MAP_PRIVATE** nor **MAP_SHARED** is set).

EMFILE The application has requested SPEC1170 compliant behavior and the number of mapped regions would excedd and implementation–dependent limit (per process or per system).

ENODEV The *fildes* parameter refers to an object that cannot be mapped, such as a terminal.

ENOMEM There is not enough address space to map *len* bytes, or the application has not requested X/Open UNIX95 Specification compliant bahavior and the **MAP_FIXED** flag was set and part of the address–space range (*addr*, *addr+len*) is already allocated.

ENXIO The addresses specified by the range (*off*, *off*+*len*) are invalid for the *fildes* parameter.

EOVERFLOW The mapping requested extends beyond the offset maximum for the file description associated with *fildes*.

Implementation Specifics

This subroutine is part of Base Operating System (BOS) Runtime.

Related Information

The **exec** subroutine, **fork** subroutine, **read** subroutine, **shmat** subroutine, **sysconf** subroutine, **write** subroutine.

The **pin** kernel service, **xmattach** kernel service.

List of Memory Manipulation Services, List of Memory Mapping Services, Understanding Memory Mapping in *AIX General Programming Concepts : Writing and Debugging Programs*.

mntctl Subroutine

Purpose

Returns information about the mount status of the system.

Library

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

Syntax

```
#include <sys/mntctl.h>
#include <sys/vmount.h>

int mntctl (Command, Size, Buffer)
int Command;
int Size;
char *Buffer;
```

Description

The **mntctl** subroutine is used to query the status of virtual file systems (also known as *mounted* file systems).

Each virtual file system (VFS) is described by a **vmount** structure. This structure is supplied when the VFS is created by the **vmount** subroutine. The **vmount** structure is defined in the **sys/vmount.h** file.

Parameters

Command	Specifies the operation to be performed. Valid commands are defined in the **sys/vmount.h** file. At present, the only command is: **MCTL_QUERY** Query mount information.
Buffer	Points to a data area that will contain an array of **vmount** structures. This data area holds the information returned by the query command. Since the **vmount** structure is variable–length, it is necessary to reference the `vmt_length` field of each structure to determine where in the *Buffer* area the next structure begins.
Size	Specifies the length, in bytes, of the buffer pointed to by the *Buffer* parameter.

Return Values

If the **mntctl** subroutine is successful, the number of **vmount** structures copied into the *Buffer* parameter is returned. If the *Size* parameter indicates the supplied buffer is too small to hold the **vmount** structures for all the current VFSs, the **mntctl** subroutine sets the first word of the *Buffer* parameter to the required size (in bytes) and returns the value 0. If the **mntctl** subroutine otherwise fails, a value of –1 is returned, and the **errno** global variable is set to indicate the error.

Error Codes

The **mntctl** subroutine fails and the requested operation is not performed if one or both of the following are true:

EINVAL	The *Command* parameter is not **MCTL_QUERY**, or the *Size* parameter is not a positive value.
EFAULT	The *Buffer* parameter points to a location outside of the allocated address space of the process.

Implementation Specifics

This subroutine is part of Base Operating System (BOS) Runtime.

Related Information

The **uvmount** or **umount** subroutine, **vmount** or **mount** subroutine.

Files, Directories, and File Systems for Programmers in *AIX General Programming Concepts : Writing and Debugging Programs*.

moncontrol Subroutine

Purpose

Starts and stops execution profiling after initialization by the **monitor** subroutine.

Library

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

Syntax

```
#include <mon.h>

int moncontrol (Mode)
int Mode;
```

Description

The **moncontrol** subroutine starts and stops profiling after profiling has been initialized by the **monitor** subroutine. It may be used with either **–p** or **–pg** profiling. When **moncontrol** stops profiling, no output data file is produced. When profiling has been started by the **monitor** subroutine and the **exit** subroutine is called, or when the **monitor** subroutine is called with a value of 0, then profiling is stopped, and an output file is produced, regardless of the state of profiling as set by the **moncontrol** subroutine.

The **moncontrol** subroutine examines global and parameter data in the following order:

1. When the **_mondata.prof_type** global variable is neither –1 (**–p** profiling defined) nor +1 (**–pg** profiling defined), no action is performed, 0 is returned, and the function is considered complete.

   The global variable is set to –1 in the **mcrt0.o** file and to +1 in the **gcrt0.o** file and defaults to 0 when the **crt0.o** file is used.
2. When the *Mode* parameter is 0, profiling is stopped. For any other value, profiling is started.

   The following global variables are used in a call to the **profil** subroutine:

_mondata.*ProfBuf*	Buffer address
_mondata.*ProfBufSiz*	Buffer size/multirange flag
_mondata.*ProfLoPC*	PC offset for hist buffer – I/O limit
_mondata.*ProfScale*	PC scale/compute scale flag.

These variables are initialized by the **monitor** subroutine each time it is called to start profiling.

Parameters

Mode	Specifies whether to start (resume) or stop profiling.

Return Values

The **moncontrol** subroutine returns the previous state of profiling. When the previous state was STOPPED, a 0 is returned. When the previous state was STARTED, a 1 is returned.

Error Codes

When the **moncontrol** subroutine detects an error from the call to the **profil** subroutine, a –1 is returned.

Implementation Specifics

This subroutine is part of Base Operating System (BOS) Runtime.

Related Information

The **monitor** subroutine, **monstartup** subroutine, **profil** subroutine.

List of Memory Manipulation Services in *AIX General Programming Concepts : Writing and Debugging Programs*.

monitor Subroutine

Purpose

Starts and stops execution profiling using data areas defined in the function parameters.

Library

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

Syntax

```
#include <mon.h>

int monitor (LowProgramCounter, HighProgramCounter, Buffer,
BufferSize, NFunction)

OR

int monitor (NotZeroA, DoNotCareA, Buffer,-1, NFunction)

OR

int monitor((caddr_t)0)

caddr_t LowProgramCounter, HighProgramCounter;
HISTCOUNTER *Buffer;
int BufferSize, NFunction;
caddr_t NotZeroA, DoNotCareA;
```

Description

The **monitor** subroutine initializes the buffer area and starts profiling, or else stops profiling and writes out the accumulated profiling data. Profiling, when started, causes periodic sampling and recording of the program location within the program address ranges specified. Profiling also accumulates function call count data compiled with the **–p** or **–pg** option.

Executable programs created with the **cc –p** or **cc –pg** command automatically include calls to the **monitor** subroutine (through the **monstartup** and **exit** subroutines) to profile the complete user program, including system libraries. In this case, you do not need to call the **monitor** subroutine.

The **monitor** subroutine is called by the **monstartup** subroutine to begin profiling and by the **exit** subroutine to end profiling. The **monitor** subroutine requires a global data variable to define which kind of profiling, **–p** or **–pg**, is in effect. The **monitor** subroutine initializes four global variables that are used as parameters to the **profil** subroutine by the **moncontrol** subroutine:

- The **monitor** subroutine calls the **moncontrol** subroutine to start the profiling data gathering.
- The **moncontrol** subroutine calls the **profil** subroutine to start the system timer–driven program address sampling.
- The **prof** command processes the data file produced by **–p** profiling.
- The **gprof** command processes the data file produced by **–pg** profiling.

The **monitor** subroutine examines the global data and parameter data in this order:

1. When the **_mondata.prof_type** global variable is neither –1 (**–p** profiling defined) nor +1 (**–pg** profiling defined), an error is returned, and the function is considered complete.

   The global variable is set to –1 in the **mcrt0.o** file and to +1 in the **gcrt0.o** file, and defaults to 0 when the **crt0.o** file is used.

2. When the first parameter to the **monitor** subroutine is 0, profiling is stopped and the data file is written out.

   If **–p** profiling was in effect, then the file is named **mon.out**. If **–pg** profiling was in effect, the file is named **gmon.out**. The function is complete.

3. When the first parameter to the **monitor** subroutine is not , the **monitor** parameters and the profiling global variable, **_mondata.prof_type,** are examined to determine how to start profiling.

4.

   When the *BufferSize* parameter is not –1, a single program address range is defined for profiling, and the first **monitor** definition in the syntax is used to define the single program range.

5.

   When the *BufferSize* parameter is –1, multiple program address ranges are defined for profiling, and the second **monitor** definition in the syntax is used to define the multiple ranges. In this case, the *ProfileBuffer* value is the address of an array of **prof** structures. The size of the **prof** array is denoted by a zero value for the *HighProgramCounter* (`p_high`) field of the last element of the array. Each element in the array, except the last, defines a single programming address range to be profiled. Programming ranges must be in ascending order of the program addresses with ascending order of the **prof** array index. Program ranges may not overlap.

   The buffer space defined by the `p_buff` and `p_bufsize` fields of all of the **prof** entries must define a single contiguous buffer area. Space for the function–count data is included in the first range buffer. Its size is defined by the *NFunction* parameter. The `p_scale` entry in the **prof** structure is ignored. The **prof** structure is defined in the **mon.h** file. It contains the following fields:

```
caddr_t p_low;          /* low sampling address */
caddr_t p_high;         /* high sampling address */
HISTCOUNTER *p_buff;    /* address of sampling buffer */
int p_bufsize;          /* buffer size- monitor/HISTCOUNTERs,\
                          profil/bytes */
uint p_scale;           /* scale factor */
```

Parameters

LowProgramCounter (**prof** name: `p_low`)	Defines the lowest execution–time program address in the range to be profiled. The value of the *LowProgramCounter* parameter cannot be 0 when using the **monitor** subroutine to begin profiling.
HighProgramCounter (**prof** name: `p_high`)	Defines the next address after the highest–execution time program address in the range to be profiled. The program address parameters may be defined by function names or address expressions. If defined by a function name, then a function name expression must be used to dereference the function pointer to get the address of the first instruction in the function. This is required because the function reference in this context produces the address of the function descriptor. The first field of the descriptor is the address of the function code. See the examples for typical expressions to use.
Buffer (**prof** name: `p_buff`)	Defines the beginning address of an array of *BufferSize* `HISTCOUNTER` s to be used for data collection. This buffer includes the space for the program address–sampling counters and the function–count data areas. In the case of a multiple range specification, the space for the function–count data area is included at the beginning of the first range in the *BufferSize* specification.
BufferSize (**prof** name: `p_bufsize`)	Defines the size of the buffer in number of `HISTCOUNTER` s. Each counter is of type `HISTCOUNTER` (defined as short in the **mon.h** file). When the buffer includes space for the function–count data area (single range specification and first range of a multi–range specification) the *NFunction* parameter defines the space to be used for the function count data, and the remainder is used for program–address sampling counters for the range defined. The scale for the **profil** call is calculated from the number of counters available for program address–sample counting and the address range defined by the *LowProgramCounter* and *HighProgramCounter* parameters. See the **mon.h** file.

NFunction
Defines the size of the space to be used for the function–count data area. The space is included as part of the first (or only) range buffer.

When **–p** profiling is defined, the *NFunction* parameter defines the maximum number of functions to be counted. The space required for each function is defined to be:

```
sizeof(struct poutcnt)
```

The **poutcnt** structure is defined in the **mon.h** file. The total function–count space required is:

```
NFunction * sizeof(struct poutcnt)
```

When **–pg** profiling is defined, the *NFunction* parameter defines the size of the space (in bytes) available for the function–count data structures, as follows:

```
 range = HighProgramCounter –
LowProgramCounter;
 tonum = TO_NUM_ELEMENTS( range );
 if ( tonum < MINARCS ) tonum = MINARCS;
 if ( tonum > TO_MAX-1 ) tonum = TO_MAX-1;
 tosize = tonum * sizeof( struct tostruct );
 fromsize = FROM_STG_SIZE( range );
 rangesize = tosize + fromsize +
sizeof(struct gfctl);
```

This is computed and summed for all defined ranges. In this expression, the functions and variables in capital letters as well as the structures are defined in the **mon.h** file.

NotZeroA
Specifies a value of parameter 1, which is any value except 0. Ignored when it is not zero.

DoNotCareA
Specifies a value of parameter 2, of any value, which is ignored.

Return Values

The **monitor** subroutine returns 0 upon successful completion.

Error Codes

If an error is found, the **monitor** subroutine sends an error message to **stderr** and returns –1.

Examples

1. This example shows how to profile the main load module of a program with **–p** profiling:

```
#include <sys/types.h>
#include <mon.h>
main()
{
extern caddr_t etext;  /*system end of main module text symbol*/
extern int start();    /*first function in main program*/
extern struct monglobal _mondata; /*profiling global variables*/
struct desc {          /*function descriptor fields*/
    caddr_t begin;     /*initial code address*/
    caddr_t toc;       /*table of contents address*/
    caddr_t env;       /*environment pointer*/
} ;                   /*function descriptor structure*/
struct desc *fd;       /*pointer to function descriptor*/
int rc;                /*monitor return code*/
int range;             /*program address range for profiling*/
int numfunc;           /*number of functions*/
HISTCOUNTER *buffer;    /*buffer address*/
int numtics;        /*number of program address sample counters*/
int BufferSize;  /*total buffer size in numbers of HISTCOUNTERs*/
fd = (struct desc*)start; /*init descriptor pointer to start\
 function*/
numfunc = 300;           /*arbitrary number for example*/
range = etext - fd->begin; /*compute program address range*/
numtics =NUM_HIST_COUNTERS(range); /*one counter for each 4 byte\
 inst*/
BufferSize = numtics + ( numfunc*sizeof (struct poutcnt) \
 HIST_COUNTER_SIZE );     /*allocate buffer space*/
buffer = (HISTCOUNTER *) malloc (BufferSize * HIST_COUNTER_SIZE);
if ( buffer == NULL )  /*didn't get space, do error recovery\
 here*/
   return(-1);
_mondata.prof_type = _PROF_TYPE_IS_P; /*define -p profiling*/
rc = monitor( fd->begin, (caddr_t)etext, buffer, BufferSize, \
 numfunc);
/*start*/
if ( rc != 0 ) /*profiling did not start, do error recovery\
 here*/
   return(-1);
/*other code for analysis*/
rc = monitor( (caddr_t)0);  /*stop profiling and write data file\
 mon.out*/
if ( rc != 0 ) /*did not stop correctly, do error recovery here*/
   return (-1);
}
```

2. This example profiles the main program and the **libc.a** shared library with **–p** profiling. The range of addresses for the shared **libc.a** is assumed to be:

```
low = d0300000
```

```
high = d0312244
```

These two values can be determined from the **loadquery** subroutine at execution time, or by using a debugger to view the loaded programs' execution addresses and the loader map.

```
#include <sys/types.h>
#include <mon.h>
main()
{
extern caddr_t etext;   /*system end of text symbol*/
extern int start();    /*first function in main program*/
extern struct monglobal _mondata; /*profiling global variables*/
struct prof pb[3];     /*prof array of 3 to define 2 ranges*/
int rc;               /*monitor return code*/
int range;            /*program address range for profiling*/
int numfunc;          /*number of functions to count (max)*/
int numtics;          /*number of sample counters*/
int num4fcnt;   /*number of HISTCOUNTERs used for fun cnt space*/
int BufferSize1;      /*first range BufferSize*/
int BufferSize2;      /*second range BufferSize*/
caddr_t liblo=0xd0300000;  /*lib low address (example only)*/
caddr_t libhi=0xd0312244;  /*lib high address (example only)*/
numfunc = 400;           /*arbitrary number for example*/
/*compute first range buffer size*/
range = etext - *(uint *) start;   /*init range*/
numtics = NUM_HIST_COUNTERS( range );
/*one counter for each 4 byte inst*/
num4fcnt = numfunc*sizeof( struct poutcnt )/HIST_COUNTER_SIZE;
BufferSize1 = numtics + num4fcnt;
/*compute second range buffer size*/
range = libhi-liblo;
BufferSize2 = range / 12; /*counter for every 12 inst bytes for\
 a change*/
/*allocate buffer space - note: must be single contiguous\
 buffer*/
pb[0].p_buff = (HISTCOUNTER *)malloc( (BufferSize1 +BufferSize2)\
 *HIST_COUNTER_SIZE);
if ( pb[0].p_buff == NULL )   /*didn't get space - do error\
 recovery here* ;/
    return(-1);
/*set up the first range values*/
pb[0].p_low = *(uint*)start;       /*start of main module*/
pb[0].p_high = (caddr_t)etext;     /*end of main module*/
pb[0].p_BufferSize = BufferSize1;  /*prog addr cnt space + \
func cnt space*/
/*set up last element marker*/
pb[2].p_high = (caddr_t)0;
_mondata.prof_type = _PROF_TYPE_IS_P;   /*define -p\
profiling*/
rc = monitor( (caddr_t)1, (caddr_t)1, pb, -1, numfunc); \
 /*start*/
if ( rc != 0 )   /*profiling did not start - do error recovery\
 here*/
   return (-1);
/*other code for analysis ...*/
rc = monitor( (caddr_t)0);   /*stop profiling and write data \
file mon.out*/
if ( rc != 0 )   /*did not stop correctly - do error recovery\
 here*/
    return (-1);
```

3. This example shows how to profile contiguously loaded functions beginning at `zit` up to but not including `zot` with **–pg** profiling:

```
#include <sys/types.h>
#include <mon.h>
main()
{
extern zit();            /*first function to profile*/
extern zot();            /*upper bound function*/
extern struct monglobal _mondata; /*profiling global variables*/
int rc;                 /*monstartup return code*/
_mondata.prof_type = _PROF_TYPE_IS_PG; /*define -pg profiling*/
/*Note cast used to obtain function code addresses*/
rc = monstartup(*(uint *)zit,*(uint *)zot); /*start*/
if ( rc != 0 ) /*profiling did not start, do error recovery\
 here*/
   return(-1);
/*other code for analysis ...*/
exit(0);    /*stop profiling and write data file gmon.out*/
}
```

Files

mon.out	Data file for **–p** profiling.
gmon.out	Data file for **–pg** profiling.
/usr/include/mon.h	Defines the **_mondata.prof_type** global variable in the **monglobal** data structure, the **prof** structure, and the functions referred to in the previous examples.

Implementation Specifics

This subroutine is part of Base Operating System (BOS) Runtime.

Related Information

The **moncontrol** subroutine, **monstartup** subroutine, **profil** subroutine.

The **gprof** command, **prof** command.

The **_end**, **_etext**, or **_edata** Identifier.

List of Memory Manipulation Services in *AIX General Programming Concepts : Writing and Debugging Programs*.

monstartup Subroutine

Purpose

Starts and stops execution profiling using default–sized data areas.

Library

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

Syntax

```
#include <mon.h>

int monstartup (LowProgramCounter, HighProgramCounter)

OR

int monstartup((caddr_t)-1), (caddr_t) FragBuffer)

OR

int monstartup((caddr_t)-1, (caddr_t)0)

caddr_t LowProgramCounter;
caddr_t HighProgramCounter;
```

Description

The **monstartup** subroutine allocates data areas of default size and starts profiling. Profiling causes periodic sampling and recording of the program location within the program address ranges specified, and accumulation of function–call count data for functions that have been compiled with the **–p** or **–pg** option.

Executable programs created with the **cc –p** or **cc –pg** command automatically include a call to the **monstartup** subroutine to profile the complete user program, including system libraries. In this case, you do not need to call the **monstartup** subroutine.

The **monstartup** subroutine is called by the **mcrt0.o** (**–p**) file or the **gcrt0.o** (**–pg**) file to begin profiling. The **monstartup** subroutine requires a global data variable to define whether **–p** or **–pg** profiling is to be in effect. The **monstartup** subroutine calls the **monitor** subroutine to initialize the data areas and start profiling.

The **prof** command is used to process the data file produced by **–p** profiling. The **gprof** command is used to process the data file produced by **–pg** profiling.

The **monstartup** subroutine examines the global and parameter data in the following order:

1. When the **_mondata.prof_type** global variable is neither –1 (**–p** profiling defined) nor +1 (**–pg** profiling defined), an error is returned and the function is considered complete.

   The global variable is set to –1 in the **mcrt0.o** file and to +1 in the **gcrt0.o** file, and defaults to 0 when **crt0.o** is used.

2. When the *LowProgramCounter* value is not –1:

   – A single program address range is defined for profiling

   AND

   – The first **monstartup** definition in the syntax is used to define the program range.

3. When the *LowProgramCounter* value is –1 and the *HighProgramCounter* value is not 0:

   – Multiple program address ranges are defined for profiling

   AND

- The second **monstartup** definition in the syntax is used to define multiple ranges. The *HighProgramCounter* parameter*,* in this case, is the address of a **frag** structure array. The **frag** array size is denoted by a zero value for the *HighProgramCounter* (`p_high`) field of the last element of the array. Each array element except the last defines one programming address range to be profiled. Programming ranges must be in ascending order of the program addresses with ascending order of the **prof** array index. Program ranges may not overlap.

4. When the *LowProgramCounter* value is –1 and the *HighProgramCounter* value is 0:

   - The whole program is defined for profiling

   AND

   - The third **monstartup** definition in the syntax is used. The program ranges are determined by **monstartup** and may be single range or multirange.

Parameters

LowProgramCounter (**frag** name: `p_low`)	Defines the lowest execution–time program address in the range to be profiled.
HighProgramCounter(**frag** name: `p_high`)	Defines the next address after the highest execution–time program address in the range to be profiled. The program address parameters may be defined by function names or address expressions. If defined by a function name, then a function name expression must be used to dereference the function pointer to get the address of the first instruction in the function. This is required because the function reference in this context produces the address of the function descriptor. The first field of the descriptor is the address of the function code. See the examples for typical expressions to use.
FragBuffer	Specifies the address of a frag structure array.

Examples

1. This example shows how to profile the main load module of a program with **–p** profiling:

```
#include <sys/types.h>
#include <mon.h>
main()
{
extern caddr_t etext;      /*system end of text
symbol*/
extern int start();      /*first function in main\
             program*/
extern struct monglobal _mondata; /*profiling global variables*/
struct desc {         /*function
descriptor fields*/
   caddr_t begin;      /*initial code
address*/
   caddr_t toc;      /*table of contents
address*/
   caddr_t env;      /*environment
pointer*/
 }
;            /*function
descriptor structure*/
struct desc *fd;      /*pointer to function\
             descriptor*/
int  rc;         /*monstartup
return code*/
fd = (struct desc *)start;   /*init descriptor pointer to\
             start
function*/
_mondata.prof_type = _PROF_TYPE_IS_P;   /*define -p profiling*/
rc = monstartup( fd->begin, (caddr_t) &etext);   /*start*/
if ( rc != 0 )         /*profiling did
not start - do\
            error
recovery here*/   return(-1);
            /*other code
for analysis ...*/
return(0);         /*stop profiling and
write data\
             file
mon.out*/
}
```

2. This example shows how to profile the complete program with **–p** profiling:

```
#include <sys/types.h>
#include <mon.h>
main()
{
extern struct monglobal _mondata;   /*profiling global\
              &
nbsp;  variables*/
int  rc;         /*monstartup
return code*/
_mondata.prof_type = _PROF_TYPE_IS_P;   /*define -p profiling*/
rc = monstartup( (caddr_t)-1, (caddr_t)0);   /*start*/
if ( rc != 0 )          /*profiling did
not start -\
              &
nbsp; do error recovery here*/
   return (-1);
            /*other code
for analysis ...*/
return(0);         /*stop profiling and
write data\
             file
mon.out*/
}
```

3. This example shows how to profile contiguously loaded functions beginning at `zit` up to but not including `zot` with **–pg** profiling:

```
#include <sys/types.h>
#include <mon.h>
main()
{
extern zit();          /*first function
to profile*/
extern zot();          /*upper bound
function*/
extern struct monglobal _mondata; /*profiling global variables*/
int  rc;         /*monstartup
return code*/
_mondata.prof_type = _PROF_TYPE_IS_PG;   /*define -pg profiling*/
/*Note cast used to obtain function code addresses*/
rc = monstartup(*(uint *)zit,*(uint *)zot);   /*start*/
if ( rc != 0 )          /*profiling did
not start - do\
             error
recovery here*/
   return(-1);
            /*other code
for analysis ...*/
exit(0);   /*stop profiling and write data file gmon.out*/
}
```

Return Values

The **monstartup** subroutine returns 0 upon successful completion.

Error Codes

If an error is found, the **monstartup** subroutine outputs an error message to **stderr** and returns –1.

Implementation Specifics

This subroutine is part of Base Operating System (BOS) Runtime.

Files

mon.out	Data file for **–p** profiling.
gmon.out	Data file for **–pg** profiling.
mon.h	Defines the **_mondata.prof_type** variable in the **monglobal** data structure, the **prof** structure, and the functions referred to in the examples.

Related Information

The **moncontrol** subroutine, **monitor** subroutine, **profil** subroutine.

The **gprof** command, **prof** command.

The **_edata _end**, **_etext**, or **_edata** Identifier.

List of Memory Manipulation Services in *AIX General Programming Concepts : Writing and Debugging Programs*.

mprotect Subroutine

Purpose

Modifies access protections for memory mapping.

Library

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

Syntax

```
#include &lt;sys/types.h&gt;
#include &lt;sys/mman.h&gt;

int mprotect (addr, len, prot)
void *addr;
size_t len;
int prot;
```

Description

The **mprotect** subroutine modifies the access protection of a mapped file region or anonymous memory region created by the **mmap** subroutine.

Parameters

addr
: Specifies the address of the region to be modified. Must be a multiple of the page size returned by the **sysconf** subroutine using the **_SC_PAGE_SIZE** value for the *Name* parameter.

len
: Specifies the length, in bytes, of the region to be modified. If the *len* parameter is not a multiple of the page size returned by the **sysconf** subroutine using the **_SC_PAGE_SIZE** value for the *Name* parameter, the length of the region will be rounded off to the next multiple of the page size.

prot
: Specifies the new access permissions for the mapped region. Legitimate values for the *prot* parameter are the same as those permitted for the **mmap** subroutine, as follows:

PROT_READ	Region can be read.
PROT_WRITE	Region can be written.
PROT_EXEC	Region can be executed.
PROT_NONE	Region cannot be accessed.

Return Values

When successful, the **mprotect** subroutine returns 0. Otherwise, it returns –1 and sets the **errno** global variable to indicate the error.

Error Codes

Attention: If the **mprotect** subroutine is unsuccessful because of a condition other than that specified by the **EINVAL** error code, the access protection for some pages in the (*addr*, *addr* + *len*) range may have been changed.

If the **mprotect** subroutine is unsuccessful, the **errno** global variable may be set to one of the following values:

EACCES The *prot* parameter specifies a protection that conflicts with the access permission set for the underlying file.

EINVAL The *prot* parameter is invalid, or the *addr* parameter is not a multiple of the page size as returned by the **sysconf** subroutine using the **_SC_PAGE_SIZE** value for the *Name* parameter.

ENOMEM The application has requested X/Open UNIX95 Specification compliant behavior and addresses in the range are invalid for the address space of the process or specify one or more pages which are not mapped.

msem_init Subroutine

Purpose

Initializes a semaphore in a mapped file or shared memory region.

Library

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

Syntax

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

msemaphore *msem_init (Sem, InitialValue)
msemaphore *Sem;
int InitialValue;
```

Description

The **msem_init** subroutine allocates a new binary semaphore and initializes the state of the new semaphore.

If the value of the *InitialValue* parameter is **MSEM_LOCKED**, the new semaphore is initialized in the locked state. If the value of the *InitialValue* parameter is **MSEM_UNLOCKED**, the new semaphore is initialized in the unlocked state.

The **msemaphore** structure is located within a mapped file or shared memory region created by a successful call to the **mmap** subroutine and having both read and write access.

Whether a semaphore is created in a mapped file or in an anonymous shared memory region, any reference by a process that has mapped the same file or shared region, using an **msemaphore** structure pointer that resolved to the same file or start of region offset, is taken as a reference to the same semaphore.

Any previous semaphore state stored in the **msemaphore** structure is ignored and overwritten.

Parameters

Sem	Points to an **msemaphore** structure in which the state of the semaphore is stored.
Initial Value	Determines whether the semaphore is locked or unlocked at allocation.

Return Values

When successful, the **msem_init** subroutine returns a pointer to the initialized **msemaphore** structure. Otherwise, it returns a null value and sets the **errno** global variable to indicate the error.

Error Codes

If the **msem_init** subroutine is unsuccessful, the **errno** global variable is set to one of the following values:

EINVAL	Indicates the *InitialValue* parameter is not valid.
ENOMEM	Indicates a new semaphore could not be created.

Implementation Specifics

The **msem_init** subroutine is part of Base Operating System (BOS) Runtime.

Related Information

The **mmap** subroutine, **msem_lock** subroutine, **msem_remove** subroutine, **msem_unlock** subroutine.

List of Memory Mapping Services and Understanding Memory Mapping in *AIX General Programming Concepts : Writing and Debugging Programs*.

msem_lock Subroutine

Purpose

Locks a semaphore.

Library

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

Syntax

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

int msem_lock (Sem, Condition)
msemaphore *Sem;
int Condition;
```

Description

The **msem_lock** subroutine attempts to lock a binary semaphore.

If the semaphore is not currently locked, it is locked and the **msem_lock** subroutine completes successfully.

If the semaphore is currently locked, and the value of the *Condition* parameter is **MSEM_IF_NOWAIT**, the **msem_lock** subroutine returns with an error. If the semaphore is currently locked, and the value of the *Condition* parameter is 0, the **msem_lock** subroutine does not return until either the calling process is able to successfully lock the semaphore or an error condition occurs.

All calls to the **msem_lock** and **msem_unlock** subroutines by multiple processes sharing a common **msemaphore** structure behave as if the call were serialized.

If the **msemaphore** structure contains any value not resulting from a call to the **msem_init** subroutine, followed by a (possibly empty) sequence of calls to the **msem_lock** and **msem_unlock** subroutines, the results are undefined. The address of an **msemaphore** structure is significant. If the **msemaphore** structure contains any value copied from an **msemaphore** structure at a different address, the result is undefined.

Parameters

Sem	Points to an **msemaphore** structure that specifies the semaphore to be locked.
Condition	Determines whether the **msem_lock** subroutine waits for a currently locked semaphore to unlock.

Return Values

When successful, the **msem_lock** subroutine returns a value of 0. Otherwise, it returns a value of –1 and sets the **errno** global variable to indicate the error.

Error Codes

If the **msem_lock** subroutine is unsuccessful, the **errno** global variable is set to one of the following values:

EAGAIN	Indicates a value of **MSEM_IF_NOWAIT** is specified for the *Condition* parameter and the semaphore is already locked.
EINVAL	Indicates the *Sem* parameter points to an **msemaphore** structure specifying a semaphore that has been removed, or the *Condition* parameter is invalid.
EINTR	Indicates the **msem_lock** subroutine was interrupted by a signal that was caught.

Implementation Specifics

The **msem_lock** subroutine is part of Base Operating System (BOS) Runtime.

Related Information

The **msem_init** subroutine, **msem_remove** subroutine, **msem_unlock** subroutine.

List of Memory Mapping Services and Understanding Memory Mapping in *AIX General Programming Concepts : Writing and Debugging Programs*.

msem_remove Subroutine

Purpose

Removes a semaphore.

Library

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

Syntax

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

int msem_remove (Sem)
msemaphore *Sem;
```

Description

The **msem_remove** subroutine removes a binary semaphore. Any subsequent use of the **msemaphore** structure before it is again initialized by calling the **msem_init** subroutine will have undefined results.

The **msem_remove** subroutine also causes any process waiting in the **msem_lock** subroutine on the removed semaphore to return with an error.

If the **msemaphore** structure contains any value not resulting from a call to the **msem_init** subroutine, followed by a (possibly empty) sequence of calls to the **msem_lock** and **msem_unlock** subroutines, the result is undefined. The address of an **msemaphore** structure is significant. If the **msemaphore** structure contains any value copied from an **msemaphore** structure at a different address, the result is undefined.

Parameters

Sem	Points to an **msemaphore** structure that specifies the semaphore to be removed.

Return Values

When successful, the **msem_remove** subroutine returns a value of 0. Otherwise, it returns a –1 and sets the **errno** global variable to indicate the error.

Error Codes

If the **msem_remove** subroutine is unsuccessful, the **errno** global variable is set to the following value:

EINVAL	Indicates the *Sem* parameter points to an **msemaphore** structure that specifies a semaphore that has been removed.

Implementation Specifics

The **msem_remove** subroutine is part of Base Operating System (BOS) Runtime.

Related Information

The **msem_init** subroutine, **msem_lock** subroutine, **msem_unlock** subroutine.

List of Memory Mapping Services and Understanding Memory Mapping in *AIX General Programming Concepts : Writing and Debugging Programs*.

msem_unlock Subroutine

Purpose

Unlocks a semaphore.

Library

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

Syntax

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

int msem_unlock (Sem, Condition)
msemaphore *Sem;
int Condition;
```

Description

The **msem_unlock** subroutine attempts to unlock a binary semaphore.

If the semaphore is currently locked, it is unlocked and the **msem_unlock** subroutine completes successfully.

If the *Condition* parameter is 0, the semaphore is unlocked, regardless of whether or not any other processes are currently attempting to lock it. If the *Condition* parameter is set to the **MSEM_IF_WAITERS** value, and another process is waiting to lock the semaphore or it cannot be reliably determined whether some process is waiting to lock the semaphore, the semaphore is unlocked by the calling process. If the *Condition* parameter is set to the **MSEM_IF_WAITERS** value and no process is waiting to lock the semaphore, the semaphore will not be unlocked and an error will be returned.

Parameters

Sem	Points to an **msemaphore** structure that specifies the semaphore to be unlocked.
Condition	Determines whether the **msem_unlock** subroutine unlocks the semaphore if no other processes are waiting to lock it.

Return Values

When successful, the **msem_unlock** subroutine returns a value of 0. Otherwise, it returns a value of –1 and sets the **errno** global variable to indicate the error.

Error Codes

If the **msem_unlock** subroutine is unsuccessful, the **errno** global variable is set to one of the following values:

EAGAIN	Indicates a *Condition* value of **MSEM_IF_WAITERS** was specified and there were no waiters.
EINVAL	Indicates the *Sem* parameter points to an **msemaphore** structure specifying a semaphore that has been removed, or the *Condition* parameter is not valid.

Implementation Specifics

The **msem_unlock** subroutine is part of Base Operating System (BOS) Runtime.

Related Information

The **msem_init** subroutine, **msem_lock** subroutine, **msem_remove** subroutine.

List of Memory Mapping Services and Understanding Memory Mapping in *AIX General Programming Concepts : Writing and Debugging Programs*.

msgctl Subroutine

Purpose

Provides message control operations.

Library

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

Syntax

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

int msgctl (MessageQueueID,Command,Buffer)
int MessageQueueID, Command;
struct msqid_ds *Buffer;
```

Description

The **msgctl** subroutine provides a variety of message control operations as specified by the *Command* parameter and stored in the structure pointed to by the *Buffer* parameter. The **msqid_ds** structure is defined in the **sys**/**msg.h** file.

The following limits apply to the message queue:

- Maximum message size is 65,535 bytes for releases prior to AIX 4.1.5 and is 4 Megabytes for release 4.1.5 and later releases.
- Maximum number of messages per queue is 8192.
- Maximum number of message queue IDs is 4096 for AIX releases before 4.3.2 and 131072 for AIX 4.3.2 and following.
- Maximum number of bytes in a queue is 4 65,535 for releases prior to AIX 4.1.5 and is 4 Megabytes for release 4.1.5 and later releases.

Parameters

MessageQueueID Specifies the message queue identifier.

Command The following values for the *Command* parameter are available:

IPC_STAT Stores the current value of the above fields of the data structure associated with the *MessageQueueID* parameter into the **msqid_ds** structure pointed to by the *Buffer* parameter.

The current process must have read permission in order to perform this operation.

IPC_SET Sets the value of the following fields of the data structure associated with the *MessageQueueID* parameter to the corresponding values found in the structure pointed to by the *Buffer* parameter:

```
msg_perm.uid
msg_perm.gid
msg_perm.mode/*Only the low-order
nine bits*/
msg_qbytes
```

The effective user ID of the current process must have root user authority or its process ID must equal the value of the `msg_perm.uid` or `msg_perm.cuid` field in the data structure associated with the *MessageQueueID* parameter in order to perform this operation. To raise the value of the `msg_qbytes` field, the effective user ID of the current process must have root user authority.

IPC_RMID Removes the message queue identifier specified by the *MessageQueueID* parameter from the system and destroys the message queue and data structure associated with it. The effective user ID of the current process must have root user authority or be equal to the value of the `msg_perm.uid` or `msg_perm.cuid` field in the data structure associated with the *MessageQueueID* parameter to perform this operation.

Buffer Points to a **msqid_ds** structure.

Return Values

Upon successful completion, the **msgctl** subroutine returns a value of 0. Otherwise, a value of –1 is returned and the **errno** global variable is set to indicate the error.

Error Codes

The **msgctl** subroutine is unsuccessful if any of the following conditions is true:

EINVAL	The *Command* or *MessageQueueID* parameter is not valid.
EACCES	The *Command* parameter is equal to the **IPC_STAT** value, and the calling process was denied read permission.
EPERM	The *Command* parameter is equal to the **IPC_RMID** value and the effective user ID of the calling process does not have root user authority. Or, the *Command* parameter is equal to the **IPC_SET** value, and the effective user ID of the calling process is not equal to the value of the `msg_perm.uid` field or the `msg_perm.cuid` field in the data structure associated with the *MessageQueueID* parameter.
EPERM	The *Command* parameter is equal to the **IPC_SET** value, an attempt was made to increase the value of the `msg_qbytes` field, and the effective user ID of the calling process does not have root user authority.
EFAULT	The *Buffer* parameter points outside of the process address space.

Implementation Specifics

This subroutine is part of Base Operating System (BOS) Runtime.

Related Information

The **msgget** subroutine, **msgrcv** subroutine, **msgsnd** subroutine, **msgxrcv** subroutine.

msgget Subroutine

Purpose

Gets a message queue identifier.

Library

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

Syntax

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

int msgget (Key, MessageFlag)
key_t Key;
int MessageFlag;
```

Description

The **msgget** subroutine returns the message queue identifier associated with the specified *Key* parameter.

A message queue identifier, associated message queue, and data structure are created for the value of the *Key* parameter if one of the following conditions is true:

- The *Key* parameter is equal to the **IPC_PRIVATE** value.
- The *Key* parameter does not already have a message queue identifier associated with it, and the **IPC_CREAT** value is set.

Upon creation, the data structure associated with the new message queue identifier is initialized as follows:

- The `msg_perm.cuid`, `msg_perm.uid`, `msg_perm.cgid`, and `msg_perm.gid` fields are set equal to the effective user ID and effective group ID, respectively, of the calling process.
- The low–order 9 bits of the `msg_perm.mode` field are set equal to the low–order 9 bits of the *MessageFlag* parameter.
- The `msg_qnum`, `msg_lspid`, `msg_lrpid`, `msg_stime`, and `msg_rtime` fields are set equal to 0.
- The `msg_ctime` field is set equal to the current time.
- The `msg_qbytes` field is set equal to the system limit.

The **msgget** subroutine performs the following actions:

- The **msgget** subroutine either finds or creates (depending on the value of the *MessageFlag* parameter) a queue with the *Key* parameter.
- The **msgget** subroutine returns the ID of the queue header to its caller.

The following limits apply to the message queue:

- Maximum message size is 4 Mega bytes.
- Maximum number of messages per queue is 8192.
- Maximum number of message queue IDs is 4096 for AIX releases before 4.3.2 and 131072 for AIX 4.3.2 and following.

Parameters

Key	Specifies either the value **IPC_PRIVATE** or an Interprocess Communication (IPC) key constructed by the **ftok** subroutine (or by a similar algorithm).
MessageFlag	Constructed by logically ORing one or more of the following values:

IPC_CREAT	Creates the data structure if it does not already exist.
IPC_EXCL	Causes the **msgget** subroutine to fail if the **IPC_CREAT** value is also set and the data structure already exists.
S_IRUSR	Permits the process that owns the data structure to read it.
S_IWUSR	Permits the process that owns the data structure to modify it.
S_IRGRP	Permits the group associated with the data structure to read it.
S_IWGRP	Permits the group associated with the data structure to modify it.
S_IROTH	Permits others to read the data structure.
S_IWOTH	Permits others to modify the data structure.

Values that begin with **S_I** are defined in the **sys/mode.h** file and are a subset of the access permissions that apply to files.

Return Values

Upon successful completion, the **msgget** subroutine returns a message queue identifier. Otherwise, a value of –1 is returned and the **errno** global variable is set to indicate the error.

Error Codes

The **msgget** subroutine is unsuccessful if any of the following conditions is true:

EACCES	A message queue identifier exists for the *Key* parameter, but operation permission as specified by the low–order 9 bits of the *MessageFlag* parameter is not granted.
ENOENT	A message queue identifier does not exist for the *Key* parameter and the **IPC_CREAT** value is not set.
ENOSPC	A message queue identifier is to be created, but the system–imposed limit on the maximum number of allowed message queue identifiers system–wide would be exceeded.
EEXIST	A message queue identifier exists for the *Key* parameter, and both **IPC_CREAT** and **IPC_EXCL** values are set.

Implementation Specifics

This subroutine is part of Base Operating System (BOS) Runtime.

Related Information

The **ftok** subroutine, **msgctl** subroutine, **msgrcv** subroutine, **msgsnd** subroutine, **msgxrcv** subroutine.

The **mode.h** file.

msgrcv Subroutine

Purpose

Reads a message from a queue.

Library

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

Syntax

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

int msgrcv (MessageQueueID,
MessagePointer,MessageSize,MessageType, MessageFlag)
int MessageQueueID, MessageFlag;
void *MessagePointer;
size_t MessageSize;
long int MessageType;
```

Description

The **msgrcv** subroutine reads a message from the queue specified by the *MessageQueueID* parameter and stores it into the structure pointed to by the *MessagePointer* parameter. The current process must have read permission in order to perform this operation.

Note: The routine may coredump instead of returning EFAULT when an invalid pointer is passed in case of 64–bit application calling 32–bit kernel interface.

The following limits apply to the message queue:

- Maximum message size is 65,535 bytes for releases prior to AIX 4.1.5 and is 4 Megabytes for release 4.1.5 and later releases.
- Maximum number of messages per queue is 8192.
- Maximum number of message queue IDs is 4096 for AIX releases before 4.3.2 and 131072 for AIX 4.3.2 and following.
- Maximum number of bytes in a queue is 4 65,535 for releases prior to AIX 4.1.5 and is 4 Megabytes for release 4.1.5 and later releases.

  Note: For a 64–bit process, the **mtype** field is 64 bits long. However, for compatibility with 32–bit processes, the most significant 32 bits must be 0 and will not be put on the message queue. For a 64–bit receiver process, the **mtype** will again be extended to 64 bits with the most significant bits 0.

Parameters

MessageQueueID	Specifies the message queue identifier.

MessagePointer Points to a **msgbuf** structure containing the message. The **msgbuf** structure is defined in the **sys/msg.h** file and contains the following fields:

```
mtyp_t   mtype;         /* Message type */
char     mtext[1];      /* Beginning of message
text */
```

The `mtype` field contains the type of the received message as specified by the sending process. The `mtext` field is the text of the message.

MessageSize Specifies the size of the `mtext` field in bytes. The received message is truncated to the size specified by the *MessageSize* parameter if it is longer than the size specified by the *MessageSize* parameter and if the **MSG_NOERROR** value is set in the *MessageFlag* parameter. The truncated part of the message is lost and no indication of the truncation is given to the calling process.

MessageType Specifies the type of message requested as follows:

- If equal to the value of 0, the first message on the queue is received.
- If greater than 0, the first message of the type specified by the *MessageType* parameter is received.
- If less than 0, the first message of the lowest type that is less than or equal to the absolute value of the *MessageType* parameter is received.

MessageFlag Specifies either a value of 0 or is constructed by logically ORing one or more of the following values:

MSG_NOERROR
Truncates the message if it is longer than the *MessageSize* parameter.

IPC_NOWAIT Specifies the action to take if a message of the desired type is not on the queue:

- If the **IPC_NOWAIT** value is set, the calling process returns a value of –1 and sets the **errno** global variable to the **ENOMSG** error code.
- If the **IPC_NOWAIT** value is not set, the calling process suspends execution until one of the following occurs:
  - A message of the desired type is placed on the queue.
  - The message queue identifier specified by the *MessageQueueID* parameter is removed from the system. When this occurs, the **errno** global variable is set to the **EIDRM** error code, and a value of –1 is returned.
  - The calling process receives a signal that is to be caught. In this case, a message is not received and the calling process resumes in the manner described in the **sigaction** subroutine.

Return Values

Upon successful completion, the **msgrcv** subroutine returns a value equal to the number of bytes actually stored into the `mtext` field and the following actions are taken with respect to fields of the data structure associated with the *MessageQueueID* parameter:

- The `msg_qnum` field is decremented by 1.
- The `msg_lrpid` field is set equal to the process ID of the calling process.
- The `msg_rtime` field is set equal to the current time.

If the **msgrcv** subroutine is unsuccessful, a value of –1 is returned and the **errno** global variable is set to indicate the error.

Error Codes

The **msgrcv** subroutine is unsuccessful if any of the following conditions is true:

EINVAL	The *MessageQueueID* parameter is not a valid message queue identifier.
EACCES	The calling process is denied permission for the specified operation.
EINVAL	The *MessageSize* parameter is less than 0.
E2BIG	The `mtext` field is greater than the *MessageSize* parameter, and the **MSG_NOERROR** value is not set.
ENOMSG	The queue does not contain a message of the desired type and the **IPC_NOWAIT** value is set.
EFAULT	The *MessagePointer* parameter points outside of the allocated address space of the process.
EINTR	The **msgrcv** subroutine is interrupted by a signal.
EIDRM	The message queue identifier specified by the *MessageQueueID* parameter has been removed from the system.

Implementation Specifics

This subroutine is part of Base Operating System (BOS) Runtime.

Related Information

The **msgctl** subroutine, **msgget** subroutine, **msgsnd** subroutine, **msgxrcv** subroutine, **sigaction** subroutine.

msgsnd Subroutine

Purpose

Sends a message.

Library

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

Syntax

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



int msgsnd (MessageQueueID, MessagePointer,MessageSize,
MessageFlag)
int MessageQueueID, MessageFlag;
const void *MessagePointer;
size_t MessageSize;
```

Description

The **msgsnd** subroutine sends a message to the queue specified by the *MessageQueueID* parameter. The current process must have write permission to perform this operation. The *MessagePointer* parameter points to an **msgbuf** structure containing the message. The **sys/msg.h** file defines the **msgbuf** structure. The structure contains the following fields:

```
mtyp_t   mtype;      /* Message type */
char     mtext[1];   /* Beginning of message text */
```

The `mtype` field specifies a positive integer used by the receiving process for message selection. The `mtext` field can be any text of the length in bytes specified by the *MessageSize* parameter. The *MessageSize* parameter can range from 0 to the maximum limit imposed by the system.

The following example shows a typical user–defined **msgbuf** structure that includes sufficient space for the largest message: `struct my_msgbuf`

```
 mtyp_t mtype;
 char mtext[MSGSIZ]; /* MSGSIZ is the size of the largest message
*/
```

Note: The routine may coredump instead of returning EFAULT when an invalid pointer is passed in case of 64–bit application calling 32–bit kernel interface.

The following system limits apply to the message queue:

- Maximum message size is 65,535 bytes for releases prior to AIX 4.1.5 and is 4 Megabytes for release 4.1.5 and later releases.
- Maximum number of messages per queue is 8192.
- Maximum number of message queue IDs is 4096 for AIX releases before 4.3.2 and 131072 for AIX 4.3.2 and following.
- Maximum number of bytes in a queue is 4 65,535 bytes for releases prior to AIX 4.1.5 is 4 Megabytes for release 4.1.5 and later releases.

Note: For a 64–bit process, the **mtype** field is 64 bits long. However, for compatibility with 32–bit processes, the most significant 32 bits must be 0 and will not be put on the message queue. For a 64–bit receiver process, the **mtype** will again be extended to 64 bits with the most significant bits 0.

The *MessageFlag* parameter specifies the action to be taken if the message cannot be sent for one of the following reasons:

- The number of bytes already on the queue is equal to the number of bytes defined by the **msg_qbytes** structure.
- The total number of messages on the queue is equal to a system–imposed limit.

These actions are as follows:

- If the *MessageFlag* parameter is set to the **IPC_NOWAIT** value, the message is not sent, and the **msgsnd** subroutine returns a value of –1 and sets the **errno** global variable to the **EAGAIN** error code.
- If the *MessageFlag* parameter is set to 0, the calling process suspends execution until one of the following occurs:
  - The condition responsible for the suspension no longer exists, in which case the message is sent.
  - The *MessageQueueID* parameter is removed from the system. (For information on how to remove the *MessageQueueID* parameter, see the **msgctl** subroutine.) When this occurs, the **errno** global variable is set equal to the **EIDRM** error code, and a value of –1 is returned.
  - The calling process receives a signal that is to be caught. In this case the message is not sent and the calling process resumes execution in the manner prescribed in the **sigaction** subroutine.

Parameters

MessageQueueID	Specifies the queue to which the message is sent.
MessagePointer	Points to a **msgbuf** structure containing the message.
MessageSize	Specifies the length, in bytes, of the message text.
MessageFlag	Specifies the action to be taken if the message cannot be sent.

Return Values

Upon successful completion, a value of 0 is returned and the following actions are taken with respect to the data structure associated with the *MessageQueueID* parameter:

- The `msg_qnum` field is incremented by 1.
- The `msg_lspid` field is set equal to the process ID of the calling process.
- The `msg_stime` field is set equal to the current time.

If the **msgsnd** subroutine is unsuccessful, a value of –1 is returned and the **errno** global variable is set to indicate the error.

Error Codes

The **msgsnd** subroutine is unsuccessful and no message is sent if one or more of the following conditions is true:

EINVAL	The *MessageQueueID* parameter is not a valid message queue identifier.
EACCES	The calling process is denied permission for the specified operation.
EINVAL	The `mtype` field is less than 1.

EAGAIN The message cannot be sent for one of the reasons stated previously, and the *MessageFlag* parameter is set to the **IPC_NOWAIT** value.

EINVAL The *MessageSize* parameter is less than 0 or greater than the system–imposed limit.

EFAULT The *MessagePointer* parameter points outside of the address space of the process.

EINTR The **msgsnd** subroutine received a signal.

EIDRM The message queue identifier specified by the *MessageQueueID* parameter has been removed from the system.

ENOMEM The message could not be sent because not enough storage space was available.

EINVAL The upper 32–bits of the 64–bit mtype field for a 64–bit process is not 0.

Implementation Specifics

This subroutine is part of Base Operating System (BOS) Runtime.

Related Information

The **msgctl** subroutine, **msgget** subroutine, **msgrcv** subroutine, **msgxrcv** subroutine, **sigaction** subroutine.

msgxrcv Subroutine

Purpose

Receives an extended message.

Library

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

Syntax

For releases prior to AIX 4.3.0:

```
#include <sys/msg.h>int msgxrcv (MessageQueueID, MessagePointer,
MessageSize, MessageType, MessageFlag) int MessageQueueID,
MessageFlag, MessageSize; struct msgxbuf * MessagePointer; long
MessageType;
```

For AIX 4.3.0 and later releases:

```
#include <sys/msg.h>int msgxrcv
(MessageQueueID, MessagePointer, MessageSize, MessageType,
MessageFlag) int MessageQueueID, MessageFlag; size_t MessageSize;
struct msgxbuf * MessagePointer; long MessageType;
```

Description

The **msgxrcv** subroutine reads a message from the queue specified by the *MessageQueueID* parameter and stores it into the extended message receive buffer pointed to by the *MessagePointer* parameter. The current process must have read permission in order to perform this operation. The **msgxbuf** structure is defined in the **sys**/**msg.h** file.

> **Note**: The routine may coredump instead of returning EFAULT when an invalid pointer is passed in case of 64–bit application calling 32–bit kernel interface.

The following limits apply to the message queue:

- Maximum message size is 65,535 bytes for releases prior to AIX 4.1.5 and is 4 Megabytes for release 4.1.5 and later releases.
- Maximum number of messages per queue is 8192.
- Maximum number of message queue IDs is 4096 for AIX releases before 4.3.2 and 131072 for AIX 4.3.2 and following.
- Maximum number of bytes in a queue is 4 65,535 for releases prior to AIX 4.1.5 and is 4 Megabytes for release 4.1.5 and later releases.

> **Note**: For a 64–bit process, the **mtype** field is 64 bits long. However, for compatibility with 32–bit processes, the most significant 32 bits must be 0 and will not be put on the message queue. For a 64–bit receiver process, the **mtype** will again be extended to 64 bits with the most significant bits 0.

Parameters

MessageQueueID Specifies the message queue identifier.

MessagePointer Specifies a pointer to an extended message receive buffer where a message is stored.

MessageSize Specifies the size of the `mtext` field in bytes. The receive message is truncated to the size specified by the *MessageSize* parameter if it is larger than the *MessageSize* parameter and the **MSG_NOERROR** value is true. The truncated part of the message is lost and no indication of the truncation is given to the calling process. If the message is longer than the number of bytes specified by the *MessageSize* parameter and the **MSG_NOERROR** value is not set, the **msgxrcv** subroutine is unsuccessful and sets the **errno** global variable to the **E2BIG** error code.

MessageType Specifies the type of message requested as follows:

- If the *MessageType* parameter is equal to 0, the first message on the queue is received.
- If the *MessageType* parameter is greater than 0, the first message of the type specified by the *MessageType* parameter is received.
- If the *MessageType* parameter is less than 0, the first message of the lowest type that is less than or equal to the absolute value of the *MessageType* parameter is received.

MessageFlag Specifies a value of 0 or a value constructed by logically ORing one or more of the following values:

MSG_NOERROR
Truncates the message if it is longer than the number of bytes specified by the *MessageSize* parameter.

IPC_NOWAIT Specifies the action to take if a message of the desired type is not on the queue:

- If the **IPC_NOWAIT** value is set, the calling process returns a value of –1 and sets the **errno** global variable to the **ENOMSG** error code.
- If the **IPC_NOWAIT** value is not set, the calling process suspends execution until one of the following occurs:
  - A message of the desired type is placed on the queue.
  - The message queue identifier specified by the *MessageQueueID* parameter is removed from the system. When this occurs, the **errno** global variable is set to the **EIDRM** error code, and a value of –1 is returned.
  - The calling process receives a signal that is to be caught. In this case, a message is not received and the calling process resumes in the manner prescribed in the **sigaction** subroutine.

Return Values

Upon successful completion, the **msgxrcv** subroutine returns a value equal to the number of bytes actually stored into the `mtext` field, and the following actions are taken with respect to the data structure associated with the *MessageQueueID* parameter:

- The `msg_qnum` field is decremented by 1.

- The `msg_lrpid` field is set equal to the process ID of the calling process.
- The `msg_rtime` field is set equal to the current time.

If the **msgxrcv** subroutine is unsuccessful, a value of –1 is returned and the **errno** global variable is set to indicate the error.

Error Codes

The **msgxrcv** subroutine is unsuccessful if any of the following conditions is true:

EINVAL	The *MessageQueueID* parameter is not a valid message queue identifier.
EACCES	The calling process is denied permission for the specified operation.
EINVAL	The *MessageSize* parameter is less than 0.
E2BIG	The `mtext` field is greater than the *MessageSize* parameter, and the **MSG_NOERROR** value is not set.
ENOMSG	The queue does not contain a message of the desired type and the **IPC_NOWAIT** value is set.
EFAULT	The *MessagePointer* parameter points outside of the process address space.
EINTR	The **msgxrcv** subroutine was interrupted by a signal.
EIDRM	The message queue identifier specified by the *MessageQueueID* parameter is removed from the system.

Implementation Specifics

This subroutine is part of Base Operating System (BOS) Runtime.

Related Information

The **msgctl** subroutine, **msgget** subroutine, **msgrcv** subroutine, **msgsnd** subroutine, **sigaction** subroutine.

msleep Subroutine

Purpose

Puts a process to sleep when a semaphore is busy.

Library

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

Syntax

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

int msleep (Sem)
msemaphore *Sem;
```

Description

The **msleep** subroutine puts a calling process to sleep when a semaphore is busy. The semaphore should be located in a shared memory region. Use the **mmap** subroutine to create the shared memory section.

All of the values in the **msemaphore** structure must result from a **msem_init** subroutine call. This call may or may not be followed by a sequence of calls to the **msem_lock** subroutine or the **msem_unlock** subroutine. If the **msemaphore** structure value originates in another manner, the results of the **msleep** subroutine are undefined.

The address of the **msemaphore** structure is significant. You should be careful not to modify the structure's address. If the structure contains values copied from a **msemaphore** structure at another address, the results of the **msleep** subroutine are undefined.

Parameters

Sem	Points to the **msemaphore** structure that specifies the semaphore.

Error Codes

If the **msleep** subroutine is unsuccessful, the **errno** global variable is set to one of the following values:

EFAULT	Indicates that the *Sem* parameter points to an invalid address or the address does not contain a valid **msemaphore** structure.
EINTR	Indicates that the process calling the **msleep** subroutine was interrupted by a signal while sleeping.

Implementation Specifics

The **msleep** subroutine is part of the Base Operating System (BOS) Runtime calls.

Related Information

The **mmap** subroutine, **msem_init** subroutine, **msem_lock** subroutine, **msem_unlock** subroutine, **mwakeup** subroutine.

Understanding Memory Mapping in *AIX General Programming Concepts : Writing and Debugging Programs*.

msync Subroutine

Purpose

Synchronizes a mapped file.

Library

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

Syntax

```
#include <sys/types.h>
#include <sys/mman.h>

int msync (addr, len, flags)
void *addr;
size_t len;
int flags;
```

Description

The **msync** subroutine controls the caching operations of a mapped file region. Use the **msync** subroutine to transfer modified pages in the region to the underlying file storage device.

If the application has requested X/Open UNIX95 Specification compliant behavior then the **st_ctime** and **st_mtime** fields of the mapped file are marked for update upon successful completion of the **msync** subroutine call if the file has been modified.

Parameters

addr — Specifies the address of the region to be synchronized. Must be a multiple of the page size returned by the **sysconf** subroutine using the **_SC_PAGE_SIZE** value for the *Name* parameter.

len — Specifies the length, in bytes, of the region to be synchronized. If the *len* parameter is not a multiple of the page size returned by the **sysconf** subroutine using the **_SC_PAGE_SIZE** value for the *Name* parameter, the length of the region is rounded up to the next multiple of the page size.

flags — Specifies one or more of the following symbolic constants that determine the way caching operations are performed:

MS_SYNC — Specifies synchronous cache flush. The **msync** subroutine does not return until the system completes all I/O operations.

This flag is invalid when the **MAP_PRIVATE** flag is used with the **mmap** subroutine. **MAP_PRIVATE** is the default privacy setting. When the **MS_SYNC** and **MAP_PRIVATE** flags both are used, the **msync** subroutine returns an **errno** value of **EINVAL**.

MS_ASYNC — Specifies an asynchronous cache flush. The **msync** subroutine returns after the system schedules all I/O operations.

This flag is invalid when the **MAP_PRIVATE** flag is used with the **mmap** subroutine. **MAP_PRIVATE** is the default privacy setting. When the **MS_SYNC** and **MAP_PRIVATE** flags both are used, the **msync** subroutine returns an **errno** value of **EINVAL**.

MS_INVALIDATE — Specifies that the **msync** subroutine invalidates all cached copies of the pages. New copies of the pages must then be obtained from the file system the next time they are referenced.

Return Values

When successful, the **msync** subroutine returns 0. Otherwise, it returns –1 and sets the **errno** global variable to indicate the error.

Error Codes

If the **msync** subroutine is unsuccessful, the **errno** global variable is set to one of the following values:

EIO — An I/O error occurred while reading from or writing to the file system.

ENOMEM — The range specified by (*addr*, *addr* + *len*) is invalid for a process' address space, or the range specifies one or more unmapped pages.

EINVAL — The *addr* argument is not a multiple of the page size as returned by the **sysconf** subroutine using the **_SC_PAGE_SIZE** value for the *Name* parameter, or the *flags* parameter is invalid. The address of the region is within the process' inheritable address space.

munmap Subroutine

Purpose

Unmaps a mapped region.

Library

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

Syntax

```
#include <sys/types.h>
#include <sys/mman.h>

int munmap (addr, len)
void *addr;
size_t len;
```

Description

The **munmap** subroutine unmaps a mapped file region or anonymous memory region. The **munmap** subroutine unmaps regions created from calls to the **mmap** subroutine only.

If an address lies in a region that is unmapped by the **munmap** subroutine and that region is not subsequently mapped again, any reference to that address will result in the delivery of a **SIGSEGV** signal to the process.

Parameters

addr	Specifies the address of the region to be unmapped. Must be a multiple of the page size returned by the **sysconf** subroutine using the **_SC_PAGE_SIZE** value for the *Name* parameter.
len	Specifies the length, in bytes, of the region to be unmapped. If the *len* parameter is not a multiple of the page size returned by the **sysconf** subroutine using the **_SC_PAGE_SIZE** value for the *Name* parameter, the length of the region is rounded up to the next multiple of the page size.

Return Values

When successful, the **munmap** subroutine returns 0. Otherwise, it returns –1 and sets the **errno** global variable to indicate the error.

Error Codes

If the **munmap** subroutine is unsuccessful, the **errno** global variable is set to the following value:

EINVAL	The *addr* parameter is not a multiple of the page size as returned by the **sysconf** subroutine using the **_SC_PAGE_SIZE** value for the *Name* parameter.
EINVAL	The application has requested X/Open UNIX95 Specification compliant behavior and the *len* arguement is 0.

mwakeup Subroutine

Purpose

Wakes up a process that is waiting on a semaphore.

Library

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

Syntax

```
#include <sys/mman.h>
int mwakeup (Sem)
msemaphore *Sem;
```

Description

The **mwakeup** subroutine wakes up a process that is sleeping and waiting for an idle semaphore. The semaphore should be located in a shared memory region. Use the **mmap** subroutine to create the shared memory section.

All of the values in the **msemaphore** structure must result from a **msem_init** subroutine call. This call may or may not be followed by a sequence of calls to the **msem_lock** subroutine or the **msem_unlock** subroutine. If the **msemaphore** structure value originates in another manner, the results of the **mwakeup** subroutine are undefined.

The address of the **msemaphore** structure is significant. You should be careful not to modify the structure's address. If the structure contains values copied from a **msemaphore** structure at another address, the results of the **mwakeup** subroutine are undefined.

Parameters

Sem	Points to the **msemaphore** structure that specifies the semaphore.

Return Values

When successful, the **mwakeup** subroutine returns a value of 0. Otherwise, this routine returns a value of –1 and sets the **errno** global variable to **EFAULT**.

Error Codes

A value of **EFAULT** indicates that the *Sem* parameter points to an invalid address or that the address does not contain a valid **msemaphore** structure.

Implementation Specifics

The **mwakeup** subroutine is part of the Base Operating System (BOS) runtime calls.

Related Information

The **mmap** subroutine, **msem_init** subroutine, **msem_lock** subroutine, **msem_unlock** subroutine, and the **msleep** subroutine.

Understanding Memory Mapping in *AIX General Programming Concepts : Writing and Debugging Programs*.

newpass Subroutine

Purpose

Generates a new password for a user.

Library

Security Library (**libc.a**)

Syntax

```
#include <usersec.h>
#include <userpw.h>

char *newpass(Password)
struct userpw *Password;
```

Description

The **newpass** subroutine generates a new password for the user specified by the *Password* parameter. The new password is then checked to ensure that it meets the password rules on the system unless the user is exempted from these restrictions. Users must have root user authority to invoke this subroutine. The password rules are defined in the **/etc/security/user** file and are described in both the **user** file and the **passwd** command.

Passwords can contain almost any legal value for a character but cannot contain (National Language Support (NLS) code points. Passwords cannot have more than the value specified by **MAX_PASS**.

The **newpass** subroutine authenticates the user prior to changing the password. If the **PW_ADMCHG** flag is set in the `upw_flags` member of the *Password* parameter, the supplied password is checked against the password to determine the user corresponding to the real user ID of the process instead of the user specified by the `upw_name` member of the *Password* parameter structure.

If a password is successfully generated, a pointer to a buffer containing the new password is returned and the last update time is reset.

Note: The **newpass** subroutine is not safe in a multi–threaded environment. To use **newpass** in a threaded application, the application must keep the integrity of each thread.

Parameters

Password Specifies a user password structure. This structure is defined in the **userpw.h** file and contains the following members:

`upw_name`	A pointer to a character buffer containing the user name.
`upw_passwd`	A pointer to a character buffer containing the current password.
`upw_lastupdate`	The time the password was last changed, in seconds since the epoch.
`upw_flags`	A bit mask containing 0 or more of the following values:

PW_NOCHECK	This bit indicates that new passwords need not meet the composition criteria for passwords on the system.
PW_ADMIN	This bit indicates that password information for this user may only be changed by the root user.
PW_ADMCHG	This bit indicates that the password is being changed by an administrator and the password will have to be changed upon the next successful running of the **login** or **su** commands to this account.

Security

Policy: Authentication To change a password, the invoker must be properly authenticated.

Note: Programs that invoke the **newpass** subroutine should be written to conform to the authentication rules enforced by **newpass**. The **PW_ADMCHG** flag should always be explicitly cleared unless the invoker of the command is an administrator.

Return Values

If a new password is successfully generated, a pointer to the new encrypted password is returned. If an error occurs, a null pointer is returned and the **errno** global variable is set to indicate the error.

Error Codes

The **newpass** subroutine fails if one or more of the following are true;

EINVAL	The structure passed to the **newpass** subroutine is invalid.
ESAD	Security authentication is denied for the invoker.
EPERM	The user is unable to change the password of a user with the **PW_ADMCHG** bit set, and the real user ID of the process is not the root user.
ENOENT	The user is not properly defined in the database.

Implementation Specifics

This subroutine is part of Base Operating System (BOS) Runtime.

Related Information

The **getpass** subroutine, **getuserpw** subroutine.

The **login** command, **passwd** command, **pwdadm** command.

List of Security and Auditing Subroutines, Subroutines Overview in *AIX General Programming Concepts : Writing and Debugging Programs*.

nftw or nftw64 Subroutine

Purpose

Walks a file tree.

Library

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

Syntax

```
#include <ftw.h>

int nftw (Path, Function, Depth, Flags)
const char *Path;
int *(*Function) ( );
int Depth;
int Flags;

int nftw64(Path,Function,Depth)
const char *Path;
int *(*Function) ( );
int Depth;
int Flags;
```

Description

The **nftw** and **nftw64** subroutines recursively descend the directory hierarchy rooted in the *Path* parameter. The nftw and nftw64 subroutines have a similar effect to ftw and ftw64 except that they take an additional argument flags, which is a bitwise inclusive–OR of zero or more of the following flags:

FTW_CHDIR	If set, the current working directory will change to each directory as files are reported. If clear, the current working directory will not change.
FTW_DEPTH	If set, all files in a directory will be reported before the directory itself. If clear, the directory will be reported before any files.
FTW_MOUNT	If set, symbolic links will not be followed. If clear the links will be followed.
FTW_PHYS	If set, symbolic links will not be followed. If clear the links will be followed, and will not report the same file more than once.

For each file in the hierarchy, the **nftw** and **nftw64** subroutines call the function specified by the *Function* parameter. The nftw subroutine passes a pointer to a null–terminated character string containing the name of the file, a pointer to a stat structure containing information about the file, an integer and a pointer to an FTW structure. The nftw64 subroutine passes a pointer to a null–terminated character string containing the name of the file, a pointer to a stat64 structure containing information about the file, an integer and a pointer to an FTW structure.

The nftw subroutine uses the stat system call which will fail on files of size larger than 2 Gigabytes. The nftw64 subroutine must be used if there is a possibility of files of size larger than 2 Gigabytes.

The integer passed to the *Function* parameter identifies the file type with one of the following values:

FTW_F	Regular file
FTW_D	Directory
FTW_DNR	Directory that cannot be read

FTW_DP	The *Object* is a directory and subdirectories have been visited. (This condition will only occur if **FTW_DEPTH** is included in flags).
FTW_SL	Symbolic Link
FTW_SLN	Symbolic Link that does not name an existin file. (This condition will only occur if the **FTW_PHYS** flag is not included in flags).
FTW_NS	File for which the **stat** structure could not be executed successfully

If the integer is **FTW_DNR**, the files and subdirectories contained in that directory are not processed.

If the integer is **FTW_NS**, the **stat** structure contents are meaningless. An example of a file that causes **FTW_NS** to be passed to the *Function* parameter is a file in a directory for which you have read permission but not execute (search) permission.

The **FTW** structure pointer passed to the *Function* parameter contains base which is the offset of the object's filename in the pathname passed as the first argument to *Function*. The value of level indicates depth relative to the root of the walk.

The **nftw** and **nftw64** subroutines use one file descriptor for each level in the tree. The *Depth* parameter specifies the maximum number of file descriptors to be used. In general, the **nftw** and **nftw64** run faster of the value of the *Depth* parameter is at least as large as the number of levels in the tree. However, the value of the *Depth* parameter must not be greater than the number of file descriptors currently available for use. If the value of the *Depth* parameter is 0 or a negative number, the effect is the same as if it were 1.

Because the **nftw** and **nftw64** subroutines are recursive, it is possible for it to terminate with a memory fault due to stack overflow when applied to very deep file structures.

The **nftw** and **nftw64** subroutines use the **malloc** subroutine to allocate dynamic storage during its operation. If the **nftw** subroutine is terminated prior to its completion, such as by the **longjmp** subroutine being executed by the function specified by the *Function* parameter or by an interrupt routine, the **nftw** subroutine cannot free that storage. The storage remains allocated. A safe way to handle interrupts is to store the fact that an interrupt has occurred, and arrange to have the function specified by the *Function* parameter return a nonzero value the next time it is called.

Parameters

Path	Specifies the directory hierarchy to be searched.
Function	User supplied function that is called for each file encountered.
Depth	Specifies the maximum number of file descriptors to be used. *Depth* cannot be greater than OPEN_MAX which is described in the sys/limits.h header file.

Return Values

If the tree is exhausted, the **nftw** and **nftw64** subroutine returns a value of 0. If the subroutine pointed to by **fn** returns a nonzero value, **nftw** and **nftw64** stops its tree traversal and returns whatever value was returned by the subroutine pointed to by **fn**. If the **nftw** and **nftw64** subroutine detects an error, it returns a –1 and sets the **errno** global variable to indicate the error.

Error Codes

If the **nftw** or **nftw64** subroutines detect an error, a value of –1 is returned and the **errno** global variable is set to indicate the error.

The **nftw** and **nftw64** subroutine are unsuccessful if:

EACCES	Search permission is denied for any component of the *Path* parameter or read permission is denied for *Path*.
ENAMETOOLONG	The length of the path exceeds **PATH_MAX** while **_POSIX_NO_TRUNC** is in effect.
ENOENT	The *Path* parameter points to the name of a file that does not exist or points to an empty string.
ENOTDIR	A component of the *Path* parameter is not a directory.

The **nftw** subroutine is unsuccessful if:

EOVERFLOW	A file in *Path* is of a size larger than 2 Gigabytes.

Implementation Specifics

This subroutines is part of Base Operating System (BOS) Runtime.

Related Information

The **stat** or **malloc** subroutine.

The **ftw** subroutine.

nl_langinfo Subroutine

Purpose

Returns information on the language or cultural area in a program's locale.

Library

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

Syntax

```
#include <nl_types.h>
#include <langinfo.h>

char *nl_langinfo (Item)
nl_item Item;
```

Description

The **nl_langinfo** subroutine returns a pointer to a string containing information relevant to the particular language or cultural area defined in the program's locale and corresponding to the *Item* parameter. The active language or cultural area is determined by the default value of the environment variables or by the most recent call to the **setlocale** subroutine. If the **setlocale** subroutine has not been called in the program, then the default C locale values will be returned from **nl_langinfo**.

Values for the *Item* parameter are defined in the **langinfo.h** file.

The following table summarizes the categories for which **nl_langinfo()** returns information, the values the *Item* parameter can take, and descriptions of the returned strings. In the table, radix character refers to the character that separates whole and fractional numeric or monetary quantities. For example, a period (.) is used as the radix character in the U.S., and a comma (,) is used as the radix character in France.

Category	**Value of *item***	**Returned Result**
LC_MONETARY	CRNCYSTR	Currency symbol and its position.
LC_NUMERIC	RADIXCHAR	Radix character.
LC_NUMERIC	THOUSEP	Separator for the thousands.
LC_MESSAGES	YESSTR	Affirmative response for yes/no queries.
LC_MESSAGES	NOSTR	Negative response for yes/no queries.
LC_TIME	D_T_FMT	String for formatting date and time.
LC_TIME	D_FMT	String for formatting date.
LC_TIME	T_FMT	String for formatting time.
LC_TIME	AM_STR	Antemeridian affix.
LC_TIME	PM_STR	Postmeridian affix.
LC_TIME	DAY_1 through DAY_7	Name of the first day of the week to the seventh day of the week.
LC_TIME	ABDAY_1 through ABDAY–7	Abbreviated name of the first day of the week to the seventh day of the week.
LC_TIME	MON_1 through MON_12	Name of the first month of the year to the twelfth month of the year.
LC_TIME	ABMON_1 through ABMON_12	Abbreviated name of the first month of the year to the twelfth month.
LC_CTYPE	CODESET	Code set currently in use in the program.

Note: The information returned by the **nl_langinfo** subroutine is located in a static buffer. The contents of this buffer are overwritten in subsequent calls to the **nl_langinfo** subroutine. Therefore, you should save the returned information.

Parameter

Item	Information needed from locale.

Return Values

In a locale where language information data is not defined, the **nl_langinfo** subroutine returns a pointer to the corresponding string in the C locale. In all locales, the **nl_langinfo** subroutine returns a pointer to an empty string if the *Item* parameter contains an invalid setting.

The **nl_langinfo** subroutine returns a pointer to a static area. Subsequent calls to the **nl_langinfo** subroutine overwrite the results of a previous call.

Implementation Specifics

This subroutine is part of Base Operating System (BOS) Runtime.

Related Information

The **localeconv** subroutine, **rpmatch** subroutine, **setlocale** subroutine.

Subroutines Overview, National Language Support Overview for Programming, and Understanding Locale Subroutines in *AIX General Programming Concepts : Writing and Debugging Programs*.

nlist64 Subroutine

Purpose

Gets entries from a name list.

Library

Standard C Library [**libc.a**]

Syntax

```
#include <nlist.h>

int nlist64(FileName, N1)
const char *FileName;
struct nlist64 *N1;
```

Description

The **nlist64** subroutine allows a program to examine the name list in the executable file named by the *FileName* parameter. It selectively extracts a list of values and places them in the array of **nlist64** structures pointed to by the *N1* parameter.

The name list specified by the *N1* parameter consists of an array of structures containing names of variables, types, and values. The list is terminated with an element that has a null string in the name structure member. Each variable name is looked up in the name list of the file. If the name is found, the type and value of the name are inserted in the next two fields. The type field is set to 0 unless the file was compiled with the **–g** option. If the name is not found, both the type and value entries are set to 0.

All entries are set to 0 if the specified file cannot be read or if it does not contain a valid name list.

The **nlist64** subroutine runs in both 32–bit and 64–bit mode. The **nlist64** subroutine runs in both 32–bit and 64–bit mode. **nlist64** can read both 32–bit XCOFF and 64–bit XCOFF files in both 32–bit and 64–bit modes.

In 32–bit mode, the **_n_name** pointer variable in the **nlist64** structure is 4 bytes wide. In the 64–bit mode, it is 8 bytes wide. In both 32–bit mode and 64–bit mode, the **n_value** variable (long long) is 8 bytes wide.

You can use the **nlist64** subroutine to examine the system name list kept in the /**unix** file. By examining this list, you can ensure that your programs obtain current system addresses.

The **nlist.h** file is automatically included by **a.out.h** for compatibility. However, do not include the **a.out.h** file if you only need the information necessary to use the **nlist64** subroutine. If you do include **a.out.h**, follow the **#include** statement with the line:

```
#undef n_name
```

Notes:

1. If both the **nlist.h** and **netdb.h** files are to be included, the **netdb.h** file should be included before the **nlist.h** file in order to avoid a conflict with the **n_name** structure member. Likewise, if both the **a.out.h** and **netdb.h** files are to be included, the **netdb.h** file should be included before the **a.out.h** file to avoid a conflict with the **n_name** structure.
2. If the **netdb.h** file and either the **nlist.h** or **syms.h** file are included, `n_name` will be defined as **_n._n_name**. This definition allows you to access the `n_name` field in the **nlist64** or **syment** structure. If you need to access the `n_name` filed in the **netent** structure, undefine `n_name` by including:

   ```
   #undef n_name
   ```

in your code before accessing the `n_name` field in the **netent** structure. If you need to access the `n_name` field in a **syment** or **nlist64** structure after undefining `n_name`, redefine `n_name` with:

```
#define n_name _n._n_name
```

3. There is no **nlist64** subroutine in **libbsd.a**

Parameters

FileName	Specifies the name of the file containing a name list.
N1	Points to the array of **nlist64** structures.

Return Values

Upon successful completion, a 0 is returned.

If the file cannot be found or if it is not a valid name list, a value of –1 is returned.

To obtain the BSD–compatible version of the subroutine, compile with the **libbsd.a** library.

Implementation Specifics

This subroutine is part of Base Operating System (BOS) Runtime.

Related Information

The **knlist** subroutine.

The **a.out** file.

Subroutines Overview in *AIX General Programming Concepts : Writing and Debugging Programs*.

nlist Subroutine

Purpose

Gets entries from a name list.

Library

Standard C Library [**libc.a**]

Berkeley Compatibility Library [**libbsd.a**]

Syntax

```
#include <nlist.h>

int nlist(FileName, N1)
const char *FileName;
struct nlist *N1;
```

Description

The **nlist** subroutine allows a program to examine the name list in the executable file named by the *FileName* parameter. It selectively extracts a list of values and places them in the array of **nlist** structures pointed to by the *N1* parameter.

The name list specified by the *N1* parameter consists of an array of structures containing names of variables, types, and values. The list is terminated with an element that has a null string in the name structure member. Each variable name is looked up in the name list of the file. If the name is found, the type and value of the name are inserted in the next two fields. The type field is set to 0 unless the file was compiled with the **–g** option. If the name is not found, both the type and value entries are set to 0.

All entries are set to 0 if the specified file cannot be read or if it does not contain a valid name list.

The **nlist** subroutine runs in both 32–bit and 64–bit mode. In 32–bit mode, **nlist** can read only 32–bit XCOFF format files and will give a **–1** return code on a 64–bit XCOFF file. In 64–bit mode, **nlist** can read both 32–bit XCOFF format files and 64–bit XCOFF files.

In 32–bit mode, the **_n_name** pointer and the **n_value** variable in the **nlist** structure are 4 bytes wide, while in the 64–bit mode, they are both 8 bytes wide.

The **nlist** subroutine in **libbsd.a** is only supported in 32–bit mode.

You can use the **nlist** subroutine to examine the system name list kept in the /**unix** file. By examining this list, you can ensure that your programs obtain current system addresses.

The **nlist.h** file is automatically included by **a.out.h** for compatibility. However, do not include the **a.out.h** file if you only need the information necessary to use the **nlist** subroutine. If you do include **a.out.h**, follow the **#include** statement with the line:

```
#undef n_name
```

Notes:

1. If both the **nlist.h** and **netdb.h** files are to be included, the **netdb.h** file should be included before the **nlist.h** file in order to avoid a conflict with the **n_name** structure member. Likewise, if both the **a.out.h** and **netdb.h** files are to be included, the **netdb.h** file should be included before the **a.out.h** file to avoid a conflict with the **n_name** structure.
2. If the **netdb.h** file and either the **nlist.h** or **syms.h** file are included, `n_name` will be defined as **_n._n_name**. This definition allows you to access the `n_name` field in the **nlist** or **syment** structure. If you need to access the `n_name` field in the **netent** structure, undefine `n_name` by including:

```
#undef n_name
```

in your code before accessing the `n_name` field in the **netent** structure. If you need to access the `n_name` field in a **syment** or **nlist** structure after undefining `n_name`, redefine `n_name` with:

```
#define n_name _n._n_name
```

Parameters

FileName	Specifies the name of the file containing a name list.
N1	Points to the array of **nlist** structures.

Return Values

Upon successful completion, a **0** is returned. In BSD, the number of unfound namelist entries is returned. If the file cannot be found or if it is not a valid name list, a value of **–1** is returned.

Compatibility Interfaces

To obtain the BSD–compatible version of the subroutine, compile with the **libbsd.a** library.

Implementation Specifics

This subroutine is part of Base Operating System (BOS) Runtime.

Related Information

The **knlist**, **nlist64** subroutine.

The **a.out** file.

Subroutines Overview in *AIX General Programming Concepts : Writing and Debugging Programs*.

ns_addr Subroutine

Purpose

XNS address conversion routines.

Library

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

Syntax

#include <sys/types.h>
#include <netns/ns.h>

struct ns_addr(char ****cp*)**

Description

The **ns_addr** subroutine interprets character strings representing XNS addresses, returning binary information suitable for use in system calls.

The **ns_addr** subroutine separates an address into one to three fields using a single delimiter and examines each field for byte separators (colon or period). The delimiters are:

.	period
:	colon
#	pound sign.

If byte separators are found, each subfield separated is taken to be a small hexadecimal number, and the entirety is taken as a network–byte–ordered quantity to be zero extended in the high–networked–order bytes. Next, the field is inspected for hyphens, which would indicate the field is a number in decimal notation with hyphens separating the millenia. The field is assumed to be a number, interpreted as hexadecimal, if a leading *0x* (as in C), a trailing *H*, (as in Mesa), or any super–octal digits are present. The field is interpreted as octal if a leading *0* is present and there are no super–octal digits. Otherwise, the field is converted as a decimal number.

Parameter

cp	Returns a pointer to the address of a **ns_addr** structure.

Implementation Specifics

The **ns_addr** subroutine is part of Base Operating System (BOS) Runtime.

ns_ntoa Subroutine

Purpose

XNS address conversion routines.

Library

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

Syntax

```
#include <sys/types.h>
#include <netns/ns.h>

char *ns_ntoa (
struct ns_addr ns)
```

Description

The **ns_ntoa** subroutine takes XNS addresses and returns ASCII strings representing the address in a notation in common use in the Xerox Development Environment:

```
<network number> <host number> <port number>
```

Trailing zero fields are suppressed, and each number is printed in hexadecimal, in a format suitable for input to the **ns_addr** subroutine. Any fields lacking super–decimal digits will have a trailing *H* appended.

Note: The string returned by **ns_ntoa** resides in static memory.

Parameter

ns	Returns a pointer to a string.

Implementation Specifics

The **ns_ntoa** subroutine is part of Base Operating System (BOS) Runtime.

odm_add_obj Subroutine

Purpose

Adds a new object into an object class.

Library

Object Data Manager Library (**libodm.a**)

Syntax

```
#include <odmi.h>

int odm_add_obj (ClassSymbol, DataStructure)
CLASS_SYMBOL ClassSymbol;
struct ClassName *DataStructure;
```

Description

The **odm_add_obj** subroutine takes as input the class symbol that identifies both the object class to add and a pointer to the data structure containing the object to be added.

The **odm_add_obj** subroutine opens and closes the object class around the subroutine if the object class was not previously opened. If the object class was previously opened, the subroutine leaves the object class open when it returns.

Parameters

ClassSymbol	Specifies a class symbol identifier returned from an **odm_open_class** subroutine. If the **odm_open_class** subroutine has not been called, then this identifier is the *ClassName*_**CLASS** structure that was created by the **odmcreate** command.
DataStructure	Specifies a pointer to an instance of the C language structure corresponding to the object class referenced by the *ClassSymbol* parameter. The structure is declared in the **.h** file created by the **odmcreate** command and has the same name as the object class.

Return Values

Upon successful completion, an identifier for the object that was added is returned. If the **odm_add_obj** subroutine is unsuccessful, a value of –1 is returned and the **odmerrno** variable is set to an error code.

Error Codes

Failure of the **odm_add_obj** subroutine sets the **odmerrno** variable to one of the following error codes:

ODMI_CLASS_DNE
The specified object class does not exist. Check path name and permissions.

ODMI_CLASS_PERMS
The object class cannot be opened because of the file permissions.

ODMI_INVALID_CLXN
Either the specified collection is not a valid object class collection or the collection does not contain consistent data.

ODMI_INVALID_PATH

The specified path does not exist on the file system. Make sure the path is accessible.

ODMI_MAGICNO_ERR
The class symbol does not identify a valid object class.

ODMI_OPEN_ERR
Cannot open the object class. Check path name and permissions.

ODMI_PARAMS
The parameters passed to the subroutine were not correct. Make sure there are the correct number of parameters and that they are valid.

ODMI_READ_ONLY
The specified object class is opened as read–only and cannot be modified.

ODMI_TOOMANYCLASSES
Too many object classes have been accessed. An application can only access less than 1024 object classes.

Implementation Specifics

This subroutine is part of Base Operating System (BOS) Runtime.

Related Information

The **odm_create_class** subroutine, **odm_open_class** subroutine, **odm_rm_obj** subroutine.

The **odmcreate** command.

See ODM Example Code and Output in *AIX General Programming Concepts : Writing and Debugging Programs* for an example of a **.h** file.

Object Data Manager (ODM) Overview for Programmers in *AIX General Programming Concepts : Writing and Debugging Programs*.

odm_change_obj Subroutine

Purpose

Changes an object in the object class.

Library

Object Data Manager Library (**libodm.a**)

Syntax

```
#include <odmi.h>

int odm_change_obj (ClassSymbol, DataStructure)
CLASS_SYMBOL ClassSymbol;
struct ClassName *DataStructure;
```

Description

The **odm_change_obj** subroutine takes as input the class symbol that identifies both the object class to change and a pointer to the data structure containing the object to be changed. The application program must first retrieve the object with an **odm_get_obj** subroutine call, change the data values in the returned structure, and then pass that structure to the **odm_change_obj** subroutine.

The **odm_change_obj** subroutine opens and closes the object class around the change if the object class was not previously opened. If the object class was previously opened, then the subroutine leaves the object class open when it returns.

Parameters

ClassSymbol
: Specifies a class symbol identifier returned from an **odm_open_class** subroutine. If the **odm_open_class** subroutine has not been called, then this identifier is the *ClassName*_**CLASS** structure that is created by the **odmcreate** command.

DataStructure
: Specifies a pointer to an instance of the C language structure corresponding to the object class referenced by the *ClassSymbol* parameter. The structure is declared in the **.h** file created by the **odmcreate** command and has the same name as the object class.

Return Values

Upon successful completion, a value of 0 is returned. If the **odm_change_obj** subroutine fails, a value of –1 is returned and the **odmerrno** variable is set to an error code.

Error Codes

Failure of the **odm_change_obj** subroutine sets the **odmerrno** variable to one of the following error codes:

ODMI_CLASS_DNE
: The specified object class does not exist. Check path name and permissions.

ODMI_CLASS_PERMS
: The object class cannot be opened because of the file permissions.

ODMI_INVALID_CLXN
: Either the specified collection is not a valid object class collection or the collection does not contain consistent data.

ODMI_INVALID_PATH

The specified path does not exist on the file system. Make sure the path is accessible.

ODMI_MAGICNO_ERR

The class symbol does not identify a valid object class.

ODMI_NO_OBJECT

The specified object identifier did not refer to a valid object.

ODMI_OPEN_ERR

Cannot open the object class. Check path name and permissions.

ODMI_PARAMS

The parameters passed to the subroutine were not correct. Make sure there are the correct number of parameters and that they are valid.

ODMI_READ_ONLY

The specified object class is opened as read–only and cannot be modified.

ODMI_TOOMANYCLASSES

Too many object classes have been accessed. An application can only access less than 1024 object classes.

Implementation Specifics

This subroutine is part of Base Operating System (BOS) Runtime.

Related Information

The **odm_get_obj** subroutine.

The **odmchange** command, **odmcreate** command.

See ODM Example Code and Output in *AIX General Programming Concepts : Writing and Debugging Programs* for an example of a **.h** file.

Object Data Manager (ODM) Overview for Programmers in *AIX General Programming Concepts : Writing and Debugging Programs*.

odm_close_class Subroutine

Purpose

Closes an ODM object class.

Library

Object Data Manager Library (**libodm.a**)

Syntax

```
#include <odmi.h>

int odm_close_class (ClassSymbol)
CLASS_SYMBOL ClassSymbol;
```

Description

The **odm_close_class** subroutine closes the specified object class.

Parameters

ClassSymbol	Specifies a class symbol identifier returned from an **odm_open_class** subroutine. If the **odm_open_class** subroutine has not been called, then this identifier is the *ClassName*_**CLASS** structure that was created by the **odmcreate** command.

Return Values

Upon successful completion, a value of 0 is returned. If the **odm_close_class** subroutine is unsuccessful, a value of –1 is returned and the **odmerrno** variable is set to an error code.

Error Codes

Failure of the **odm_close_class** subroutine sets the **odmerrno** variable to one of the following error codes:

ODMI_CLASS_DNE
: The specified object class does not exist. Check path name and permissions.

ODMI_CLASS_PERMS
: The object class cannot be opened because of the file permissions.

ODMI_INVALID_CLXN
: Either the specified collection is not a valid object class collection or the collection does not contain consistent data.

ODMI_INVALID_PATH
: The specified path does not exist on the file system. Make sure the path is accessible.

ODMI_MAGICNO_ERR
: The class symbol does not identify a valid object class.

ODMI_OPEN_ERR
: Cannot open the object class. Check path name and permissions.

ODMI_TOOMANYCLASSES
: Too many object classes have been accessed. An application can only access less than 1024 object classes.

Implementation Specifics

This subroutine is part of Base Operating System (BOS) Runtime.

Related Information

The **odm_open_class** subroutine.

Object Data Manager (ODM) Overview for Programmers in *AIX General Programming Concepts : Writing and Debugging Programs*.

odm_create_class Subroutine

Purpose

Creates an object class.

Library

Object Data Manager Library (**libodm.a**)

Syntax

```
#include <odmi.h>

int odm_create_class (ClassSymbol)
CLASS_SYMBOL ClassSymbol;
```

Description

The **odm_create_class** subroutine creates an object class. However, the **.c** and **.h** files generated by the **odmcreate** command are required to be part of the application.

Parameters

ClassSymbol	Specifies a class symbol of the form *ClassName*_**CLASS,** which is declared in the **.h** file created by the **odmcreate** command.

Return Values

Upon successful completion, a value of 0 is returned. If the **odm_create_class** subroutine is unsuccessful, a value of –1 is returned and the **odmerrno** variable is set to an error code.

Error Codes

Failure of the **odm_create_class** subroutine sets the **odmerrno** variable to one of the following error codes:

- **ODMI_CLASS_EXISTS**
- **ODMI_CLASS_PERMS**
- **ODMI_INVALID_CLXN**
- **ODMI_INVALID_PATH**
- **ODMI_MAGICNO_ERR**
- **ODMI_OPEN_ERR**

See Appendix B, ”ODM Error Codes” for explanations of the ODM error codes.

Implementation Specifics

This subroutine is part of Base Operating System (BOS) Runtime.

Related Information

The **odm_mount_class** subroutine.

The **odmcreate** command.

See ODM Example Code and Output in *AIX General Programming Concepts : Writing and Debugging Programs* for an example of a **.h** file.

Object Data Manager (ODM) Overview for Programmers in *AIX General Programming Concepts : Writing and Debugging Programs*.

odm_err_msg Subroutine

Purpose

Returns an error message string.

Library

Object Data Manager Library (**libodm.a**)

Syntax

```
#include <odmi.h>

int odm_err_msg (ODMErrno, MessageString)
long ODMErrno;
char **MessageString;
```

Description

The **odm_err_msg** subroutine takes as input an *ODMErrno* parameter and an address in which to put the string pointer of the message string that corresponds to the input ODM error number. If no corresponding message is found for the input error number, a null string is returned and the subroutine is unsuccessful.

Parameters

ODMErrno	Specifies the error code for which the message string is retrieved.
MessageString	Specifies the address of a string pointer that will point to the returned error message string.

Return Values

Upon successful completion, a value of 0 is returned. If the **odm_err_msg** subroutine is unsuccessful, a value of –1 is returned, and the *MessageString* value returned is a null string.

Examples

The following example shows the use of the **odm_err_msg** subroutine:

```
#include <odmi.h>
char *error_message;

...
/*--------------------------------------------------------------*
/
/*ODMErrno was returned from a previous ODM subroutine call.
*/
/*--------------------------------------------------------------*
/
returnstatus = odm_err_msg ( odmerrno, &error_message );
if ( returnstatus < 0 )
   printf ( ”Retrieval of error message failed\n” );
else
   printf ( error_message );
```

Implementation Specifics

This subroutine is part of Base Operating System (BOS) Runtime.

Related Information

See Appendix B, ”ODM Error Codes” for explanations of the ODM error codes.

Object Data Manager (ODM) Overview for Programmers in *AIX General Programming Concepts : Writing and Debugging Programs*.

odm_free_list Subroutine

Purpose

Frees memory previously allocated for an **odm_get_list** subroutine.

Library

Object Data Manager Library (**libodm.a**)

Syntax

```
#include <odmi.h>

int odm_free_list (ReturnData, DataInfo)
struct ClassName *ReturnData;
struct listinfo *DataInfo;
```

Description

The **odm_free_list** subroutine recursively frees up a tree of memory object lists that were allocated for an **odm_get_list** subroutine.

Parameters

ReturnData Points to the array of *ClassName* structures returned from the **odm_get_list** subroutine.

DataInfo Points to the **listinfo** structure that was returned from the **odm_get_list** subroutine. The **listinfo** structure has the following form:

```
struct listinfo {
char ClassName[16];      /* class name for query *
/
char criteria[256];      /* query criteria */
int num;                 /* number of matches foun
d */
int valid;               /* for ODM use */
CLASS_SYMBOL class;      /* symbol for queried cla
ss */
};
```

Return Values

Upon successful completion, a value of 0 is returned. If the **odm_free_list** subroutine is unsuccessful, a value of –1 is returned and the **odmerrno** variable is set to an error code.

Error Codes

Failure of the **odm_free_list** subroutine sets the **odmerrno** variable to one of the following error codes:

ODMI_MAGICNO_ERR
The class symbol does not identify a valid object class.

ODMI_PARAMS
The parameters passed to the subroutine were not correct. Make sure there are the correct number of parameters and that they are valid.

Implementation Specifics

This subroutine is part of Base Operating System (BOS) Runtime.

Related Information

The **odm_get_list** subroutine.

Object Data Manager (ODM) Overview for Programmers in *AIX General Programming Concepts : Writing and Debugging Programs*.

odm_get_by_id Subroutine

Purpose

Retrieves an object from an ODM object class by its ID.

Library

Object Data Manager Library (**libodm.a**)

Syntax

```
#include <odmi.h>

struct ClassName *odm_get_by_id(ClassSymbol, ObjectID, ReturnData
)
CLASS_SYMBOL ClassSymbol;
int ObjectID;
struct ClassName *ReturnData;
```

Description

The **odm_get_by_id** subroutine retrieves an object from an object class. The object to be retrieved is specified by passing its *ObjectID* parameter from its corresponding *ClassName* structure.

Parameters

ClassSymbol	Specifies a class symbol identifier of the form *ClassName*_**CLASS,** which is declared in the **.h** file created by the **odmcreate** command.
ObjectID	Specifies an identifier retrieved from the corresponding *ClassName* structure of the object class.
ReturnData	Specifies a pointer to an instance of the C language structure corresponding to the object class referenced by the *ClassSymbol* parameter. The structure is declared in the **.h** file created by the **odmcreate** command and has the same name as the object class.

Return Values

Upon successful completion, a pointer to the *ClassName* structure containing the object is returned. If the **odm_get_by_id** subroutine is unsuccessful, a value of –1 is returned and the **odmerrno** variable is set to an error code.

Error Codes

Failure of the **odm_get_by_id** subroutine sets the **odmerrno** variable to one of the following error codes:

ODMI_CLASS_DNE
The specified object class does not exist. Check path name and permissions.

ODMI_CLASS_PERMS
The object class cannot be opened because of the file permissions.

ODMI_INVALID_CLXN
Either the specified collection is not a valid object class collection or the collection does not contain consistent data.

ODMI_INVALID_PATH

The specified path does not exist on the file system. Make sure the path is accessible.

ODMI_MAGICNO_ERR

The class symbol does not identify a valid object class.

ODMI_MALLOC_ERR

Cannot allocate sufficient storage. Try again later or contact the person responsible for the system.

ODMI_NO_OBJECT

The specified object identifier did not refer to a valid object.

ODMI_OPEN_ERR

Cannot open the object class. Check path name and permissions.

ODMI_PARAMS

The parameters passed to the subroutine were not correct. Make sure there are the correct number of parameters and that they are valid.

ODMI_TOOMANYCLASSES

Too many object classes have been accessed. An application can only access less than 1024 object classes.

Implementation Specifics

This subroutine is part of Base Operating System (BOS) Runtime.

Related Information

The **odm_get_obj**, **odm_get_first**, or **odm_get_next** subroutine.

The **odmcreate** command.

See ODM Example Code and Output in *AIX General Programming Concepts : Writing and Debugging Programs* for an example of a **.h** file.

Object Data Manager (ODM) Overview for Programmers in *AIX General Programming Concepts : Writing and Debugging Programs*.

odm_get_list Subroutine

Purpose

Retrieves all objects in an object class that match the specified criteria.

Library

Object Data Manager Library (**libodm.a**)

Syntax

#include <odmi.h>

struct *ClassName* ***odm_get_list (***ClasSymbol***,** *Criteria***,** *ListInfo***,** *MaxReturn* **,** *LinkDepth***)**
struct *ClassName*_**CLASS** *ClassSymbol***;**
char ***Criteria***;**
struct listinfo ***ListInfo***;**
int *MaxReturn***,** *LinkDepth***;**

Description

The **odm_get_list** subroutine takes an object class and criteria as input, and returns a list of objects that satisfy the input criteria. The subroutine opens and closes the object class around the subroutine if the object class was not previously opened. If the object class was previously opened, the subroutine leaves the object class open when it returns.

Parameters

ClassSymbol Specifies a class symbol identifier returned from an **odm_open_class** subroutine. If the **odm_open_class** subroutine has not been called, then this is the *ClassName*_**CLASS** structure created by the **odmcreate** command.

Criteria Specifies a string that contains the qualifying criteria for selecting the objects to remove.

ListInfo Specifies a structure containing information about the retrieval of the objects. The **listinfo** structure has the following form:

```
struct listinfo {
char ClassName[16];      /* class name used for query */
char criteria[256];   /* query criteria */
int num;                 /* number of matches found */
int valid;               /* for ODM use */
CLASS_SYMBOL class;   /* symbol for queried class */
};
```

MaxReturn Specifies the expected number of objects to be returned. This is used to control the increments in which storage for structures is allocated, to reduce the **realloc** subroutine copy overhead.

LinkDepth Specifies the number of levels to recurse for objects with **ODM_LINK** descriptors. A setting of 1 indicates only the top level is retrieved; 2 indicates **ODM_LINK**s will be followed from the top/first level only: 3 indicates **ODM_LINK**s will be followed at the first and second level, and so on.

Return Values

Upon successful completion, a pointer to an array of C language structures containing the objects is returned. This structure matches that described in the **.h** file that is returned from the **odmcreate** command. If no match is found, null is returned. If the **odm_get_list** subroutine fails, a value of –1 is returned and the **odmerrno** variable is set to an error code.

Error Codes

Failure of the **odm_get_list** subroutine sets the **odmerrno** variable to one of the following error codes:

ODMI_BAD_CRIT	The specified search criteria is incorrectly formed. Make sure the criteria contains only valid descriptor names and the search values are correct.
ODMI_CLASS_DNE	The specified object class does not exist. Check path name and permissions.
ODMI_CLASS_PERMS	The object class cannot be opened because of the file permissions.
ODMI_INTERNAL_ERR	An internal consistency problem occurred. Make sure the object class is valid or contact the person responsible for the system.
ODMI_INVALID_CLXN	Either the specified collection is not a valid object class collection or the collection does not contain consistent data.
ODMI_INVALID_PATH	The specified path does not exist on the file system. Make sure the path is accessible.
ODMI_LINK_NOT_FOUND	The object class that is accessed could not be opened. Make sure the linked object class is accessible.
ODMI_MAGICNO_ERR	The class symbol does not identify a valid object class.
ODMI_MALLOC_ERR	Cannot allocate sufficient storage. Try again later or contact the person responsible for the system.
ODMI_OPEN_ERR	Cannot open the object class. Check path name and permissions.
ODMI_PARAMS	The parameters passed to the subroutine were not correct. Make sure there are the correct number of parameters and that they are valid.
ODMI_TOOMANYCLASSES	Too many object classes have been accessed. An application can only access less than 1024 object classes.

Implementation Specifics

This subroutine is part of Base Operating System (BOS) Runtime.

Related Information

The **odm_get_by_id** subroutine, **odm_get_obj** subroutine, **odm_open_class** subroutine, or **odm_free_list** subroutine.

The **odmcreate** command, **odmget** command.

For information on qualifying criteria, see ”Understanding ODM Object Searches” in *AIX General Programming Concepts : Writing and Debugging Programs*.

See ODM Example Code and Output in *AIX General Programming Concepts : Writing and Debugging Programs* for an example of a **.h** file.

Object Data Manager (ODM) Overview for Programmers in *AIX General Programming Concepts : Writing and Debugging Programs*.

odm_get_obj, odm_get_first, or odm_get_next Subroutine

Purpose

Retrieves objects, one object at a time, from an ODM object class.

Library

Object Data Manager Library (**libodm.a**)

Syntax

```
#include <odmi.h>

struct ClassName *odm_get_obj (ClassSymbol, Criteria, ReturnData,
 FIRST_NEXT)

struct ClassName *odm_get_first (ClassSymbol, Criteria, ReturnDat
a)

struct ClassName *odm_get_next (ClassSymbol, ReturnData)

CLASS_SYMBOL ClassSymbol;
char *Criteria;
struct ClassName *ReturnData;
int FIRST_NEXT;
```

Description

The **odm_get_obj**, **odm_get_first**, and **odm_get_next** subroutines retrieve objects from ODM object classes and return the objects into C language structures defined by the **.h** file produced by the **odmcreate** command.

The **odm_get_obj**, **odm_get_first**, and **odm_get_next** subroutines open and close the specified object class if the object class was not previously opened. If the object class was previously opened, the subroutines leave the object class open upon return.

Parameters

ClassSymbol	Specifies a class symbol identifier returned from an **odm_open_class** subroutine. If the **odm_open_class** subroutine has not been called, then this identifier is the *ClassName*_**CLASS** structure that was created by the **odmcreate** command.
Criteria	Specifies the string that contains the qualifying criteria for retrieval of the objects.

ReturnData Specifies the pointer to the data structure in the **.h** file created by the **odmcreate** command. The name of the structure in the **.h** file is *ClassName*. If the *ReturnData* parameter is null (`ReturnData == null`), space is allocated for the parameter and the calling application is responsible for freeing this space at a later time.

If variable length character strings (vchar) are returned, they are referenced by pointers in the *ReturnData* structure. Calling applications must free each vchar between each call to the **odm_get** subroutines; otherwise storage will be lost.

FIRST_NEXT Specifies whether to get the first object that matches the criteria or the next object. Valid values are:

ODM_FIRST Retrieve the first object that matches the search criteria.

ODM_NEXT Retrieve the next object that matches the search criteria. The *Criteria* parameter is ignored if the *FIRST_NEXT* parameter is set to **ODM_NEXT**.

Return Values

Upon successful completion, a pointer to the retrieved object is returned. If no match is found, null is returned. If an **odm_get_obj**, **odm_get_first**, or **odm_get_next** subroutine is unsuccessful, a value of –1 is returned and the **odmerrno** variable is set to an error code.

Error Codes

Failure of the **odm_get_obj**, **odm_get_first** or **odm_get_next** subroutine sets the **odmerrno** variable to one of the following error codes:

ODMI_BAD_CRIT
The specified search criteria is incorrectly formed. Make sure the criteria contains only valid descriptor names and the search values are correct.

ODMI_CLASS_DNE
The specified object class does not exist. Check path name and permissions.

ODMI_CLASS_PERMS
The object class cannot be opened because of the file permissions.

ODMI_INTERNAL_ERR
An internal consistency problem occurred. Make sure the object class is valid or contact the person responsible for the system.

ODMI_INVALID_CLXN
Either the specified collection is not a valid object class collection or the collection does not contain consistent data.

ODMI_INVALID_PATH
The specified path does not exist on the file system. Make sure the path is accessible.

ODMI_MAGICNO_ERR
The class symbol does not identify a valid object class.

ODMI_MALLOC_ERR
Cannot allocate sufficient storage. Try again later or contact the person responsible for the system.

ODMI_OPEN_ERR

Cannot open the object class. Check path name and permissions.

ODMI_TOOMANYCLASSES

Too many object classes have been accessed. An application can only access less than 1024 object classes.

Implementation Specifics

These subroutines are part of Base Operating System (BOS) Runtime.

Related Information

The **odm_get_list** subroutine, **odm_open_class** subroutine, **odm_rm_by_id** subroutine, **odm_rm_obj** subroutine.

The **odmcreate** command, **odmget** command.

For more information about qualifying criteria, see ”Understanding ODM Object Searches” in *AIX General Programming Concepts : Writing and Debugging Programs*.

See ODM Example Code and Output in *AIX General Programming Concepts : Writing and Debugging Programs* for an example of a **.h** file.

Object Data Manager (ODM) Overview for Programmers in *AIX General Programming Concepts : Writing and Debugging Programs*.

odm_initialize Subroutine

Purpose

Prepares ODM for use by an application.

Library

Object Data Manager Library (**libodm.a**)

Syntax

```
#include <odmi.h>

int odm_initialize( )
```

Description

The **odm_initialize** subroutine starts ODM for use with an application program.

Return Values

Upon successful completion, a value of 0 is returned. If the **odm_initialize** subroutine is unsuccessful, a value of –1 is returned and the **odmerrno** variable is set to an error code.

Error Codes

Failure of the **odm_initialize** subroutine sets the **odmerrno** variable to one of the following error codes:

ODMI_INVALID_PATH
: The specified path does not exist on the file system. Make sure the path is accessible.

ODMI_MALLOC_ERR
: Cannot allocate sufficient storage. Try again later or contact the person responsible for the system.

Implementation Specifics

This subroutine is part of Base Operating System (BOS) Runtime.

Related Information

The **odm_terminate** subroutine.

Object Data Manager (ODM) Overview for Programmers in *AIX General Programming Concepts : Writing and Debugging Programs*.

odm_lock Subroutine

Purpose

Puts an exclusive lock on the requested path name.

Library

Object Data Manager Library (**libodm.a**)

Syntax

```
#include <odmi.h>

int odm_lock (LockPath, TimeOut)
char *LockPath;
int TimeOut;
```

Description

The **odm_lock** subroutine is used by an application to prevent other applications or methods from accessing an object class or group of object classes. A lock on a directory path name does not prevent another application from acquiring a lock on a subdirectory or object class within that directory.

Note: Coordination of locking is the responsibility of the application accessing the object classes.

The **odm_lock** subroutine returns a lock identifier that is used to call the **odm_unlock** subroutine.

Parameters

LockPath
Specifies a string containing the path name in the file system in which to locate object classes or the path name to an object class to lock.

TimeOut
Specifies the amount of time, in seconds, to wait if another application or method holds a lock on the requested object class or classes. The possible values for the *TimeOut* parameter are:

TimeOut **= ODM_NOWAIT**
The **odm_lock** subroutine is unsuccessful if the lock cannot be granted immediately.

TimeOut **=** *Integer*
The **odm_lock** subroutine waits the specified amount of seconds to retry an unsuccessful lock request.

TimeOut **= ODM_WAIT**
The **odm_lock** subroutine waits until the locked path name is freed from its current lock and then locks it.

Return Values

Upon successful completion, a lock identifier is returned. If the **odm_lock** subroutine is unsuccessful, a value of –1 is returned and the **odmerrno** variable is set to an error code.

Error Codes

Failure of the **odm_lock** subroutine sets the **odmerrno** variable to one of the following error codes:

ODMI_BAD_LOCK

Cannot set a lock on the file. Check path name and permissions.

ODMI_BAD_TIMEOUT

The time–out value was not valid. It must be a positive integer.

ODMI_BAD_TOKEN

Cannot create or open the lock file. Check path name and permissions.

ODMI_LOCK_BLOCKED

Cannot grant the lock. Another process already has the lock.

ODMI_LOCK_ENV

Cannot retrieve or set the lock environment variable. Remove some environment variables and try again.

ODMI_MALLOC_ERR

Cannot allocate sufficient storage. Try again later or contact the person responsible for the system.

ODMI_UNLOCK

Cannot unlock the lock file. Make sure the lock file exists.

Implementation Specifics

This subroutine is part of Base Operating System (BOS) Runtime.

Related Information

The **odm_unlock** subroutine.

Object Data Manager (ODM) Overview for Programmers in *AIX General Programming Concepts : Writing and Debugging Programs*.

odm_mount_class Subroutine

Purpose

Retrieves the class symbol structure for the specified object class name.

Library

Object Data Manager Library (**libodm.a**)

Syntax

```
#include <odmi.h>

CLASS_SYMBOL odm_mount_class (ClassName)
char *ClassName;
```

Description

The **odm_mount_class** subroutine retrieves the class symbol structure for a specified object class. The subroutine can be called by applications (for example, the ODM commands) that have no previous knowledge of the structure of an object class before trying to access that class. The **odm_mount_class** subroutine determines the class description from the object class header information and creates a **CLASS_SYMBOL** object class that is returned to the caller.

The object class is not opened by the **odm_mount_class** subroutine. Calling the subroutine subsequent times for an object class that is already open or mounted returns the same **CLASS_SYMBOL** object class.

Mounting a class that links to another object class recursively mounts to the linked class. However, if the recursive mount is unsuccessful, the original **odm_mount_class** subroutine does not fail; the **CLASS_SYMBOL** object class is set up with a null link.

Parameters

ClassName	Specifies the name of an object class from which to retrieve the class description.

Return Values

Upon successful completion, a **CLASS_SYMBOL** is returned. If the **odm_mount_class** subroutine is unsuccessful, a value of –1 is returned and the **odmerrno** variable is set to an error code.

Error Codes

Failure of the **odm_mount_class** subroutine sets the **odmerrno** variable to one of the following error codes:

ODMI_BAD_CLASSNAME
: The specified object class name does not match the object class name in the file. Check path name and permissions.

ODMI_BAD_CLXNNAME
: The specified collection name does not match the collection name in the file.

ODMI_CLASS_DNE
: The specified object class does not exist. Check path name and permissions.

ODMI_CLASS_PERMS
The object class cannot be opened because of the file permissions.

ODMI_CLXNMAGICNO_ERR
The specified collection is not a valid object class collection.

ODMI_INVALID_CLASS
The specified file is not an object class.

ODMI_INVALID_CLXN
Either the specified collection is not a valid object class collection or the collection does not contain consistent data.

ODMI_MAGICNO_ERR
The class symbol does not identify a valid object class.

ODMI_MALLOC_ERR
Cannot allocate sufficient storage. Try again later or contact the person responsible for the system.

ODMI_OPEN_ERR
Cannot open the object class. Check path name and permissions.

ODMI_PARAMS
The parameters passed to the subroutine were not correct. Make sure there are the correct number of parameters and that they are valid.

ODMI_TOOMANYCLASSES
Too many object classes have been accessed. An application can only access less than 1024 object classes.

Implementation Specifics

This subroutine is part of Base Operating System (BOS) Runtime.

Related Information

The **odm_create_class** subroutine.

Object Data Manager (ODM) Overview for Programmers in *AIX General Programming Concepts : Writing and Debugging Programs*.

odm_open_class Subroutine

Purpose

Opens an ODM object class.

Library

Object Data Manager Library (**libodm.a**)

Syntax

```
#include <odmi.h>

CLASS_SYMBOL odm_open_class (ClassSymbol)
CLASS_SYMBOL ClassSymbol;
```

Description

The **odm_open_class** subroutine can be called to open an object class. Most subroutines implicitly open a class if the class is not already open. However, an application may find it useful to perform an explicit open if, for example, several operations must be done on one object class before closing the class.

Parameter

ClassSymbol Specifies a class symbol of the form *ClassName*_**CLASS** that is declared in the **.h** file created by the **odmcreate** command.

Return Values

Upon successful completion, a *ClassSymbol* parameter for the object class is returned. If the **odm_open_class** subroutine is unsuccessful, a value of –1 is returned and the **odmerrno** variable is set to an error code.

Error Codes

Failure of the **odm_open_class** subroutine sets the **odmerrno** variable to one of the following error codes:

ODMI_CLASS_DNE
: The specified object class does not exist. Check path name and permissions.

ODMI_CLASS_PERMS
: The object class cannot be opened because of the file permissions.

ODMI_INVALID_PATH
: The specified path does not exist on the file system. Make sure the path is accessible.

ODMI_MAGICNO_ERR
: The class symbol does not identify a valid object class.

ODMI_OPEN_ERR
: Cannot open the object class. Check path name and permissions.

ODMI_TOOMANYCLASSES
: Too many object classes have been accessed. An application can only access less than 1024 object classes.

Implementation Specifics

This subroutine is part of Base Operating System (BOS) Runtime.

Related Information

The **odm_close_class** subroutine.

The **odmcreate** command.

See ODM Example Code and Output in *AIX General Programming Concepts : Writing and Debugging Programs* for an example of a **.h** file.

Object Data Manager (ODM) Overview for Programmers in *AIX General Programming Concepts : Writing and Debugging Programs*.

odm_rm_by_id Subroutine

Purpose

Removes objects specified by their IDs from an ODM object class.

Library

Object Data Manager Library (**libodm.a**)

Syntax

```
#include <odmi.h>

int odm_rm_by_id(ClassSymbol, ObjectID)
CLASS_SYMBOL ClassSymbol;
int ObjectID;
```

Description

The **odm_rm_by_id** subroutine is called to delete an object from an object class. The object to be deleted is specified by passing its object ID from its corresponding *ClassName* structure.

Parameters

ClassSymbol	Identifies a class symbol returned from an **odm_open_class** subroutine. If the **odm_open_class** subroutine has not been called, this is the *ClassName*_**CLASS** structure that was created by the **odmcreate** command.
ObjectID	Identifies the object. This information is retrieved from the corresponding *ClassName* structure of the object class.

Return Values

Upon successful completion, a value of 0 is returned. If the **odm_rm_by_id** subroutine is unsuccessful, a value of –1 is returned and the **odmerrno** variable is set to an error code.

Error Codes

Failure of the **odm_rm_by_id** subroutine sets the **odmerrno** variable to one of the following error codes:

ODMI_CLASS_DNE
: The specified object class does not exist. Check path name and permissions.

ODMI_CLASS_PERMS
: The object class cannot be opened because of the file permissions.

ODMI_FORK
: Cannot fork the child process. Make sure the child process is executable and try again.

ODMI_INVALID_CLXN
: Either the specified collection is not a valid object class collection or the collection does not contain consistent data.

ODMI_INVALID_PATH
: The specified path does not exist on the file system. Make sure the path is accessible.

ODMI_MAGICNO_ERR
The class symbol does not identify a valid object class.

ODMI_MALLOC_ERR
Cannot allocate sufficient storage. Try again later or contact the person responsible for the system.

ODMI_NO_OBJECT
The specified object identifier did not refer to a valid object.

ODMI_OPEN_ERR
Cannot open the object class. Check path name and permissions.

ODMI_OPEN_PIPE
Cannot open a pipe to a child process. Make sure the child process is executable and try again.

ODMI_PARAMS
The parameters passed to the subroutine were not correct. Make sure there are the correct number of parameters and that they are valid.

ODMI_READ_ONLY
The specified object class is opened as read–only and cannot be modified.

ODMI_READ_PIPE
Cannot read from the pipe of the child process. Make sure the child process is executable and try again.

ODMI_TOOMANYCLASSES
Too many object classes have been accessed. An application can only access less than 1024 object classes.

Implementation Specifics

This subroutine is part of Base Operating System (BOS) Runtime.

Related Information

The **odm_get_obj** subroutine, **odm_open_class** subroutine.

The **odmdelete** command.

Object Data Manager (ODM) Overview for Programmers in *AIX General Programming Concepts : Writing and Debugging Programs*.

odm_rm_class Subroutine

Purpose

Removes an object class from the file system.

Library

Object Data Manager Library (**libodm.a**)

Syntax

```
#include <odmi.h>

int odm_rm_class (ClassSymbol)
CLASS_SYMBOL ClassSymbol;
```

Description

The **odm_rm_class** subroutine removes an object class from the file system. All objects in the specified class are deleted.

Parameter

ClassSymbol Identifies a class symbol returned from the **odm_open_class** subroutine. If the **odm_open_class** subroutine has not been called, this is the *ClassName*_**CLASS** structure created by the **odmcreate** command.

Return Values

Upon successful completion, a value of 0 is returned. If the **odm_rm_class** subroutine is unsuccessful, a value of –1 is returned and the **odmerrno** variable is set to an error code.

Error Codes

Failure of the **odm_rm_class** subroutine sets the **odmerrno** variable to one of the following error codes:

ODMI_CLASS_DNE
The specified object class does not exist. Check path name and permissions.

ODMI_CLASS_PERMS
The object class cannot be opened because of the file permissions.

ODMI_INVALID_CLXN
Either the specified collection is not a valid object class collection or the collection does not contain consistent data.

ODMI_INVALID_PATH
The specified path does not exist on the file system. Make sure the path is accessible.

ODMI_MAGICNO_ERR
The class symbol does not identify a valid object class.

ODMI_OPEN_ERR
Cannot open the object class. Check path name and permissions.

ODMI_TOOMANYCLASSES

Too many object classes have been accessed. An application can only access less than 1024 object classes.

ODMI_UNLINKCLASS_ERR

Cannot remove the object class from the file system. Check path name and permissions.

ODMI_UNLINKCLXN_ERR

Cannot remove the object class collection from the file system. Check path name and permissions.

Implementation Specifics

This subroutine is part of Base Operating System (BOS) Runtime.

Related Information

The **odm_open_class** subroutine.

The **odmcreate** command, **odmdrop** command.

Object Data Manager (ODM) Overview for Programmers in *AIX General Programming Concepts : Writing and Debugging Programs*.

odm_rm_obj Subroutine

Purpose

Removes objects from an ODM object class.

Library

Object Data Manager Library (**libodm.a**)

Syntax

```
#include <odmi.h>

int odm_rm_obj (ClassSymbol, Criteria)
CLASS_SYMBOL ClassSymbol;
char *Criteria;
```

Description

The **odm_rm_obj** subroutine deletes objects from an object class.

Parameters

ClassSymbol	Identifies a class symbol returned from an **odm_open_class** subroutine. If the **odm_open_class** subroutine has not been called, this is the *ClassName*_**CLASS** structure that was created by the **odmcreate** command.
Criteria	Contains as a string the qualifying criteria for selecting the objects to remove.

Return Values

Upon successful completion, the number of objects deleted is returned. If the **odm_rm_obj** subroutine is unsuccessful, a value of –1 is returned and the **odmerrno** variable is set to an error code.

Error Codes

Failure of the **odm_rm_obj** subroutine sets the **odmerrno** variable to one of the following error codes:

ODMI_BAD_CRIT
The specified search criteria is incorrectly formed. Make sure the criteria contains only valid descriptor names and the search values are correct.

ODMI_CLASS_DNE
The specified object class does not exist. Check path name and permissions.

ODMI_CLASS_PERMS
The object class cannot be opened because of the file permissions.

ODMI_FORK
Cannot fork the child process. Make sure the child process is executable and try again.

ODMI_INTERNAL_ERR
An internal consistency problem occurred. Make sure the object class is valid or contact the person responsible for the system.

ODMI_INVALID_CLXN

Either the specified collection is not a valid object class collection or the collection does not contain consistent data.

ODMI_INVALID_PATH

The specified path does not exist on the file system. Make sure the path is accessible.

ODMI_MAGICNO_ERR

The class symbol does not identify a valid object class.

ODMI_MALLOC_ERR

Cannot allocate sufficient storage. Try again later or contact the person responsible for the system.

ODMI_OPEN_ERR

Cannot open the object class. Check path name and permissions.

ODMI_OPEN_PIPE

Cannot open a pipe to a child process. Make sure the child process is executable and try again.

ODMI_PARAMS

The parameters passed to the subroutine were not correct. Make sure there are the correct number of parameters and that they are valid.

ODMI_READ_ONLY

The specified object class is opened as read–only and cannot be modified.

ODMI_READ_PIPE

Cannot read from the pipe of the child process. Make sure the child process is executable and try again.

ODMI_TOOMANYCLASSES

Too many object classes have been accessed. An application can only access less than 1024 object classes.

Implementation Specifics

This subroutine is part of Base Operating System (BOS) Runtime.

Related Information

The **odm_add_obj** subroutine, **odm_open_class** subroutine.

The **odmcreate** command, **odmdelete** command.

Object Data Manager (ODM) Overview for Programmers in *AIX General Programming Concepts : Writing and Debugging Programs*.

For information on qualifying criteria, see ”Understanding ODM Object Searches” in *AIX General Programming Concepts : Writing and Debugging Programs*.

odm_run_method Subroutine

Purpose

Runs a specified method.

Library

Object Data Manager Library (**libodm.a**)

Syntax

```
#include <odmi.h>

int odm_run_method(MethodName, MethodParameters, NewStdOut, NewStdError)
char *MethodName, *MethodParameters;
char **NewStdOut, **NewStdError;
```

Description

The **odm_run_method** subroutine takes as input the name of the method to run, any parameters for the method, and addresses of locations for the **odm_run_method** subroutine to store pointers to the stdout (standard output) and stderr (standard error output) buffers. The application uses the pointers to access the stdout and stderr information generated by the method.

Parameters

MethodName	Indicates the method to execute. The method can already be known by the applications, or can be retrieved as part of an **odm_get_obj** subroutine call.
MethodParameters	Specifies a list of parameters for the specified method.
NewStdOut	Specifies the address of a pointer to the memory where the standard output of the method is stored. If the *NewStdOut* parameter points to a null value (`*NewStdOut == NULL`), standard output is not captured.
NewStdError	Specifies the address of a pointer to the memory where the standard error output of the method will be stored. If the *NewStdError* parameter points to a null value (`*NewStdError == NULL`), standard error output is not captured.

Return Values

Upon successful completion, a value of 0 is returned. If the **odm_run_method** subroutine fails, a value of –1 is returned and the **odmerrno** variable is set to an error code.

Error Codes

Failure of the **odm_run_method** subroutine sets the **odmerrno** variable to one of the following error codes:

ODMI_FORK

Cannot fork the child process. Make sure the child process is executable and try again.

ODMI_MALLOC_ERR

Cannot allocate sufficient storage. Try again later or contact the person responsible for the system.

ODMI_OPEN_PIPE
Cannot open a pipe to a child process. Make sure the child process is executable and try again.

ODMI_PARAMS
The parameters passed to the subroutine were not correct. Make sure there are the correct number of parameters and that they are valid.

ODMI_READ_PIPE
Cannot read from the pipe of the child process. Make sure the child process is executable and try again.

Implementation Specifics

This subroutine is part of Base Operating System (BOS) Runtime.

Related Information

The **odm_get_obj** subroutine.

Object Data Manager (ODM) Overview for Programmers in *AIX General Programming Concepts : Writing and Debugging Programs*.

odm_set_path Subroutine

Purpose

Sets the default path for locating object classes.

Library

Object Data Manager Library (**libodm.a**)

Syntax

```
#include <odmi.h>

char *odm_set_path (NewPath)
char *NewPath;
```

Description

The **odm_set_path** subroutine is used to set the default path for locating object classes. The subroutine allocates memory, sets the default path, and returns the pointer to memory. Once the operation is complete, the calling application should free the pointer using the **free** subroutine.

Parameters

NewPath	Contains, as a string, the path name in the file system in which to locate object classes.

Return Values

Upon successful completion, a string pointing to the previous default path is returned. If the **odm_set_path** subroutine is unsuccessful, a value of –1 is returned and the **odmerrno** variable is set to an error code.

Error Codes

Failure of the **odm_set_path** subroutine sets the **odmerrno** variable to one of the following error codes:

ODMI_INVALID_PATH
The specified path does not exist on the file system. Make sure the path is accessible.

ODMI_MALLOC_ERR
Cannot allocate sufficient storage. Try again later or contact the person responsible for the system.

Implementation Specifics

This subroutine is part of Base Operating System (BOS) Runtime.

Related Information

The **free** subroutine.

Object Data Manager (ODM) Overview for Programmers in *AIX General Programming Concepts : Writing and Debugging Programs*.

odm_set_perms Subroutine

Purpose

Sets the default permissions for an ODM object class at creation time.

Library

Object Data Manager Library (**libodm.a**)

Syntax

```
#include <odmi.h>

int odm_set_perms (NewPermissions)
int NewPermissions;
```

Description

The **odm_set_perms** subroutine defines the default permissions to assign to object classes at creation.

Parameters

NewPermissions	Specifies the new default permissions parameter as an integer.

Return Values

Upon successful completion, the current default permissions are returned. If the **odm_set_perms** subroutine is unsuccessful, a value of –1 is returned.

Implementation Specifics

This subroutine is part of Base Operating System (BOS) Runtime.

Related Information

See Appendix B, ”ODM Error Codes” for explanations of the ODM error codes.

Object Data Manager (ODM) Overview for Programmers in *AIX General Programming Concepts : Writing and Debugging Programs*.

odm_terminate Subroutine

Purpose

Terminates an ODM session.

Library

Object Data Manager Library (**libodm.a**)

Syntax

```
#include <odmi.h>

int odm_terminate ( )
```

Description

The **odm_terminate** subroutine performs the cleanup necessary to terminate an ODM session. After running an **odm_terminate** subroutine, an application must issue an **odm_initialize** subroutine to resume ODM operations.

Return Values

Upon successful completion, a value of 0 is returned. If the **odm_terminate** subroutine is unsuccessful, a value of –1 is returned and the **odmerrno** variable is set to an error code.

Error Codes

Failure of the **odm_terminate** subroutine sets the **odmerrno** variable to one of the following error codes:

ODMI_CLASS_DNE
: The specified object class does not exist. Check path name and permissions.

ODMI_CLASS_PERMS
: The object class cannot be opened because of the file permissions.

ODMI_INVALID_CLXN
: Either the specified collection is not a valid object class collection or the collection does not contain consistent data.

ODMI_INVALID_PATH
: The specified path does not exist on the file system. Make sure the path is accessible.

ODMI_LOCK_ID
: The lock identifier does not refer to a valid lock. The lock identifier must be the same as what was returned from the **odm_lock** subroutine.

ODMI_MAGICNO_ERR
: The class symbol does not identify a valid object class.

ODMI_OPEN_ERR
: Cannot open the object class. Check path name and permissions.

ODMI_TOOMANYCLASSES
: Too many object classes have been accessed. An application can only access less than 1024 object classes.

ODMI_UNLOCK
Cannot unlock the lock file. Make sure the lock file exists.

Implementation Specifics

This subroutine is part of Base Operating System (BOS) Runtime.

Related Information

The **odm_initialize** subroutine.

Object Data Manager (ODM) Overview for Programmers in *AIX General Programming Concepts : Writing and Debugging Programs*.

odm_unlock Subroutine

Purpose

Releases a lock put on a path name.

Library

Object Data Manager Library (**libodm.a**)

Syntax

```
#include <odmi.h>

int odm_unlock (LockID)
int LockID;
```

Description

The **odm_unlock** subroutine releases a previously granted lock on a path name. This path name can be a directory containing subdirectories and object classes.

Parameters

LockID	Identifies the lock returned from the **odm_lock** subroutine.

Return Values

Upon successful completion a value of 0 is returned. If the **odm_unlock** subroutine is unsuccessful, a value of –1 is returned and the **odmerrno** variable is set to an error code.

Error Codes

Failure of the **odm_unlock** subroutine sets the **odmerrno** variable to one of the following error codes:

ODMI_LOCK_ID
: The lock identifier does not refer to a valid lock. The lock identifier must be the same as what was returned from the **odm_lock** subroutine.

ODMI_UNLOCK
: Cannot unlock the lock file. Make sure the lock file exists.

Implementation Specifics

This subroutine is part of Base Operating System (BOS) Runtime.

Related Information

The **odm_lock** subroutine.

Object Data Manager (ODM) Overview for Programmers in *AIX General Programming Concepts : Writing and Debugging Programs*.

open, openx, open64, creat, or creat64 Subroutine

Purpose

Opens a file for reading or writing.

Syntax

```
#include <fcntl.h>

int open (Path, OFlag, [Mode])
const char *Path;
int OFlag;
mode_t Mode;

int openx (Path, OFlag, Mode, Extension)
const char *Path;
int OFlag;
mode_t Mode;
int Extension;

int creat (Path, [Mode])
const char *Path;
mode_t Mode;
```

Note: The **open64** and **creat64** subroutines apply to Version 4.2 and later releases.

```
int open64 (Path, [Mode])
const char *Path;
int OFlag;
mode_t Mode;

int creat64 (Path, [Mode])
const char *Path;
mode_t Mode;
```

Description

Note: The **open64** and **creat64** subroutines apply to Version 4.2 and later releases.

The **open**, **openx**, and **creat** subroutines establish a connection between the file named by the *Path* parameter and a file descriptor. The opened file descriptor is used by subsequent I/O subroutines, such as **read** and **write**, to access that file.

The **openx** subroutine is the same as the **open** subroutine, with the addition of an *Extension* parameter, which is provided for device driver use. The **creat** subroutine is equivalent to the **open** subroutine with the **O_WRONLY**, **O_CREAT**, and **O_TRUNC** flags set.

The returned file descriptor is the lowest file descriptor not previously open for that process. No process can have more than **OPEN_MAX** file descriptors open simultaneously.

The file offset, marking the current position within the file, is set to the beginning of the file. The new file descriptor is set to remain open across exec subroutines.

The **open64** and **creat64** subroutines are equivalent to the **open** and **creat** subroutines except that the **O_LARGEFILE** flag is set in the open file description associated with the returned file descriptor. This flag allows files larger than **OFF_MAX** to be accessed. If the caller attempts to open a file larger than **OFF_MAX** and **O_LARGEFILE** is not set, the open will fail and **errno** will be set to **EOVERFLOW**.

In the large file enabled programming environment, **open** is redefined to be **open64** and **creat** is redefined to be **creat64**.

Parameters

Path Specifies the file to be opened.

Mode Specifies the read, write, and execute permissions of the file to be created (requested by the **O_CREAT** flag). If the file already exists, this parameter is ignored. The *Mode* parameter is constructed by logically ORing one or more of the following values, which are defined in the **sys/mode.h** file:

S_ISUID Enables the **setuid** attribute for an executable file. A process executing this program acquires the access rights of the owner of the file.

S_ISGID Enables the **setgid** attribute for an executable file. A process executing this program acquires the access rights of the group of the file. Also, enables the group–inheritance attribute for a directory. Files created in this directory have a group equal to the group of the directory.

The following attributes apply only to files that are directly executable. They have no meaning when applied to executable text files such as shell scripts and **awk** scripts.

S_ISVTX Enables the **link/unlink** attribute for a directory. Files cannot be linked to in this directory. Files can only be unlinked if the requesting process has write permission for the directory and is either the owner of the file or the directory.

S_ISVTX Enables the **save text** attribute for an executable file. The program is not unmapped after usage.

S_ENFMT Enables enforcement–mode record locking for a regular file. File locks requested with the **lockf** subroutine are enforced.

S_IRUSR Permits the file's owner to read it.

S_IWUSR Permits the file's owner to write to it.

S_IXUSR Permits the file's owner to execute it (or to search the directory).

S_IRGRP Permits the file's group to read it.

S_IWGRP Permits the file's group to write to it.

S_IXGRP Permits the file's group to execute it (or to search the directory).

S_IROTH Permits others to read the file.

S_IWOTH Permits others to write to the file.

S_IXOTH Permits others to execute the file (or to search the directory).

Other mode values exist that can be set with the **mknod** subroutine but not with the **chmod** subroutine.

Extension Provides communication with character device drivers that require additional information or return additional status. Each driver interprets the *Extension* parameter in a device–dependent way, either as a value or as a pointer to a communication area. Drivers must apply reasonable defaults when the *Extension* parameter value is 0.

OFlag Specifies the type of access, special open processing, the type of update, and the initial state of the open file. The parameter value is constructed by logically ORing special open processing flags. These flags are defined in the **fcntl.h** file and are described in the following flags.

Flags That Specify Access Type

The following *OFlag* parameter flag values specify type of access:

O_RDONLY The file is opened for reading only.

O_WRONLY The file is opened for writing only.

O_RDWR The file is opened for both reading and writing.

Note: One of the file access values must be specified. Do not use **O_RDONLY**, **O_WRONLY**, or **O_RDWR** together. If none is set, none is used, and the results are unpredictable.

Flags That Specify Special Open Processing

The following *OFlag* parameter flag values specify special open processing:

O_CREAT If the file exists, this flag has no effect, except as noted under the **O_EXCL** flag. If the file does not exist, a regular file is created with the following characteristics:

- The owner ID of the file is set to the effective user ID of the process.
- The group ID of the file is set to the group ID of the parent directory if the parent directory has the **SetGroupID** attribute (**S_ISGID** bit) set. Otherwise, the group ID of the file is set to the effective group ID of the calling process.
- The file permission and attribute bits are set to the value of the *Mode* parameter, modified as follows:
  - All bits set in the process file mode creation mask are cleared. (The file creation mask is described in the **umask** subroutine.)
  - The **S_ISVTX** attribute bit is cleared.

O_EXCL If the **O_EXCL** and **O_CREAT** flags are set, the open is unsuccessful if the file exists.

Note: The **O_EXCL** flag is not fully supported for Network File Systems (NFS). The NFS protocol does not guarantee the designed function of the **O_EXCL** flag.

O_NSHARE Assures that no process has this file open and precludes subsequent opens. If the file is on a physical file system and is already open, this open is unsuccessful and returns immediately unless the *OFlag* parameter also specifies the **O_DELAY** flag. This flag is effective only with physical file systems.

Note: This flag is not supported by NFS.

O_RSHARE Assures that no process has this file open for writing and precludes subsequent opens for writing. The calling process can request write access. If the file is on a physical file system and is open for writing or open with the **O_NSHARE** flag, this open fails and returns immediately unless the *OFlag* parameter also specifies the **O_DELAY** flag.

Note: This flag is not supported by NFS.

O_DEFER The file is opened for deferred update. Changes to the file are not reflected on permanent storage until an **fsync** subroutine operation is performed. If no **fsync** subroutine operation is performed, the changes are discarded when the file is closed.

Note: This flag is not supported by NFS.

Note: This flag causes modified pages to be backed by paging space. Before using this flag make sure there is sufficient paging space.

O_NOCTTY This flag specifies that the controlling terminal should not be assigned during this open.

O_TRUNC If the file does not exist, this flag has no effect. If the file exists, is a regular file, and is successfully opened with the **O_RDWR** flag or the **O_WRONLY** flag, all of the following apply:

- The length of the file is truncated to 0.
- The owner and group of the file are unchanged.
- The **SetUserID** attribute of the file mode is cleared.
- The **SetUserID** attribute of the file is cleared.

O_DIRECT This flag specifies that direct i/o will be used for this file while it is opened.

The **open** subroutine is unsuccessful if any of the following conditions are true:

- The file supports enforced record locks and another process has locked a portion of the file.
- The file is on a physical file system and is already open with the **O_RSHARE** flag or the **O_NSHARE** flag.
- The file does not allow write access.
- The file is already opened for deferred update.

Flag That Specifies Type of Update

A program can request some control on when updates should be made permanent for a regular file opened for write access. The following *OFlag* parameter values specify the type of update performed:

O_SYNC:	If set, updates to regular files and writes to block devices are synchronous updates. File update is performed by the following subroutines: • **fclear** • **ftruncate** • **open** with **O_TRUNC** • **write** On return from a subroutine that performs a synchronous update (any of the preceding subroutines, when the **O_SYNC** flag is set), the program is assured that all data for the file has been written to permanent storage, even if the file is also open for deferred update.

Note: The **O_DSYNC** flag applies to AIX Version 4.2.1 and later releases.

O_DSYNC:	If set, the file data as well as all file system meta–data required to retrieve the file data are written to their permanent storage locations. File attributes such as access or modification times are not required to retrieve file data, and as such, they are not guaranteed to be written to their permanent storage locations before the preceding subroutines return. (Subroutines listed in the **O_SYNC** description.)
O_SYNC | O_DSYNC:	If both flags are set, the file's data and all of the file's meta–data (including access time) are written to their permanent storage locations.

Note: The **O_RSYNC** flag applies to AIX Version 4.3.0 and later releases.

O_RSYNC:	This flag is used in combination with **O_SYNC** or **D_SYNC**, and it extends their write operation behaviors to read operations. For example, when **O_SYNC** and **R_SYNC** are both set, a read operation will not return until the file's data and all of the file's meta–data (including access time) are written to their permanent storage locations.

Flags That Define the Open File Initial State

The following *OFlag* parameter flag values define the initial state of the open file:

O_APPEND	The file pointer is set to the end of the file prior to each write operation.
O_DELAY	Specifies that if the **open** subroutine could not succeed due to an inability to grant the access on a physical file system required by the **O_RSHARE** flag or the **O_NSHARE** flag, the process blocks instead of returning the **ETXTBSY** error code.
O_NDELAY	Opens with no delay.
O_NONBLOCK	Specifies that the **open** subroutine should not block.

The **O_NDELAY** flag and the **O_NONBLOCK** flag are identical except for the value returned by the **read** and **write** subroutines. These flags mean the process does not block on the state of an object, but does block on input or output to a regular file or block device.

The **O_DELAY** flag is relevant only when used with the **O_NSHARE** or **O_RSHARE** flags. It is unrelated to the **O_NDELAY** and **O_NONBLOCK** flags.

General Notes on OFlag Parameter Flags

The effect of the **O_CREAT** flag is immediate, even if the file is opened with the **O_DEFER** flag.

When opening a file on a physical file system with the **O_NSHARE** flag or the **O_RSHARE** flag, if the file is already open with conflicting access the following can occur:

- If the **O_DELAY** flag is clear (the default), the **open** subroutine is unsuccessful.
- If the **O_DELAY** flag is set, the **open** subroutine blocks until there is no conflicting open. There is no deadlock detection for processes using the **O_DELAY** flag.

When opening a file on a physical file system that has already been opened with the **O_NSHARE** flag, the following can occur:

- If the **O_DELAY** flag is clear (the default), the open is unsuccessful immediately.
- If the **O_DELAY** flag is set, the open blocks until there is no conflicting open.

When opening a file with the **O_RDWR**, **O_WRONLY**, or **O_TRUNC** flag, and the file is already open with the **O_RSHARE** flag:

- If the **O_DELAY** flag is clear (the default), the open is unsuccessful immediately.
- If the **O_DELAY** flag is set, the open blocks until there is no conflicting open.

When opening a first–in–first–out (FIFO) with the **O_RDONLY** flag, the following can occur:

- If the **O_NDELAY** and **O_NONBLOCK** flags are clear, the open blocks until a process opens the file for writing. If the file is already open for writing (even by the calling process), the **open** subroutine returns without delay.
- If the **O_NDELAY** flag or the **O_NONBLOCK** flag is set, the open succeeds immediately even if no process has the FIFO open for writing.

When opening a FIFO with the **O_WRONLY** flag, the following can occur:

- If the **O_NDELAY** and **O_NONBLOCK** flags are clear (the default), the open blocks until a process opens the file for reading. If the file is already open for writing (even by the calling process), the **open** subroutine returns without delay.
- If the **O_NDELAY** flag or the **O_NONBLOCK** flag is set, the **open** subroutine returns an error if no process currently has the file open for reading.

When opening a block special or character special file that supports nonblocking opens, such as a terminal device, the following can occur:

- If the **O_NDELAY** and **O_NONBLOCK** flags are clear (the default), the open blocks until the device is ready or available.
- If the **O_NDELAY** flag or the **O_NONBLOCK** flag is set, the **open** subroutine returns without waiting for the device to be ready or available. Subsequent behavior of the device is device–specific.

Any additional information on the effect, if any, of the **O_NDELAY**, **O_RSHARE**, **O_NSHARE**, and **O_DELAY** flags on a specific device is documented in the description of the special file related to the device type.

If path refers to a STREAMS file, *oflag* may be constructed from **O_NONBLOCK** OR–ed with either **O_RDONLY**, **O_WRONLY** or **O_RDWR**. Other flag values are not applicable to STREAMS devices and have no effect on them. The value **O_NONBLOCK** affects the operation of STREAMS drivers and certain functions applied to file descriptors associated with STREAMS files. For STREAMS drivers, the implementation of **O_NONBLOCK** is device–specific.

If path names the master side of a pseudo–terminal device, then it is unspecified whether **open** locks the slave side so that it cannot be opened. Portable applications must call **unlockpt** before opening the slave side.

The largest value that can be represented correctly in an object of type **off_t** will be established as the offset maximum in the open file description.

Return Values

Upon successful completion, the file descriptor, a nonnegative integer, is returned. Otherwise, a value of –1 is returned, no files are created or modified, and the **errno** global variable is set to indicate the error.

Error Codes

The **open**, **openx**, and **creat** subroutines are unsuccessful and the named file is not opened if one or more of the following are true:

EACCES
One of the following is true:
- The file exists and the type of access specified by the *OFlag* parameter is denied.
- Search permission is denied on a component of the path prefix specified by the *Path* parameter. Access could be denied due to a secure mount.
- The file does not exist and write permission is denied for the parent directory of the file to be created.
- The **O_TRUNC** flag is specified and write permission is denied.

EAGAIN
The **O_TRUNC** flag is set and the named file contains a record lock owned by another process.

EDQUOT
The directory in which the entry for the new link is being placed cannot be extended, or an i–node could not be allocated for the file, because the user or group quota of disk blocks or i–nodes in the file system containing the directory has been exhausted.

EEXIST
The **O_CREAT** and **O_EXCL** flags are set and the named file exists.

EFBIG
An attempt was made to write a file that exceeds the process' file limit or the maximum file size. If the user has set the envirnoment variable **XPG_SUS_ENV=ON** prior to execution of the process, then the **SIGXFSZ** signal is posted to the process when exceeding the process' file size limit.

EINTR
A signal was caught during the **open** subroutine.

EIO
The *path* parameter names a STREAMS file and a hangup or error occured.

EISDIR
Named file is a directory and write access is required (the **O_WRONLY** or **O_RDWR** flag is set in the *OFlag* parameter).

EMFILE
The system limit for open file descriptors per process has already been reached (**OPEN_MAX**).

ENAMETOOLONG
The length of the *Path* parameter exceeds the system limit (**PATH_MAX**); or a path–name component is longer than **NAME_MAX** and **_POSIX_NO_TRUNC** is in effect.

ENFILE
The system file table is full.

ENOENT
The **O_CREAT** flag is not set and the named file does not exist; or the **O_CREAT** flag is not set and either the path prefix does not exist or the *Path* parameter points to an empty string.

ENOMEM
The *Path* parameter names a STREAMS file and the system is unable to allocate resources.

ENOSPC The directory or file system that would contain the new file cannot be extended.

ENOSR The *Path* argument names a STREAMS–based file and the system is unable to allocate a STREAM.

ENOTDIR A component of the path prefix specified by the *Path* component is not a directory.

ENXIO One of the following is true:

- Named file is a character special or block special file, and the device associated with this special file does not exist.
- Named file is a multiplexed special file and either the channel number is outside of the valid range or no more channels are available.
- The **O_DELAY** flag or the **O_NONBLOCK** flag is set, the named file is a FIFO, the **O_WRONLY** flag is set, and no process has the file open for reading.

EROFS Named file resides on a read–only file system and write access is required (either the **O_WRONLY**, **O_RDWR**, **O_CREAT** (if the file does not exist), or **O_TRUNC** flag is set in the *OFlag* parameter).

ETXTBSY File is on a physical file system and is already open in a manner (with the **O_RSHARE** or **O_NSHARE** flag) that precludes this open; or the **O_NSHARE** or **O_RSHARE** flag was requested with the **O_NDELAY** flag set, and there is a conflicting open on a physical file system.

Note: The **EOVERFLOW** error code applies to Version 4.2 and later releases.

EOVERFLOW A call was made to **open** and **creat** and the file already existed and its size was larger than **OFF_MAX** and the **O_LARGEFILE** flag was not set.

The **open**, **openx**, and **creat** subroutines are unsuccessful if one of the following are true:

EFAULT The *Path* parameter points outside of the allocated address space of the process.

EINVAL The value of the *OFlag* parameter is not valid.

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

ETXTBSY The file specified by the *Path* parameter is a pure procedure (shared text) file that is currently executing, and the **O_WRONLY** or **O_RDWR** flag is set in the *OFlag* parameter.

Implementation Specifics

These subroutines are part of Base Operating System (BOS) Runtime.

Related Information

The **chmod** subroutine, **close** subroutine, **exec** subroutine, **fcntl**, **dup**, or **dup2** subroutine, **fsync** subroutine, **ioctl** subroutine, **lockfx** subroutine, **lseek** subroutine, **read** subroutine, **stat** subroutine, **umask** subroutine, **write** subroutine.

The Input and Output Handling Programmer's Overview in *AIX General Programming Concepts : Writing and Debugging Programs*.

opendir, readdir, telldir, seekdir, rewinddir, or closedir Subroutine

Purpose

Performs operations on directories.

Library

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

Syntax

```
#include <dirent.h>

DIR *opendir (DirectoryName)
const char *DirectoryName;

struct dirent *readdir (DirectoryPointer)
DIR *DirectoryPointer;

long int telldir(DirectoryPointer)
DIR *DirectoryPointer;

void seekdir(DirectoryPointer,Location)
DIR *DirectoryPointer;
long Location;

void rewinddir (DirectoryPointer)
DIR *DirectoryPointer;

int closedir (DirectoryPointer)
DIR *DirectoryPointer;
```

Description

Attention: Do not use the **readdir** subroutine in a multithreaded environment. See the multithread alternative in the **readdir_r** subroutine article.

The **opendir** subroutine opens the directory designated by the *DirectoryName* parameter and associates a directory stream with it.

Note: An open directory must always be closed with the **closedir** subroutine to ensure that the next attempt to open that directory is successful.

The **opendir** subroutine also returns a pointer to identify the directory stream in subsequent operations. The null pointer is returned when the directory named by the *DirectoryName* parameter cannot be accessed or when not enough memory is available to hold the entire stream. A successful call to any of the **exec** functions closes any directory streams opened in the calling process.

The **readdir** subroutine returns a pointer to the next directory entry. The **readdir** subroutine returns entries for . (dot) and .. (dot dot), if present, but never returns an invalid entry (with `d_ino` set to 0). When it reaches the end of the directory, or when it detects an invalid **seekdir** operation, the **readdir** subroutine returns the null value. The returned pointer designates data that may be overwritten by another call to the **readdir** subroutine on the same directory stream. A call to the **readdir** subroutine on a different directory stream does not overwrite this data. The **readdir** subroutine marks the `st_atime` field of the directory for update each time the directory is actually read.

The **telldir** subroutine returns the current location associated with the specified directory stream.

The **seekdir** subroutine sets the position of the next **readdir** subroutine operation on the directory stream. An attempt to seek an invalid location causes the **readdir** subroutine to return the null value the next time it is called. The position should be that returned by a previous **telldir** subroutine call.

The **rewinddir** subroutine resets the position of the specified directory stream to the beginning of the directory.

The **closedir** subroutine closes a directory stream and frees the structure associated with the *DirectoryPointer* parameter.

If you use the **fork** subroutine to create a new process from an existing one, either the parent or the child (but not both) may continue processing the directory stream using the **readdir**, **rewinddir**, or **seekdir** subroutine.

Parameters

DirectoryName	Names the directory.
DirectoryPointer	Points to the **DIR** structure of an open directory.
Location	Specifies the offset of an entry relative to the start of the directory.

Return Values

On successful completion, the **opendir** subroutine returns a pointer to an object of type **DIR**. Otherwise, a null value is returned and the **errno** global variable is set to indicate the error.

On successful completion, the **readdir** subroutine returns a pointer to an object of type **struct dirent**. Otherwise, a null value is returned and the **errno** global variable is set to indicate the error. When the end of the directory is encountered, a null value is returned and the **errno** global variable is not changed by this function call.

On successful completion, the **closedir** subroutine returns a value of 0. Otherwise, a value of –1 is returned and the **errno** global variable is set to indicate the error.

Error Codes

If the **opendir** subroutine is unsuccessful, it returns a null value and sets the **errno** global variable to one of the following values:

EACCES	Indicates that search permission is denied for any component of the *DirectoryName* parameter, or read permission is denied for the *DirectoryName* parameter.
ENAMETOOLONG	Indicates that the length of the *DirectoryName* parameter argument exceeds the **PATH_MAX** value, or a path–name component is longer than the **NAME_MAX** value while the **POSIX_NO_TRUNC** value is in effect.
ENOENT	Indicates that the named directory does not exist.
ENOTDIR	Indicates that a component of the *DirectoryName* parameter is not a directory.
EMFILE	Indicates that too many file descriptors are currently open for the process.
ENFILE	Indicates that too many file descriptors are currently open in the system.

If the **readdir** subroutine is unsuccessful, it returns a null value and sets the **errno** global variable to the following value:

EBADF	Indicates that the *DirectoryPointer* parameter argument does not refer to an open directory stream.

If the **closedir** subroutine is unsuccessful, it returns a value of –1 and sets the **errno** global variable to the following value:

EBADF	Indicates that the *DirectoryPointer* parameter argument does not refer to an open directory stream.

Examples

To search a directory for the entry `name`:

```
len = strlen(name);
DirectoryPointer = opendir(".");
for (dp = readdir(DirectoryPointer); dp != NULL; dp =
 readdir(DirectoryPointer))
        if (dp->d_namlen == len && !strcmp(dp->d_name, name)) {
                closedir(DirectoryPointer);
                return FOUND;
        }
closedir(DirectoryPointer);
return NOT_FOUND;
```

Implementation Specifics

These subroutines are part of Base Operating System (BOS) Runtime.

Related Information

The **close** subroutine, **exec** subroutines, **fork** subroutine, **lseek** subroutine, **openx**, **open**, or **creat** subroutine, **read**, **readv**, **readx**, or **readvx** subroutine, **scandir** or **alphasort** subroutine.

Files, Directories, and File Systems for Programmers in *AIX General Programming Concepts : Writing and Debugging Programs*.

passwdexpired Subroutine

Purpose

Checks the user's password to determine if it has expired.

Syntax

```
passwdexpired (UserName, Message)
char *UserName;
char **Message;
```

Description

The **passwdexpired** subroutine checks a user's password to determine if it has expired. The subroutine checks the **registry** variable in the **/etc/security/user** file to ascertain where the user is administered. If the **registry** variable is not defined, the **passwdexpired** subroutine checks the local, NIS, and DCE databases for the user definition and expiration time.

The **passwdexpired** subroutine may pass back informational messages, such as how many days remain until password expiration.

Parameters

UserName	Specifies the user's name whose password is to be checked.
Message	Points to a pointer that the **passwdexpired** subroutine allocates memory for and fills in. This string is suitable for printing and issues messages, such as in how many days the password will expire.

Return Values

Upon successful completion, the **passwdexpired** subroutine returns a value of 0. If this subroutine fails, it returns one of the following values:

1	Indicates that the password is expired, and the user must change it.
2	Indicates that the password is expired, and only a system administrator may change it.
–1	Indicates that an internal error has occurred, such as a memory allocation (malloc) failure or database corruption.

Error Codes

The **passwdexpired** subroutine fails if one or more of the following values is true:

ENOENT	Indicates that the user could not be found.
EPERM	Indicates that the user did not have permission to check password expiration.
ENOMEM	Indicates that memory allocation (malloc) failed.
EINVAL	Indicates that the parameters are not valid.

Implementation Specifics

This subroutine is part of Base Operating System (BOS) Runtime.

Related Information

The **authenticate** subroutine.

The **login** command.

pathconf or fpathconf Subroutine

Purpose

Retrieves file–implementation characteristics.

Library

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

Syntax

```
#include <unistd.h>

long pathconf (Path, Name)
const char *Path;
int Name;

long fpathconf(FileDescriptor, Name)
int FileDescriptor, Name;
```

Description

The **pathconf** subroutine allows an application to determine the characteristics of operations supported by the file system contained by the file named by the *Path* parameter. Read, write, or execute permission of the named file is not required, but all directories in the path leading to the file must be searchable.

The **fpathconf** subroutine allows an application to retrieve the same information for an open file.

Parameters

Path
: Specifies the path name.

FileDescriptor
: Specifies an open file descriptor.

Name
: Specifies the configuration attribute to be queried. If this attribute is not applicable to the file specified by the *Path* or *FileDescriptor* parameter, the **pathconf** subroutine returns an error. Symbolic values for the *Name* parameter are defined in the **unistd.h** file:

_PC_LINK_MAX
: Specifies the maximum number of links to the file.

_PC_MAX_CANON
: Specifies the maximum number of bytes in a canonical input line. This value is applicable only to terminal devices.

_PC_MAX_INPUT
: Specifies the maximum number of bytes allowed in an input queue. This value is applicable only to terminal devices.

_PC_NAME_MAX
: Specifies the maximum number of bytes in a file name, not including a terminating null character. This number can range from 14 through 255. This value is applicable only to a directory file.

_PC_PATH_MAX
: Specifies the maximum number of bytes in a path name, not including a terminating null character.

_PC_PIPE_BUF Specifies the maximum number of bytes guaranteed to be written atomically. This value is applicable only to a first–in–first–out (FIFO).

_PC_CHOWN_RESTRICTED
Returns 0 if the use of the **chown** subroutine is restricted to a process with appropriate privileges, and if the **chown** subroutine is restricted to changing the group ID of a file only to the effective group ID of the process or to one of its supplementary group IDs.

_PC_NO_TRUNC Returns 0 if long component names are truncated. This value is applicable only to a directory file.

_PC_VDISABLE This is always 0. No disabling character is defined. This value is applicable only to a terminal device.

Note: The **_PC_FILESIZEBITS** and **PC_SYNC_IO** flags apply to AIX Version 4.3 and later releases.

_PC_FILESIZEBITS Returns the minimum number of bits required to hold the file system's maximum file size as a signed integer. The smallest value returned is **32**.

_PC_SYNC_IO Returns **–1** if the file system does not support the **Synchronized Input and Output** option. Any value other than **–1** is returned if the file system supports the option.

Notes:

1. If the *Name* parameter has a value of **_PC_LINK_MAX**, and if the *Path* or *FileDescriptor* parameter refers to a directory, the value returned applies to the directory itself.
2. If the *Name* parameter has a value of **_PC_NAME_MAX** or **_PC_NO_TRUNC**, and if the *Path* or *FileDescriptor* parameter refers to a directory, the value returned applies to filenames within the directory.
3. If the *Name* parameter has a value if **_PC_PATH_MAX**, and if the *Path* or *FileDescriptor* parameter refers to a directory that is the working directory, the value returned is the maximum length of a relative pathname.
4. If the *Name* parameter has a value of **_PC_PIPE_BU**F, and if the *Path* parameter refers to a FIFO special file or the *FileDescriptor* parameter refers to a pipe or a FIFO special file, the value returned applies to the referenced object. If the *Path* or *FileDescriptor* parameter refers to a directory, the value returned applies to any FIFO special file that exists or can be created within the directory.
5. If the *Name* parameter has a value of **_PC_CHOWN_RESTRICTED**, and if the *Path* or *FileDescriptor* parameter refers to a directory, the value returned applies to any files, other than directories, that exist or can be created within the directory.

Return Values

If the **pathconf** or **fpathconf** subroutine is successful, the specified parameter is returned. Otherwise, a value of –1 is returned and the **errno** global variable is set to indicate the error. If the variable corresponding to the *Name* parameter has no limit for the *Path* parameter or the *FileDescriptor* parameter, both the **pathconf** and **fpathconf** subroutines return a value of –1 without changing the **errno** global variable.

Error Codes

The **pathconf** or **fpathconf** subroutine fails if the following error occurs:

EINVAL	The name parameter specifies an unknown or inapplicable characteristic.

The **pathconf** subroutine can also fail if any of the following errors occur:

EACCES	Search permission is denied for a component of the path prefix.
EINVAL	The implementation does not support an association of the *Name* parameter with the specified file.
ENAMETOOLONG	The length of the *Path* parameter string exceeds the **PATH_MAX** value.
ENAMETOOLONG	Pathname resolution of a symbolic link produced an intermediate result whose length exceeds PATH_MAX.
ENOENT	The named file does not exist or the *Path* parameter points to an empty string.
ENOTDIR	A component of the path prefix is not a directory.
ELOOP	Too many symbolic links were encountered in resolving path.

The **fpathconf** subroutine can fail if either of the following errors occur:

EBADF	The *File Descriptor* parameter is not valid.
EINVAL	The implementation does not support an association of the *Name* parameter with the specified file.

Implementation Specifics

These subroutines are part of Base Operating System (BOS) Runtime.

Related Information

The **chown** subroutine, **confstr** subroutine, **sysconf** subroutine.

Files, Directories, and File Systems for Programmers, Subroutines Overview in *AIX General Programming Concepts : Writing and Debugging Programs*.

pause Subroutine

Purpose

Suspends a process until a signal is received.

Library

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

Syntax

```
#include <unistd.h>

int pause (void)
```

Description

The **pause** subroutine suspends the calling process until it receives a signal. The signal must not be one that is ignored by the calling process. The **pause** subroutine does not affect the action taken upon the receipt of a signal.

Return Values

If the signal received causes the calling process to end, the **pause** subroutine does not return.

If the signal is caught by the calling process and control is returned from the signal–catching function, the calling process resumes execution from the point of suspension. The **pause** subroutine returns a value of –1 and sets the **errno** global variable to **EINTR**.

Implementation Specifics

This subroutine is part of Base Operating System (BOS) Runtime.

Related Information

The **incinterval**, **alarm**, **or settimer** subroutine, **kill** or **killpg** subroutine, **sigaction**, **sigvec**, or **signal** subroutine, **wait**, **waitpid**, or **wait3** subroutine.

pclose Subroutine

Purpose

Closes a pipe to a process.

Library

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

Syntax

```
#include <stdio.h>
int pclose (Stream)
FILE *Stream;
```

Description

The **pclose** subroutine closes a pipe between the calling program and a shell command to be executed. Use the **pclose** subroutine to close any stream you opened with the **popen** subroutine. The **pclose** subroutine waits for the associated process to end, and then returns the exit status of the command.

> **Attention:** If the original processes and the **popen** process are reading or writing a common file, neither the **popen** subroutine nor the **pclose** subroutine should use buffered I/O. If they do, the results are unpredictable.

Avoid problems with an output filter by flushing the buffer with the **fflush** subroutine.

Parameter

Stream	Specifies the **FILE** pointer of an opened pipe.

Return Values

The **pclose** subroutine returns a value of –1 if the *Stream* parameter is not associated with a **popen** command or if the status of the child process could not be obtained. Otherwise, the value of the termination status of the command language interpreter is returned; this will be 127 if the command language interpreter cannot be executed.

Error Codes

If the application has called:

- The **wait** subroutine,
- The **waitpid** subroutine with a process ID less than or equal to zero or equal to the process ID of the command line interpreter, or
- Any other function that could perform one of the two steps above, and

one of these calls caused the termination status to be unavailable to the **pclose** subroutine, a value of –1 is returned and the **errno** global variable is set to **ECHILD**.

Implementation Specifics

This subroutine is part of Base Operating System (BOS) Runtime.

Related Information

The **fclose** or **fflush** subroutine, **fopen**, **freopen**, or **fdopen** subroutine, **pipe** subroutine, **popen** subroutine, **wait**, **waitpid**, or **wait3** subroutine.

Files, Directories, and File Systems for Programmers in *AIX General Programming Concepts : Writing and Debugging Programs*.

perror Subroutine

Purpose

Writes a message explaining a subroutine error.

Library

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

Syntax

```
#include <errno.h>

void perror (String)
const char *String;

extern int errno;
extern char *sys_errlist[ ];
extern int sys_nerr;
```

Description

The **perror** subroutine writes a message on the standard error output that describes the last error encountered by a system call or library subroutine. The error message includes the *String* parameter string followed by a : (colon), a space character, the message, and a new–line character. The *String* parameter string should include the name of the program that caused the error. The error number is taken from the **errno** global variable, which is set when an error occurs but is not cleared when a successful call to the **perror** subroutine is made.

To simplify various message formats, an array of message strings is provided in the **sys_errlist** structure or use the **errno** global variable as an index into the **sys_errlist** structure to get the message string without the new–line character. The largest message number provided in the table is **sys_nerr**. Be sure to check the **sys_nerr** structure because new error codes can be added to the system before they are added to the table.

The **perror** subroutine retrieves an error message based on the language of the current locale.

After successfully completing, and before a call to the **exit** or **abort** subroutine or the completion of the **fflush** or **fclose** subroutine on the standard error stream, the **perror** subroutine marks for update the `st_ctime` and `st_mtime` fields of the file associated with the standard error stream.

Parameter

String	Specifies a parameter string that contains the name of the program that caused the error. The ensuing printed message contains this string, a : (colon), and an explanation of the error.

Implementation Specifics

This subroutine is part of Base Operating System (BOS) Runtime.

Related Information

The **abort** subroutine, **exit** subroutine, **fflush** or **fclose** subroutine, **printf**, **fprintf**, **sprintf**, **wsprintf**, **vprintf**, **vfprintf**, **vsprintf**, or **vwsprintf** subroutine, **strerror** subroutine.

Subroutines Overview in *AIX General Programming Concepts : Writing and Debugging Programs*.

pipe Subroutine

Purpose

Creates an interprocess channel.

Library

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

Syntax

```
#include <unistd.h>

int pipe (FileDescriptor)
int FileDescriptor[2];
```

Description

The **pipe** subroutine creates an interprocess channel called a pipe and returns two file descriptors, *FileDescriptor***[**0**]** and *FileDescriptor***[**1**]**. *FileDescriptor***[**0**]** is opened for reading and *FileDescriptor***[**1**]** is opened for writing.

A read operation on the *FileDescriptor***[**0**]** parameter accesses the data written to the *FileDescriptor***[**1**]** parameter on a first–in, first–out (FIFO) basis.

Write requests of **PIPE_BUF** bytes or fewer will not be interleaved (mixed) with data from other processes doing writes on the same pipe. **PIPE_BUF** is a system variable described in the **pathconf** subroutine. Writes of greater than **PIPE_BUF** bytes may have data interleaved, on arbitrary boundaries, with other writes.

If **O_NONBLOCK** or **O_NDELAY** are set, writes requests of **PIPE_BUF** bytes or fewer will either succeed completely or fail and return –1 with the **errno** global variable set to **EAGAIN**. A write request for more than **PIPE_BUF** bytes will either transfer what it can and return the number of bytes actually written, or transfer no data and return –1 with the **errno** global variable set to **EAGAIN**.

Parameters

FileDescriptor	Specifies the address of an array of two integers into which the new file descriptors are placed.

Return Values

Upon successful completion, a value of 0 is returned. Otherwise, a value of –1 is returned, and the **errno** global variable is set to identify the error.

Error Codes

The **pipe** subroutine is unsuccessful if one or more the following are true:

EFAULT	The *FileDescriptor* parameter points to a location outside of the allocated address space of the process.
EMFILE	The number of open of file descriptors exceeds the **OPEN_MAX** value.
ENFILE	The system file table is full, or the device containing pipes has no free i–nodes.

Implementation Specifics

This subroutine is part of Base Operating System (BOS) Runtime.

Related Information

The **read** subroutine, **select** subroutine, **write** subroutine.

The **ksh** command, **sh** command.

Files, Directories, and File Systems for Programmers in *AIX General Programming Concepts : Writing and Debugging Programs*.

plock Subroutine

Purpose

Locks the process, text, or data in memory.

Library

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

Syntax

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

int plock (Operation)
int Operation;
```

Description

The **plock** subroutine allows the calling process to lock or unlock its text region (text lock), its data region (data lock), or both its text and data regions (process lock) into memory. The **plock** subroutine does not lock the shared text segment or any shared data segments. Locked segments are pinned in memory and are immune to all routine paging. Memory locked by a parent process is not inherited by the children after a **fork** subroutine call. Likewise, locked memory is unlocked if a process executes one of the **exec** subroutines. The calling process must have the root user authority to use this subroutine.

A real–time process can use this subroutine to ensure that its code, data, and stack are always resident in memory.

Note: Before calling the **plock** subroutine, the user application must lower the maximum stack limit value using the **ulimit** subroutine.

Parameters

Operation Specifies one of the following:

PROCLOCK	Locks text and data into memory (process lock).
TXTLOCK	Locks text into memory (text lock).
DATLOCK	Locks data into memory (data lock).
UNLOCK	Removes locks.

Return Values

Upon successful completion, a value of 0 is returned to the calling process. Otherwise, a value of –1 is returned and the **errno** global variable is set to indicate the error.

Error Codes

The **plock** subroutine is unsuccessful if one or more of the following is true:

EPERM	The effective user ID of the calling process does not have the root user authority.
EINVAL	The *Operation* parameter has a value other than **PROCLOCK**, **TXTLOCK**, **DATLOCK**, or **UNLOCK**.
EINVAL	The *Operation* parameter is equal to **PROCLOCK**, and a process lock, text lock, or data lock already exists on the calling process.
EINVAL	The *Operation* parameter is equal to **TXTLOCK**, and a text lock or process lock already exists on the calling process.

EINVAL	The *Operation* parameter is equal to **DATLOCK**, and a data lock or process lock already exists on the calling process.
EINVAL	The *Operation* parameter is equal to **UNLOCK**, and no type of lock exists on the calling process.

Implementation Specifics

This subroutine is part of Base Operating System (BOS) Runtime.

Related Information

The **exec** subroutines, **_exit**, **exit**, or **atexit** subroutine, **fork** subroutine, **ulimit** subroutine.

pm_battery_control Subroutine

Purpose

Controls and queries the battery status.

Library

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

Syntax

```
#include <sys/pm.h>
int pm_battery_control(Command, Battery);
int Command;
struct pm_battery *Battery;
```

Description

The **pm_battery_control** subroutine controls and queries the battery status.

Parameters

Command
Specifies one of the following:

PM_BATTERY_DISCHARGE
Discharges the battery.

PM_BATTERY_QUERY
Queries fuel state of the battery.

Battery
Points a following **pm_battery** structure to return battery information. When *Command* is **PM_BATTERY_QUERY**, the following structure is used:

```
struct pm_battery {
   int  attribute;/*battery attributes
are as follows*/
         PM_BATTERY  /* battery is
supported */
         PM_BATTERY_EXIST    /* battery
exists */
         PM_NICD      /*NiCd or NiMH */
         PM_CHARGE   /* now charging */
         PM_DISCHARGE   /* now
discharging */
         PM_AC       /* AC power is in
use */
         PM_DC       /* DC power is in
use */
   int capacity;   /* battery capacity */
   int remain;   /* current remaining
capacity */
   int discharge_remain;
         /*remaining capacity while
discharging */
   int  discharge_time;   /* discharge
time */
   int  full_charge_count; /*full charge
count */
}
If a field is not applicable, -1 is set.
```

Return Values

PM_SUCCESS	Indicates successful completion.
PM_ERROR	Indicates an error condition. The variable **errno** is set to identify the error.

Error Codes

EINVAL	The argument or command is not valid.

Implementation Specifics

The **pm_battery_control** subroutine is part of the Base Operating System (BOS) Runtime.

Related Information

The **pm_control_state** subroutine, **pm_control_parameter** subroutine.

pm_control_parameter Subroutine

Purpose

Controls and queries Power Management parameters.

Library

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

Syntax

```
#include <sys/pm.h>
int pm_control_parameter (control, argument)
int control;
caddr_t argument;
```

Description

The **pm_control_parameter** subroutine controls and queries Power Management parameters.

Parameters

control Specifies one of the following Power Management (PM) control commands:

PM_CTRL_QUERY_SYSTEM_IDLE_TIMER
Queries system idle timer.

PM_CTRL_SET_SYSTEM_IDLE_TIMER
Sets system idle timer.

PM_CTRL_QUERY_DEVICE_IDLE_TIMER
Queries device idle timer.

PM_CTRL_SET_DEVICE_IDLE_TIMER
Sets device idle timer.

PM_CTRL_QUERY_LID_CLOSE_ACTION
Queries the LID close action.

PM_CTRL_SET_LID_CLOSE_ACTION
Sets the LID close action.

PM_CTRL_QUERY_SYSTEM_IDLE_ACTION
Queries the system idle action.

PM_CTRL_SET_SYSTEM_IDLE_ACTION
Sets the system idle action.

PM_CTRL_QUERY_MAIN_SWITCH_ACTION
Queries the main power switch action.

PM_CTRL_SET_MAIN_SWITCH_ACTION
Sets the main power switch action.

PM_CTRL_QUERY_LOW_BATTERY_ACTION
Queries the low battery action.

PM_CTRL_SET_LOW_BATTERY_ACTION
Sets the low battery action.

PM_CTRL_QUERY_BEEP
Queries whether beep is enabled or not.

PM_CTRL_SET_BEEP
Enables/disables beep.

PM_CTRL_QUERY_PM_DD_NUMBER
Queries the number of PM aware DDs.

PM_CTRL_QUERY_PM_DD_LIST
Returns an array of devno of PM aware DDs.

PM_CTRL_QUERY_LID_STATE
Queries the LID state.

argument The value of the *argument* parameter depends on the Power Management control command.

For the following Power Management commands, the *argument* parameter is a pointer to an integer in which result value is stored:

- **PM_CTRL_QUERY_SYSTEM_IDLE_TIMER**
- **PM_CTRL_QUERY_LID_CLOSE_ACTION**
- **PM_CTRL_QUERY_SYSTEM_IDLE_ACTION**
- **PM_CTRL_QUERY_MAIN_SWITCH_ACTION**
- **PM_CTRL_QUERY_LOW_BATTERY_ACTION**
- **PM_CTRL_QUERY_BEEP**
- **PM_CTRL_QUERY_PM_DD_NUMBER**
- **PM_CTRL_QUERY_LID_STATE**

For the following Power Management commands, the *argument* parameter is an integer to be set.

- **PM_CTRL_SET_SYSTEM_IDLE_TIMER**
- **PM_CTRL_SET_LID_CLOSE_ACTION**
- **PM_CTRL_SET_SYSTEM_IDLE_ACTION**
- **PM_CTRL_SET_MAIN_SWITCH_ACTION**
- **PM_CTRL_SET_LOW_BATTERY_ACTION**
- **PM_CTRL_SET_BEEP**

For the **PM_CTRL_PM_QUERY_DEVICE_TIMER** and **PM_CTRL_SET_DEVICE_TIMER** commands, the *argument* parameter is a pointer to the following structure:

```
struct pm_device_timer_struct {
   dev_t   devno;   /* device major/minor number */
   int   mode;    /* device mode */
   int   device_idle_time;/* if -1, don't care */
   int   device_standby_time;/*if -1, don't care */
}
```

For the **PM_CTRL_QUERY_PM_DD_LIST** command, the *argument* parameter specifies a pointer to an array of integers.

Return Values

PM_SUCCESS	Indicates successful completion.
PM_ERROR	Indicates an error condition. The variable **errno** is set to identify the error.

Error Codes

EINVAL	The *argument* or *control* is not valid.

Implementation Specifics

The **pm_control_parameter** subroutine is part of the Base Operating System (BOS) Runtime.

Related Information

The **pm_control_state** subroutine, **pm_event_query** subroutine, **pm_battery_control** subroutine.

pm_control_parameter System Call

Purpose

Controls and queries the **PM** parameters.

Syntax

```
#include <pm.h>


int pm_control_parameter (ctrl, arg);
int ctrl;
caddr_t arg;
```

Description

The **pm_control_parameter** system call controls and queries the PM parameters.

Parameters

ctrl Specifies the function to be performed. It is one of the following values:

PM_CTRL_SET_PARAMETERS
Sets the **PM** parameters.

PM_CTRL_QUERY_DEVICE_NUMBER
Queries the number of **PM** aware devices.

PM_CTRL_QUERY_DEVICE_LIST
Gets all of **PM** aware device information.

PM_CTRL_QUERY_DEVICE_INFO
Queries **PM** aware device information.

PM_CTRL_SET_DEVICE_INFO
Sets **PM** aware device information.

PM_CTRL_SET_HIBERNATION_VOLUME
Tells **PM** hibernation volume information to **PM** core.

arg Specifies a pointer to a structure that depends on the function specified by the *ctrl* parameter.

When the *ctrl* parameter is **PM_CTRL_SET_PARAMETERS**, *arg* is a pointer to the following **pm_parameters_t** structure:

```
typedef struct _pm_parameters {
   Simple_lock lock; /*lock data to serialize
access*/
   core_data_t core_data;
   daemon_data_t daemon_data;
} pm_parameters_t;
```

where,

```
typedef struct _daemon_data{
   int system_idle_action; /*system idle action*/
   int lid_close_action;   /*lid close action*/
   int main_switch_action; /*main power switch
action*/
   int low_battery_action; /*low battery action*/
   int specified_time_action; /*action at specified
time*/
   int resume_passwd;      /*enable/disable resume
password*/
   int kill_lft_session;   /*continue/kill LFT
session*/
   int kill_tty_session;   /*continue/kill TTY
session*/
   int permission;         /*permitted state by
superuser*/
} daemon_data_t;
typedef struct _core_data{
   int system_idle_time;   /*system idle time in
seconds*/
   int pm_beep;            /*enable/disable beep*/
   int ringing_resume;     /*enable/disable ringing
resume*/
   time_t resume_time;     /*specified time to
resume*/
   time_t specified_time;  /*specified time to sus
or hiber*/
   int sus_to_hiber;       /*duration from suspend
to hibernation*/
   int kill_syncd;         /*if syncd has been
killed*/
   char reserve[4];        /*reserved*/
} core_data_t;
```

When the *ctrl* parameter is **PM_CTRL_QUERY_DEVICE_NUMBER**, *arg* is a pointer to an integer where the number of PM aware device drivers is returned.

When the *ctrl* parameter is **PM_CTRL_QUERY_DEVICE_LIST**, *arg* is a pointer to an array of device logical names.

When the *ctrl* parameter is **PM_CTRL_QUERY_DEVICE_INFO**, or **PM_CTRL_SET_DEVICE_INFO**, *arg* is a pointer to the following **pm_device_info_t** structure:

```
struct _pm_device_info {
   char name[16];           /*device logical name*/
   int mode;                /*current device PM mode*/
   int attribute;           /*PM attribute*/
   int idle_time;           /*device idle time*/
   int standby_time;        /*device standby time*/
   int idle_time1;          /*idle time 1 for DPMS */
   int idle_time2;          /*idle time 2 for DPMS */
   char reserve[24];        /*reserved*/
} pm_device_info_t;
```

When the *ctrl* parameter is **PM_CTRL_SET_HIBERNATION_VOLUME**, *arg* is a pointer to the following pm_hibernation_t structure:

```
typedef struct _pm_hibernation {
   dev_t devno;             /*major/minor device number*/
   long ppnum;              /*physical partition number*/
   long ppsize;             /*physical partition size*/
   long snum;               /*valid sector list item number*/
   pm_sector_t *slist;      /*sector list*/
 pm_hibernation_t;
```

Where,

```
typedef struct _pm_sector {
   long start;              /*RBA(Relative Block Address) in
sectors*/
   long size;               /*sector size in sectors(512 bytes)*/
} pm_sector_t;
```

Note: The functions in AIX 4.1.1 still remain as they were. But they are left only for backward compatibility and may be deleted in the future. New programs should not use them.

Return Values

PM_SUCCESS	Indicates successful completion.
PM_ERROR	Indicates an error condition. The variable errno is set to identify the error.

Error Codes

EINVAL	Invalid argument.

Implementation Specifics

The **pm_control_parameter** system call is part of the Base Operation System (BOS) Runtime.

Related Information

The **pm_battery_control** subroutine.

The **pm_control_state** system call, **pm_system_event_query** system call.

pm_control_state Subroutine

Purpose

Controls and queries the Power Management states

Library

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

Syntax

```
#include <sys/pm.h>
int pm_control_state(control, PMS)
int control;
struct pm_state *PMS;
```

Description

The **pm_control_state** subroutine controls and queries the Power Management (PM) states.

Parameters

control Specifies one of the following Power Management control commands:

PM_CTRL_QUERY_STATE	Queries the current system PM state.
PM_CTRL_REQUEST_STATE	Requests to move to system full–on, system PM enable, system standby or system suspend state.
PM_CTRL_START_STATE	Forces to move to system full–on, system PM enable, system standby or system suspend state.
PM_CTRL_QUERY_REQUEST	Queries the result of the requested action.

PMS Specifies a pointer to the following **pm_state** structure.

```
struct pm_state {
   int   state;
   int   id;
   int   event;
   int   devno;
}
```

The contents of the structure depends on the PM control command.

- When the *control* is **PM_CTRL_QUERY_STATE**, state is returned.
- When the *control* is **PM_CTRL_REQUEST_STATE**, input is state and output is id.
- When the *control* is **PM_CTRL_START_STATE**, input is state and output is event and devno (if event is **PM_EVENT_ERROR)**.
- When the *control* is **PM_CTRL_QUERY_REQUEST**, input is id and output is event and devno (if event is **PM_EVENT_ERROR**).

Event value is one of the following,

PM_EVENT_LID_OPEN	LID open
PM_EVENT_RTC	specified time to resume
PM_EVENT_RINGING	ringing
PM_EVENT_MOUSE	mouse event
PM_EVENT_KEYBOARD	keyboard event
PM_EVENT_EXTRA_INPUTDD	extra input DD
PM_EVENT_EXTRA_BUTTON	extra button
PM_EVENT_ERROR	action failed

Return Values

PM_SUCCESS	Indicates successful completion.
PM_ERROR	Indicates an error condition. The variable **errno** is set to identify the error.

Error Codes

EINVAL	The argument or command is not valid.

Implementation Specifics

The **pm_control_state** subroutine is part of the Base Operating System (BOS) Runtime.

Related Information

The **pm_control_parameter** subroutine, **pm_event_query** subroutine, **pm_battery_control** subroutine.

pm_control_state System Call

Purpose

Controls and queries the **PM** state.

Syntax

```
#include <pm.h>



int
pm_control_state (ctrl, arg);
int ctrl;
caddr_t arg;
```

Parameters

ctrl Specifies the function to be performed. It is one of the following values:

PM_CTRL_QUERY_SYSTEM_STATE
Queries the **PM** state.

PM_CTRL_START_SYSTEM_STATE
Initiates the **PM** state change.

arg Specifies a pointer to the following **pm_system_state_t** structure:

```
struct _pm_system_state {
   int state;              /*system PM
state*/
   int event;              /*resume
event*/
   char name[16];          /*device name
which caused an error*/
   char reserve[8];        /*reserved*/
} pm_system_state_t;
```

The state value is one of the following:

PM_SYSTEM_FULL_ON	System full on
PM_SYSTEM_ENABLE	System PM enable
PM_SYSTEM_STANDBY	System standby
PM_SYSTEM_SUSPEND	System suspend
PM_SYSTEM_HIBERNATION	System hibernation
PM_TRANSITION_START	Transition request started
PM_TRANSITION_END	Transition request completed

The event value is one of the following:

PM_EVENT_POWER_SWITCH_ON

PM_EVENT_LID_OPEN

PM_EVENT_RTC

PM_EVENT_RING

PM_EVENT_MOUSE

PM_EVENT_KEYBOARD

PM_EVENT_EXTRA_INPUTDD

PM_EVENT_EXTRA_BUTTON

PM_EVENT_REJECT_BY_HIB_VOL

PM_EVENT_NOT_SUPPORTED

PM_EVENT_GENERAL_ERROR

PM_EVENT_REJECT_BY_DD

Note: The functions at AIX 4.1.1 still remain as they were. But they are left only for backward compatibility and may be deleted in the future. New programs should not use them.

Description

The **pm_control_state** system call controls and queries the **PM** state.

Return Values

PM_SUCCESS	Indicates successful completion.
PM_ERROR	Indicates an error condition. The variable errno is set to identify the error.

Error Codes

EINVAL	Invalid argument.

Implementation Specifics

The **pm_control_state** system call is part of the Base Operation System (BOS) Runtime.

Related Information

The **pm_battery_control** subroutine.

The **pm_control_parameter** system call, **pm_system_event_query** system call.

pm_event_query Subroutine

Purpose

Queries a Power Management Event.

Library

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

Syntax

```
#include <sys/pm.h>
int pm_event_query(Event, Action);
int *Event;
int *Action;
```

Description

The **pm_event_query** subroutine queries a Power Management (PM) event.

Parameters

Event Returns one of the following events:

PM_EVENT_NONE	no event
PM_EVENT_LID_CLOSE	LID close
PM_EVENT_SYSTEM_IDLE_TIMER	system timer expiration
PM_EVENT_LOW_BATTERY	low battery
PM_EVENT_SOFTWARE_REQUEST	requested by software
PM_EVENT_DATA_CHANGE	PM data change notice
PM_EVENT_AC	power change from DC to AC
PM_EVENT_DC	power change from AC to DC
PM_EVENT_DISPLAY_MESSAGE	display message request
PM_EVENT_SPECIFIED_TIME	Specified time for suspend/hibernation

Action Returns one of the following actions (system state) to be requested. It is a default state transition action in PM core:

PM_SYSTEM_NONE

PM_SYSTEM_FULL_ON

PM_SYSTEM_ENABLE

PM_SYSTEM_STANDBY

PM_SYSTEM_SUSPEND

PM_SYSTEM_SHUTDOWN

Return Values

PM_SUCCESS Indicates successful completion.

PM_ERROR Indicates an error condition. The variable **errno** is set to identify the error.

Error Codes

EINVAL	The argument or command is not valid.
EBUSY	Another process is blocked for query.

Implementation Specifics

The **pm_event_query** subroutine is part of the Base Operating System (BOS) Runtime.

Related Information

The **pm_control_state** subroutine, **pm_control_parameter** subroutine, **pm_battery_control** subroutine.

pm_system_event_query System Call

Purpose

Controls and queries the **PM** event.

Syntax

```
#include <pm.h>

int pm_system_event_query (event);
int event;
```

Description

The **pm_system_event_query** system call queries the **PM** event.

event Returns one of the following events:

PM_EVENT_NONE

PM_EVENT_LID_OPEN

PM_EVENT_LID_CLOSE

PM_EVENT_LOW_BATTERY

PM_EVENT_SYSTEM_IDLE_TIMER

PM_EVENT_POWER_SWITCH_OFF

PM_EVENT_POWER_SWITCH_ON

PM_EVENT_SPECIFIED_TIME

PM_EVENT_MOUSE

PM_EVENT_KEYBOARD

PM_EVENT_EXTRA_INPUTDD

PM_EVENT_EXTRA_BUTTON

PM_EVENT_TERMINATE

PM_EVENT_AC

PM_EVENT_DC

PM_SUCCESS Indicates successful completion.

PM_ERROR Indicates an error condition. The variable errno is set to identify the error.

PM library

The PM library is supported to control/query **PM** information from application programs.

Error Codes

EINVAL Invalid argument.

Implementation Specifics

The **pm_system_event_query** system call is part of the Base Operation System (BOS) Runtime.

Related Information

The **pm_control_parameter** system call, **pm_battery_control** subroutine, **pm_control_state** system call.

pmlib_get_event_notice Subroutine

Purpose

Gets a new **PM** event.

Library

PM (Power Management) Library (**libpm.a**)

Syntax

```
#include <pmlib.h>

int pmlib_get_event_notice(event)
int *event;
```

Description

The **pmlib_get_event_notice** subroutine gets the latest event. It is recommended **PM**–aware application calls this subroutine when signal notification from **pm** daemon arrives.

Parameters

event Points to an integer that is the latest **PM** event that the **PM** daemon holds, *event* can be bit–wise OR of following values:

PMLIB_EVENT_NONE	No event.
PMLIB_EVENT_AC	Power source is changed to AC.
PMLIB_EVENT_DC	Power source is changed to DC.
PMLIB_EVENT_NOTICE_TO_FULL_ON	System will change state to full–on.
PMLIB_EVENT_NOTICE_TO_STANDBY	System will change state to standby.
PMLIB_EVENT_NOTICE_TO_SUSPEND	System will change state to suspend.
PMLIB_EVENT_NOTICE_TO_ENABLE	System will change state to **PM** enable.
PMLIB_EVENT_NOTICE_TO_HIBERNATION	System will change state to hibernation.
PMLIB_EVENT_NOTICE_TO_SHUTDOWN	System will change state to shutdown.
PMLIB_EVENT_NOTICE_TO_TERMINATE	**PM** will be unconfigured.
PMLIB_EVENT_NOTICE_OF_REJECTION	State change request was rejected.
PMLIB_EVENT_NOTICE_COMPLETION	State change was completed.
PMLIB_EVENT_RESUME_FROM_STANDBY	System is resumed from standby.
PMLIB_EVENT_RESUME_FROM_SUSPEND	System is resumed from suspend.
PMLIB_EVENT_RESUME_FROM_HIBERNATION	System is resumed from hibernation.
PMLIB_EVENT_START_TO_CHANGE_STATE	System state change started.
PMLIB_EVENT_FORCE_TO_CHANGE_STATE	System is forced to change state.
PMLIB_EVENT_FAIL_TO_CHANGE_STATE	System state change failed.

Return Values

Upon successful completion, **PMLIB_SUCCESS** is returned. If the **pmlib_get_event_notice** subroutine fails, **PMLIB_ERROR** is returned and errno variable is set to an error code.

Error Codes

ESRCH	**PM** daemon is not running.
EINVAL	Invalid argument.

Note: If an application program is registered as **PM** aware, the **PM** daemon sends a SIGPM (equal to SIGPWR) signal to the application when an **PM** event occurs. The application program needs to prepare a signal handler and to use this **pmlib_get_event_notice** subroutine to get the to get the **PM** event.

Implementation Specifics

The **pmlib_get_event_notice** subroutine is part of the Base Operation System (BOS) Runtime.

Related Information

The **pmlib_request_state** subroutine, **pmlib_request_battery** subroutine, **pmlib_request_parameter** subroutine, **pmlib_register_application** subroutine.

pmlib_register_application Subroutine

Purpose

Registers or unregister a **PM** aware application

Library

PM (Power Management) Library (**libpm.a**)

Syntax

```
#include <pmlib.h>

int pmlib_register_application(cmd);
int cmd;
```

Parameters

cmd	Determines the action to be taken by the **pmlib_register_application** subroutine and is one of the following values:
	PMLIB_REGISTER Registers an application.
	PMLIB_UNREGISTER Unregisters an application.

Description

The **pmlib_register_application** registers or unregisters the caller process as a **PM**–aware application. The **pmlib_register_application** subroutine can be called by any user.

Return Values

Upon successful completion, **PMLIB_SUCCESS** is returned. If the **pmlib_request_state** subroutine fails, **PMLIB_ERROR** is returned and errno variable is set to an error code.

Error Codes

ESRCH	**PM** daemon is not running.
EINVAL	Invalid argument.

Implementation Specifics

The **pmlib_register_application** subroutine is part of the Base Operation System (BOS) Runtime.

Related Information

The **pmlib_get_event_notice** subroutine, **pmlib_request_state** subroutine, **pmlib_request_battery** subroutine, **pmlib_request_parameter** subroutine.

pmlib_request_battery Subroutine

Purpose

Queries and controls the battery status.

Library

PM (Power Management) Library (**libpm.a**)

Syntax

```
#include <pmlib.h>

int pmlib_request_battery (cmd, pmb);
int cmd;
pmlib_battery_t *pmb;
```

Parameters

cmd Determines the action to be taken by the **pmlib_request_battery** subroutine and isone of the following values:

PMLIB_QUERY_BATTERY	Queries the battery state.
PMLIB_DISCHARGE_BATTERY	Discharges the battery.

pmb Points to the following **pmlib_battery_t** structure:

```
typedef struct _pmlib_battery {
   int attribute;          /*battery attribute*/
   int capacity;           /*battery capacity*/
   int remain;             /*current remain capacity*/
   int refresh_discharge_capacity;
   int refresh_discharge_time; /*discharge time*/
   int full_change_count;
} pmlib_battery_t;
```

When cmd is **PMLIB_QUERY_BATTERY**, the returned **pmb.attribute** is bit–wise OR of following values:

PMLIB_BATTERY_SUPPORTED	Battery is supported.
PMLIB_BATTERY_EXIST	Battery exists.
PMLIB_BATTERY_NICD	Battery is NiCd
PMLIB_BATTERY_CHARGING	Battery is being charged.
PMLIB_BATTERY_DISCHARGING	Battery is being discharged.
PMLIB_BATTERY_AC	AC adapter is in use.
PMLIB_BATTERY_DC	Battery is in use.
PMLIB_BATTERY_REFRESH_REQ	Need to refresh battery.

Description

The **pmlib_request_battery** subroutine queries the battery information or requests to discharge the battery. The **pmlib_request_** subroutine can be called by any user.

Return Values

Upon successful completion, **PMLIB_SUCCESS** is returned. If the **pmlib_request_state** subroutine fails, **PMLIB_ERROR** is returned and errno variable is set to an error code.

Error Codes

ESRCH	**PM** daemon is not running.
EINVAL	Invalid argument.

Implementation Specifics

The **pmlib_request_battery** subroutine is part of the Base Operation System (BOS) Runtime.

Related Information

The **pmlib_get_event_notice** subroutine, **pmlib_request_state** subroutine, **pmlib_request_parameter** subroutine, **pmlib_register_application** subroutine.

pmlib_request_parameter Subroutine

Purpose

Queries and controls the **PM** system parameters.

Library

PM (Power Management) Library (**libpm.a**)

Syntax

```
#include <pmlib.h>

int pmlib_request_parameter(ctrl, arg);
int ctrl;
caddr_t *arg;
```

The **pmlib_request_parameter** subroutines queries and changes the **PM** system or devices parameters. Any of these queries can be called by any user, but the set can be called only by root.

Parameters

ctrl Determines the action to be taken by the **pmlib_request_parameter** subroutine and is one of the following values:

PMLIB_QUERY_SYSTEM_IDLE_TIME	Queries system idle timer.
PMLIB_SET_SYSTEM_IDLE_TIME	Sets system idle timer.
PMLIB_QUERY_DEVICE_INFO	Queries device information.
PMLIB_SET_DEVICE_INFO	Sets device information.
PMLIB_QUERY_SYSTEM_IDLE_ACTION	Queries action for system idle.
PMLIB_SET_SYSTEM_IDLE_ACTION	Sets action for system idle.
PMLIB_QUERY_LID_CLOSE_ACTION	Queries action for lid close.
PMLIB_SET_LID_CLOSE_ACTION	Sets action for lid close.
PMLIB_QUERY_MAIN_SWITCH_ACTION	Queries action for main switch.
PMLIB_SET_MAIN_SWITCH_ACTION	Sets action for main switch.
PMLIB_QUERY_LOW_BATTERY_ACTION	Queries action for low battery.

PMLIB_SET_LOW_BATTERY_ACTION	Sets action for low battery.
PMLIB_QUERY_PERMISSION	Queries the action permitted to any user.
PMLIB_SET_PERMISSION	Sets the action permitted to any user.
PMLIB_QUERY_BEEP	Queries whether beep is enabled or not.
PMLIB_SET_BEEP	Sets whether beep is enabled or not.
PMLIB_QUERY_RINGING_RESUME	Queries if ringing resume is enabled or not.
PMLIB_SET_RINGING_RESUME	Sets if ringing resume is enabled or not.
PMLIB_QUERY_RESUME_TIME	Queries resume time.
PMLIB_SET_RESUME_TIME	Sets resume time.
PMLIB_QUERY_DURATION_TO_HIBERNATION	Queries duration to hibernation.
PMLIB_SET_DURATION_TO_HIBERNATION	Sets duration to hibernation.
PMLIB_QUERY_SPECIFIED_TIME	Queries specified time.
PMLIB_SET_SPECIFIED_TIME	Sets specified time.
PMLIB_QUERY_SPECIFIED_TIME_ACTION	Queries action for specified time.
PMLIB_SET_SPECIFIED_TIME_ACTION	Sets action for specified time.
PMLIB_QUERY_DEVICE_NUMBER	Queries number of **PM** aware devices.
PMLIB_QUERY_DEVICE_NAMES	Queries the list of names of **PM** aware devices.
PMLIB_QUERY_SUPPORTED_STATES	Queries the system states supported.
PMLIB_QUERY_KILL_LFT_SESSION	Queries if LFT session is terminated.
PMLIB_SET_KILL_LFT_SESSION	Sets if LFT session is terminated.
PMLIB_QUERY_KILL_TTY_SESSION	Queries if TTY session is terminated.
PMLIB_SET_KILL_TTY_SESSION	Sets if TTY session is terminated.
PMLIB_QUERY_PASSWD_ON_RESUME	Queries if resume password is required.
PMLIB_SET_PASSWD_ON_RESUME	Sets if resume password is required.
PMLIB_QUERY_KILL_SYNCD	Queries if syncd is terminated at standby.
PMLIB_SET_KILL_SYNCD	Sets if syncd is terminated at system standby.

When *ctrl* is **PMLIB_QUERY_SYSTEM_IDLE_TIMER**, *arg* points to an integer that is the system idle timer value when function is returned.

When *ctrl* is **PMLIB_SET_SYSTEM_IDLE_TIMER**, *arg* points to an integer that is the system idle timer value to set. The system idle timer value should be within the range from 60 to 7200. If 0 is set, the system–idle check is disabled.

When *ctrl* is **PMLIB_QUERY_DEVICE_INFO**, *arg* points to the following **pmlib_device_info_t** structure. The device name needs to be specified in the name[16] member. Then, **mode**, **idle_time**, **standby_time**, **idle_time1** and **idle_time2** members are set when function is returned.

```
typedef struct _pmlib_device_info{
   char name[16];       /*name of device*/
   int mode;            /*device mode*/
   int attribute;       /*device attribute*/
   int idle_time;       /*idle timer value during system PM
enable*/
   int standby_time;    /*idle timer value during system standby*/
   int idle_time1;      /*idle timer value for DPMS suspend*/
   int idle_time2;      /*idle timer value for DPMS off*/
   char reserved[24];   /*reserved area for future use*/
} pmlib_device_info_t;
```

When *ctrl* is **PMLIB_SET_DEVICE_INFO**, *arg* points to **pmlib_device_info_t** structure, and the devicename to name[16]. **idle_time**, **standby_time**, **idle_time1** and **idle_time2** members need to be set. The value of **idle_time** is within the range from 60 to 7200. If **idle_time**, **standby_time**, **idle_time1**, or **idle_time2** is set to –1, the current value is not changed.

When *ctrl* is **PMLIB_QUERY_SYSTEM_IDLE_ACTION**, *arg* points to an integer that is the action for system–idle timer expiration when the function is returned. Possible values for action are as follows:

PMLIB_NONE	State doesn't change
PMLIB_SYSTEM_FULL_ON	Full on
PMLIB_SYSTEM_ENABLE	PM enable
PMLIB_SYSTEM_STANDBY	Stand by
PMLIB_SYSTEM_SUSPEND	Suspend
PMLIB_SYSTEM_HIBERNATION	Hibernation
PMLIB_SYSTEM_SHUTDOWN	Shutdown

When *ctrl* is **PMLIB_SET_SYSTEM_IDLE_ACTION**, *arg* points to an integer that is the action to be set for system–idle timer expiration. The value for action should be one of the values described at **PMLIB_QUERY_SYSTEM_IDLE_ACTION**.

When *ctrl* is **PMLIB_QUERY_LID_CLOSE_ACTION**, *arg* points to an integer that is the action for lid close when the function is returned. Possible values for action are one of the values described at **PMLIB_QUERY_SYSTEM_IDLE_ACTION**.

When *ctrl* is **PMLIB_SET_LID_CLOSE_ACTION**, *arg* points to an integer that is the action to be set for lid close. The value for action should be one of the values described at **PMLIB_QUERY_SYSTEM_IDLE_ACTION**.

When *ctrl* is **PMLIB_QUERY_MAIN_SWITCH_ACTION**, *arg* points to an integer that is the action for the main switch when the function is returned. Possible values for action are one of the values described at **PMLIB_QUERY_SYSTEM_IDLE_ACTION**.

When *ctrl* is **PMLIB_SET_MAIN_SWITCH_ACTION**, *arg* points to an integer that is the action to set for the main switch. The value for action should be one of the values described at **PMLIB_QUERY_SYSTEM_IDLE_ACTION**.

When *ctrl* is **PMLIB_QUERY_LOW_BATTERY_ACTION**, *arg* points to an integer that is the action for the low battery when the function is returned. Possible values for action are one of the values described at **PMLIB_QUERY_SYSTEM_IDLE_ACTION**.

When *ctrl* is **PMLIB_SET_LOW_BATTERY_ACTION**, *arg* points to an integer that is the action to be set for the low battery. The value for action should be one of the values described at **PMLIB_QUERY_SYSTEM_IDLE_ACTION**.

When *ctrl* is **PMLIB_QUERY_PERMISSION**, *arg* points to an integer that is the action for allowed to any user when the function is returned. Possible values for action are bit–wise OR of following values:

PMLIB_SYSTEM_FULL_ON	Full on
PMLIB_SYSTEM_ENABLE	PM enable
PMLIB_SYSTEM_STANDBY	Standby
PMLIB_SYSTEM_SUSPEND	Suspend
PMLIB_SYSTEM_HIBERNATION	Hibernation
PMLIB_SYSTEM_SHUTDOWN	Shutdown

When *ctrl* is **PMLIB_SET_PERMISSION**, *arg* points to an integer that is the action to be set for allowed to any user. Possible values for action are bit–wise OR of following values:

PMLIB_SYSTEM_FULL_ON	Full on
PMLIB_SYSTEM_ENABLE	PM enable
PMLIB_SYSTEM_STANDBY	Standby
PMLIB_SYSTEM_SUSPEND	Suspend
PMLIB_SYSTEM_HIBERNATION	Hibernation
PMLIB_SYSTEM_SHUTDOWN	Shutdown

When *ctrl* is **PMLIB_QUERY_BEEP**, *arg* points to an integer that is **PMLIB_ON** or **PMLIB_OFF** for beep on/off.

When *ctrl* is **PMLIB_SET_BEEP**, *arg* points to an integer that is **PMLIB_ON** or **PMLIB_OFF** to set for beep on/off.

When *ctrl* is **PMLIB_QUERY_RINGING_RESUME**, *arg* points to an integer that is **PMLIB_ON** or **PMLIB_OFF** for ringing resume on/off when the function is returned.

When *ctrl* is **PMLIB_SET_RINGING_RESUME**, *arg* points to an integer that is **PMLIB_ON** or **PMLIB_OFF** for ringing resume on/off.

When *ctrl* is **PMLIB_QUERY_RESUME_TIME**, *arg* points to an integer that is the time for resume when the function is returned. The integer value is the time in seconds since 00:00:00 GMT, January 1, 1970.

When *ctrl* is **PMLIB_SET_RESUME_TIME**, *arg* points to an integer that is the time for resume to be set. The integer value should be the time in seconds since 00:00:00 GMT, January 1, 1970.

When *ctrl* is **PMLIB_QUERY_DURATION_TO_HIBERNATION**, *arg* points to an integer that is the duration to hibernation in seconds when the function is returned.

When *ctrl* is **PMLIB_SET_DURATION_TO_HIBERNATION**, *arg* points to an integer that is the duration to hibernation in seconds to be set.

When *ctrl* is **PMLIB_QUERY_SPECIFIED_TIME**, *arg* points to an integer that is the specified time when the function is returned. The integer value is the time in seconds since 00:00:00 GMT, January 1, 1970.

When *ctrl* is **PMLIB_SET_SPECIFIED_TIME**, *arg* points to an integer that is the specified time to be set. The integer value should be the time in seconds since 00:00:00 GMT, January 1, 1970.

When *ctrl* is **PMLIB_QUERY_SPECIFIED_TIME_ACTION**, *arg* points to an integer that is the action for the specified time when the function is returned. Possible values for action is one of the values described at **PMLIB_QUERY_SYSTEM_IDLE_ACTION**.

When *ctrl* is **PMLIB_SET_SPECIFIED_TIME_ACTION**, *arg* points to an integer that is the action to be set for the specified time. The value for action should be one of the values described at **PMLIB_QUERY_SYSTEM_IDLE_ACTION**.

When *ctrl* is **PMLIB_QUERY_DEVICE_NUMBER**, *arg* points to an integer that is the number of **PM** aware device drivers when the function is returned.

When *ctrl* is **PMLIB_QUERY_DEVICE_NAMES**, *arg* points to the head of array of name[16] that is the names of **PM** aware device drivers when the function is returned.

When *ctrl* is **PMLIB_QUERY_SUPPORTED_STATES**, *arg* points to an integer that is the action supported when the function is returned. The integer is bit–wise OR of following values:

PMLIB_SYSTEM_FULL_ON	Full on
PMLIB_SYSTEM_ENABLE	PM enable
PMLIB_SYSTEM_STANDBY	Standby
PMLIB_SYSTEM_SUSPEND	Suspend
PMLIB_SYSTEM_HIBERNATION	Hibernation
PMLIB_SYSTEM_SHUTDOWN	Shutdown

When *ctrl* is **PMLIB_QUERY_KILL_LFT_SESSION**, *arg* points to an integer that is **PMLIB_ON** or **PMLIB_OFF** to show if **LFT** session is terminated.

When *ctrl* is **PMLIB_SET_KILL_LFT_SESSION**, *arg* points to an integer that is **PMLIB_ON** or **PMLIB_OFF** to set if **LFT** session is terminated.

When *ctrl* is **PMLIB_QUERY_KILL_TTY_SESSION**, *arg* points to an integer that is **PMLIB_ON** or **PMLIB_OFF** to show if **TTY** sessions are terminated.

When *ctrl* is **PMLIB_SET_KILL_TTY_SESSION**, *arg* points to an integer that is **PMLIB_ON** or **PMLIB_OFF** to set if **TTY** sessions are terminated.

When *ctrl* is **PMLIB_QUERY_PASSWD_ON_RESUME**, *arg* points to an integer that is **PMLIB_ON** or **PMLIB_OFF** to show if resume password is required.

When *ctrl* is **PMLIB_SET_PASSWD_ON_RESUME**, *arg* points to an integer that is **PMLIB_ON** or **PMLIB_OFF** to set if resume password is required.

When *ctrl* is **PMLIB_QUERY_KILL_SYNCD**, *arg* points to an integer that is **PMLIB_ON** or **PMLIB_OFF** to show if *sync* daemon is terminated during system standby.

When *ctrl* is **PMLIB_SET_KILL_SYNCD**, *arg* points to an integer that is **PMLIB_ON** or **PMLIB_OFF** to set if *sync* daemon is terminated during system standby.

Return Values

Upon successful completion, **PMLIB_SUCCESS** is returned. If the **pmlib_request_state** subroutine fails, **PMLIB_ERROR** is returned and errno variable is set to an error code.

Error Codes

ESRCH	**PM** daemon is not running.
EINVAL	Invalid argument.

EPERM	Missing privilege.
ENOMEM	Insufficient storage.

Implementation Specifics

The **pmlib_request_parameter** subroutine is part of the Base Operation System (BOS) Runtime.

Related Information

The **pmlib_get_event_notice** subroutine, **pmlib_request_state** subroutine, **pmlib_request_battery** subroutine, **pmlib_register_application** subroutine.

pmlib_request_state Subroutine

Purpose

Requests system state change.

Library

PM (Power Management) Library (**libpm.a**)

Syntax

```
#include <pmlib.h>

int pmlib_request_state (ctrl, pms);
int ctrl;
pmlib_state_t *pms;
```

Parameters

ctrl Determines the action to be taken by the **pmlib_request_state** subroutine and is one of the following values:

PMLIB_REQUEST_STATE	Requests to change system state.
PMLIB_QUERY_STATE	Requests to query system state.
PMLIB_QUERY_ERROR	Requests to query error of system state change.
PMLIB_CONFIRM	Confirms system state change.

pms Points to the following **pmlib_state_t** structure:

```
typedef struct _pmlib_state {
   int state;      /*system state for set/query*/
   int error;      /*error value for query error*/
   pid_t pid;      /*process id of application which
                    prevented the state change*/
   char name[16];  /*name of application or PM aware DD
                    which prevented the state change*/
} pmlib_state_t;
```

When *ctrl* is **PMLIB_REQUEST_STATE**, set one of the following state values to **pms.state**:

PMLIB_SYSTEM_FULL_ON	Full on
PMLIB_SYSTEM_ENABLE	PM enable
PMLIB_SYSTEM_STANDBY	Standby
PMLIB_SYSTEM_SUSPEND	Suspend
PMLIB_SYSTEM_HIBERNATION	Hibernation
PMLIB_SYSTEM_SHUTDOWN	Shutdown

When *ctrl* is **PMLIB_QUERY_STATE**, one of state values described at **PMLIB_REQUEST_STATE** is set to **pms.state** when the function returns. **PM** aware DD's name is also returned if it rejects the **PM** command.

When *ctrl* is **PMLIB_QUERY_ERROR**, one of the following error values are set to **pms.error**:

PMLIB_NO_ERROR	No error.
PMLIB_ERROR_REJECT_BY_DEVICE	Device rejected to change state.
PMLIB_ERROR_REJECT_BY_APPL	Application rejected to change state.

PMLIB_ERROR_REJECT_BY_SYSTEM	System does not allow to change state.
PMLIB_ERROR_REJECTED_BY_HIB_VOL	Hibernation volume is invalid.
PMLIB_ERROR_REJECTED_BY_EVENT	A event prevented the state change.
PMLIB_ERROR_INVALID_PRIVILEGE	Caller was not allowed to change state.
PMLIB_ERROR_DEVICE_ERROR	A device rejected to change mode.
PMLIB_ERROR_OTHERS	Other error occurred.

If an application caused system state change failure, the process id of that application is set to **pms.pid**, and the name set to **pms.name** when the function returns.

When *ctrl* is **PMLIB_CONFIRM**, set one of the following state values to **pms.state**.

PMLIB_SYSTEM_CHANGE_OK	OK to change the system state.
PMLIB_SYSTEM_CHANGE_NO	No change to the system state.

Description

The **pmlib_request_state** subroutine is called to request to change **PM** system state, request to query **PM** system state, request to query the error of **PM** system state change, or request to confirm **PM** system state change. Non–root user can request to change state only if the specified action an the action within the range allowed to any user.

Return Values

Upon successful completion, **PMLIB_SUCCESS** is returned. If the **pmlib_request_state** subroutine fails, **PMLIB_ERROR** is returned and the errno variable is set to an error code.

Error Codes

ESRCH	**PM** daemon is not running.
EINVAL	Invalid argument.
EPERM	Missing privilege.
EBUSY	State change processing has already been started.

Implementation Specifics

The **pmlib_request_state** subroutine is part of the Base Operation System (BOS) Runtime.

Related Information

The **pmlib_get_event_notice** subroutine, **pmlib_request_battery** subroutine, **pmlib_request_parameter** subroutine, **pmlib_register_application** subroutine.

poll Subroutine

Purpose

Checks the I/O status of multiple file descriptors and message queues.

Library

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

Syntax

```
#include <sys/poll.h>
#include <sys/select.h>
#include <sys/types.h>

int poll(ListPointer, Nfdsmsgs, Timeout)
void *ListPointer;
unsigned long Nfdsmsgs;
long Timeout;
```

Description

The **poll** subroutine checks the specified file descriptors and message queues to see if they are ready for reading (receiving) or writing (sending), or to see if they have an exceptional condition pending.

Note: The **poll** subroutine applies only to character devices, pipes, message queues, and sockets. Not all character device drivers support it. See the descriptions of individual character devices for information about whether and how specific device drivers support the **poll** and **select** subroutines.

Parameters

ListPointer Specifies a pointer to an array of **pollfd** structures, **pollmsg** structures, or to a **pollist** structure. Each structure specifies a file descriptor or message queue ID and the events of interest for this file or message queue. The **pollfd**, **pollmsg**, and **pollist** structures are defined in the **/usr/include/sys/poll.h** file. If a **pollist** structure is to be used, a structure similar to the following should be defined in a user program. The **pollfd** structure must precede the **pollmsg** structure.

```
struct pollist {
   struct pollfd fds[3];
   struct pollmsg msgs[2];
   } list;
```

The structure can then be initialized as follows:

```
list.fds[0].fd = file_descriptorA;
list.fds[0].events = requested_events;
list.msgs[0].msgid = message_id;
list.msgs[0].events = requested_events;
```

The rest of the elements in the**fds**and**msgs**arrays can be initialized the same way. The **poll** subroutine can then be called, as follows:

```
nfds = 3;   /* number of pollfd structs */
nmsgs = 2;   /* number of pollmsg structs */
timeout = 1000   /* number of milliseconds to timeout
*/
poll(&list, (nmsgs<<16)|(nfds), 1000);
```

The exact number of elements in the **fds** and **msgs** arrays must be used in the calculation of the *Nfdsmsgs* parameter.

Nfdsmsgs Specifies the number of file descriptors and the exact number of message queues to check. The low–order 16 bits give the number of elements in the array of **pollfd** structures, while the high–order 16 bits give the exact number of elements in the array of **pollmsg** structures. If either half of the *Nfdsmsgs* parameter is equal to a value of 0, the corresponding array is assumed not to be present.

Timeout Specifies the maximum length of time (in milliseconds) to wait for at least one of the specified events to occur. If the *Timeout* parameter value is –1, the **poll** subroutine does not return until at least one of the specified events has occurred. If the value of the *Timeout* parameter is 0, the **poll** subroutine does not wait for an event to occur but returns immediately, even if none of the specified events have occurred.

poll Subroutine STREAMS Extensions

In addition to the functions described above, the **poll** subroutine multiplexes input/output over a set of file descriptors that reference open streams. The **poll** subroutine identifies those streams on which you can send or receive messages, or on which certain events occurred.

You can receive messages using the **read** subroutine or the **getmsg** system call. You can send messages using the **write** subroutine or the **putmsg** system call. Certain **streamio** operations, such as **I_RECVFD** and **I_SENDFD** can also be used to send and receive messages. See the **streamio** operations.

The *ListPointer* parameter specifies the file descriptors to be examined and the events of interest for each file descriptor. It points to an array having one element for each open file descriptor of interest. The array's elements are **pollfd** structures. In addition to the **pollfd** structure in the /**usr**/**include**/**sys**/**poll.h** file, STREAMS supports the following members:

```
int fd;             /*  file descriptor  */
 short events;       /* requested events */
 short revents;      /* returned events  */
```

The `fd` field specifies an open file descriptor and the `events` and `revents` fields are bit–masks constructed by ORing any combination of the following event flags:

POLLIN	A nonpriority or file descriptor–passing message is present on the stream–head read queue. This flag is set even if the message is of 0 length. In the `revents` field this flag is mutually exclusive with the **POLLPRI** flag. See the **I_RECVFD** command.
POLLRDNORM	A nonpriority message is present on the stream–head read queue.
POLLRDBAND	A priority message (band > 0) is present on the stream–head read queue.
POLLPRI	A high–priority message is present on the stream–head read queue. This flag is set even if the message is of 0 length. In the `revents` field, this flag is mutually exclusive with the **POLLIN** flag.
POLLOUT	The first downstream write queue in the stream is not full. Normal priority messages can be sent at any time. See the **putmsg** system call.
POLLWRNORM	The same as **POLLOUT**.
POLLWRBAND	A priority band greater than 0 exists downstream and priority messages can be sent at anytime.
POLLMSG	A message containing the **SIGPOLL** signal has reached the front of the stream–head read queue.

Return Values

On successful completion, the **poll** subroutine returns a value that indicates the total number of file descriptors and message queues that satisfy the selection criteria. The return value is similar to the *Nfdsmsgs* parameter in that the low–order 16 bits give the number of file descriptors, and the high–order 16 bits give the number of message queue identifiers that had nonzero `revents` values. The **NFDS** and **NMSGS** macros, found in the **sys/select.h** file, can be used to separate these two values from the return value. The **NFDS** macro returns **NFDS#**, where the number returned indicates the number of files reporting some event or error, and the **NMSGS** macro returns **NMSGS***#*, where the number returned indicates the number of message queues reporting some event or error.

A value of 0 indicates that the **poll** subroutine timed out and that none of the specified files or message queues indicated the presence of an event (all `revents` fields were values of 0).

If unsuccessful, a value of –1 is returned and the global variable **errno** is set to indicate the error.

Error Codes

The **poll** subroutine does not run successfully if one or more of the following are true:

EAGAIN	Allocation of internal data structures was unsuccessful.
EINTR	A signal was caught during the **poll** system call and the signal handler was installed with an indication that subroutines are not to be restarted.

EINVAL The number of **pollfd** structures specified by the *Nfdsmsgs* parameter is greater than the maximum number of open files, **OPEN_MAX**. This error is also returned if the number of **pollmsg** structures specified by the *Nfdsmsgs* parameter is greater than the maximum number of allowable message queues.

EFAULT The *ListPointer* parameter in conjunction with the *Nfdsmsgs* parameter addresses a location outside of the allocated address space of the process.

Implementation Specifics

This subroutine is part of Base Operating System (BOS) Runtime.

For compatibility with previous releases of this operating system and with BSD systems, the **select** subroutine is also supported.

Related Information

The **read** subroutine, **select** subroutine, **write** subroutine.

The **getmsg** system call, **putmsg** system call.

The **streamio** operations.

The STREAMS Overview and the Input and Output Handling Programmer's Overview in *AIX General Programming Concepts : Writing and Debugging Programs*.

popen Subroutine

Purpose

Initiates a pipe to a process.

Library

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

Syntax

```
#include <stdio.h>

FILE *popen (Command, Type)
const char *Command, *Type;
```

Description

The **popen** subroutine creates a pipe between the calling program and a shell command to be executed.

Note: The **popen** subroutine runs only **sh** shell commands. The results are unpredictable if the *Command* parameter is not a valid **sh** shell command. If the terminal is in a trusted state, the **tsh** shell commands are run.

If streams opened by previous calls to the **popen** subroutine remain open in the parent process, the **popen** subroutine closes them in the child process.

The **popen** subroutine returns a pointer to a **FILE** structure for the stream.

Attention: If the original processes and the process started with the **popen** subroutine concurrently read or write a common file, neither should use buffered I/O. If they do, the results are unpredictable.

Some problems with an output filter can be prevented by flushing the buffer with the **fflush** subroutine.

Parameters

Command	Points to a null–terminated string containing a shell command line.
Type	Points to a null–terminated string containing an I/O mode. If the *Type* parameter is the value **r**, you can read from the standard output of the command by reading from the file *Stream*. If the *Type* parameter is the value **w**, you can write to the standard input of the command by writing to the file *Stream*. Because open files are shared, a type **r** command can be used as an input filter and a type **w** command as an output filter.

Return Values

The **popen** subroutine returns a null pointer if files or processes cannot be created, or if the shell cannot be accessed.

Error Codes

The **popen** subroutine may set the **EINVAL** variable if the *Type* parameter is not valid. The **popen** subroutine may also set **errno** global variables as described by the **fork** or **pipe** subroutines.

Implementation Specifics

This subroutine is part of Base Operating System (BOS) Runtime.

Related Information

The **fclose** or **fflush** subroutine, **fopen**, **freopen**, or **fdopen** subroutine, **fork or vfork** subroutine, **pclose** subroutine, **pipe** subroutine, **wait**, **waitpid**, or **wait3** subroutine.

Files, Directories, and File Systems for Programmers in *AIX General Programming Concepts : Writing and Debugging Programs*.

printf, fprintf, sprintf, wsprintf, vprintf, vfprintf, vsprintf, or vwsprintf Subroutine

Purpose

Prints formatted output.

Library

Standard C Library (**libc.a**) or the Standard C Library with 128–Bit long doubles (**libc128.a**)

Syntax

```
#include <stdio.h>

int printf (Format, [Value, . . .])
const char *Format;

int fprintf (Stream, Format, [Value, . . .])
FILE *Stream;
const char *Format;

int sprintf (String, Format, [Value, . . .])
char *String;
const char *Format;

#include <stdarg.h>

int vprintf (Format, Value)
const char *Format;
va_list Value;

int vfprintf (Stream, Format, Value)
FILE *Stream;
const char *Format;
va_list Value;

int vsprintf (String, Format, Value)
char *String;
const char *Format;
va_list Value;

#include <wchar.h>

int vwsprintf (String, Format, Value)
wchar_t *String;
const char *Format;
va_list Value;

int wsprintf (String, Format,[Value, . . .])
wchar_t *String;
const char *Format;
```

Description

The **printf** subroutine converts, formats, and writes the *Value* parameter values, under control of the *Format* parameter, to the standard output stream. The **printf** subroutine provides conversion types to handle code points and **wchar_t** wide character codes.

The **fprintf** subroutine converts, formats, and writes the *Value* parameter values, under control of the *Format* parameter, to the output stream specified by the *Stream* parameter. This subroutine provides conversion types to handle code points and **wchar_t** wide character codes.

The **sprintf** subroutine converts, formats, and stores the *Value* parameter values, under control of the *Format* parameter, into consecutive bytes, starting at the address specified by the *String* parameter. The **sprintf** subroutine places a null character (\0) at the end. You must ensure that enough storage space is available to contain the formatted string. This subroutine provides conversion types to handle code points and **wchar_t** wide character codes.

The **wsprintf** subroutine converts, formats, and stores the *Value* parameter values, under control of the *Format* parameter, into consecutive **wchar_t** characters starting at the address specified by the *String* parameter. The **wsprintf** subroutine places a null character (\0) at the end. The calling process should ensure that enough storage space is available to contain the formatted string. The field width unit is specified as the number of **wchar_t** characters. The **wsprintf** subroutine is the same as the **printf** subroutine, except that the *String* parameter for the **wsprintf** subroutine uses a string of **wchar_t** wide–character codes.

All of the above subroutines work by calling the **_doprnt** subroutine, using variable–length argument facilities of the **varargs** macros.

The **vprintf**, **vfprintf**, **vsprintf**, and **vwsprintf** subroutines format and write **varargs** macros parameter lists. These subroutines are the same as the **printf**, **fprintf**, **sprintf**, and **wsprintf** subroutines, respectively, except that they are not called with a variable number of parameters. Instead, they are called with a parameter–list pointer as defined by the **varargs** macros.

Parameters

Value — Specifies 0 or more arguments that map directly to the objects in the *Format* parameter.

Stream — Specifies the output stream.

String — Specifies the starting address.

Format — A character string that contains two types of objects:

- Plain characters, which are copied to the output stream.
- Conversion specifications, each of which causes 0 or more items to be retrieved from the *Value* parameter list. In the case of the **vprintf**, **vfprintf**, **vsprintf**, and **vwsprintf** subroutines, each conversion specification causes 0 or more items to be retrieved from the **varargs** macros parameter lists.

  If the *Value* parameter list does not contain enough items for the *Format* parameter, the results are unpredictable. If more parameters remain after the entire *Format* parameter has been processed, the subroutine ignores them.

  Each conversion specification in the *Format* parameter has the following elements:

- A % (percent sign).
- 0 or more options, which modify the meaning of the conversion specification. The option characters and their meanings are:

' Formats the integer portions resulting from **i**, **d**, **u**, **f**, **g** and **G** decimal conversions with **thousands_sep** grouping characters. For other conversions the behavior is undefined. This option uses the nonmonetary grouping character.

– Left–justifies the result of the conversion within the field.

+ Begins the result of a signed conversion with a + (plus sign) or – (minus sign).

space character
Prefixes a space character to the result if the first character of a signed conversion is not a sign. If both the space–character and **+** option characters appear, the space–character option is ignored.

Converts the value to an alternate form. For **c**, **d**, **s**, and **u** conversions, the option has no effect. For **o** conversion, it increases the precision to force the first digit of the result to be a 0. For **x** and **X** conversions, a nonzero result has a 0x or 0X prefix. For **e**, **E**, **f**, **g**, and **G** conversions, the result always contains a decimal point, even if no digits follow it. For **g** and **G** conversions, trailing 0's are not removed from the result.

0 Pads to the field width with leading 0's (following any indication of sign or base) for **d**, **i**, **o**, **u**, **x**, **X**, **e**, **E**, **f**, **g**, and **G** conversions; the field is not space–padded. If the **0** and **–** options both appear, the **0** option is ignored. For **d**, **i**, **o u**, **x**, and **X** conversions, if a precision is specified, the **0** option is also ignored. If the **0** and **'** options both appear, grouping characters are inserted before the field is padded. For other conversions, the results are unreliable.

B Specifies a no–op character.

N Specifies a no–op character.

J Specifies a no–op character.

- An optional decimal digit string that specifies the minimum field width. If the converted value has fewer characters than the field width, the field is padded on the left to the length specified by the field width. If the **–** (left–justify) option is specified, the field is padded on the right.
- An optional precision. The precision is a **.** (dot) followed by a decimal digit string. If no precision is specified, the default value is 0. The precision specifies the following limits:

- Minimum number of digits to appear for the **d**, **i**, **o**, **u**, **x**, or **X** conversions.
- Number of digits to appear after the decimal point for the **e**, **E**, and **f** conversions.
- Maximum number of significant digits for **g** and **G** conversions.
- Maximum number of bytes to be printed from a string in **s** and **S** conversions.
- Maximum number of bytes, converted from the **wchar_t** array, to be printed from the **S** conversions. Only complete characters are printed.

- An optional **l** (lowercase *L*), **ll** (lowercase *LL*), **h**, or **L** specifier indicates one of the following:
  - An optional **h** specifying that a subsequent **d**, **i**, **u**, **o**, **x**, or **X** conversion specifier applies to a **short int** or **unsigned short int** *Value* parameter (the parameter will have been promoted according to the integral promotions, and its value will be converted to a **short int** or **unsigned short int** before printing).
  - An optional **h** specifying that a subsequent **n** conversion specifier applies to a pointer to a **short int** parameter.
  - An optional **l** (lowercase *L*) specifying that a subsequent **d**, **i**, **u**, **o**, **x**, or **X** conversion specifier applies to a **long int** or u**nsigned long int** parameter .
  - An optional **l** (lowercase *L*) specifying that a subsequent **n** conversion specifier applies to a pointer to a **long int** parameter.
  - An optional **ll** (lowercase *LL*) specifying that a subsequent **d**, **i**, **u**, **o**, **x**, or **X** conversion specifier applies to a **long long int** or **unsigned long long int** parameter.
  - An optional **ll** (lowercase *LL*) specifying that a subsequent **n** conversion specifier applies to a pointer to a **long long int** parameter.
  - An optional **L** specifying that a following **e**, **E**, **f**, **g**, or **G** conversion specifier applies to a **long double** parameter. If linked with **libc.a**, **long double** is the same as double (64bits). If linked with **libc128.a** and **libc.a**, **long double** is 128 bits.

- The following characters indicate the type of conversion to be applied:

% Performs no conversion. Prints (**%**).

d or **i** Accepts a *Value* parameter specifying an integer and converts it to signed decimal notation. The precision specifies the minimum number of digits to appear. If the value being converted can be represented in fewer digits, it is expanded with leading 0's. The default precision is 1. The result of converting a value of 0 with a precision of 0 is a null string. Specifying a field width with a 0 as a leading character causes the field–width value to be padded with leading 0's.

u Accepts a *Value* parameter specifying an unsigned integer and converts it to unsigned decimal notation. The precision specifies the minimum number of digits to appear. If the value being converted can be represented in fewer digits, it is expanded with leading 0's. The default precision is 1. The result of converting a value of 0 with a precision of 0 is a null string. Specifying a field width with a 0 as a leading character causes the field–width value to be padded with leading 0's.

o Accepts a *Value* parameter specifying an unsigned integer and converts it to unsigned octal notation. The precision specifies the minimum number of digits to appear. If the value being converted can be represented in fewer digits, it is expanded with leading 0's. The default precision is 1. The result of converting a value of 0 with a precision of 0 is a null string. Specifying a field–width with a 0 as a leading character causes the field width value to be padded with leading 0's. An octal value for field width is not implied.

x or **X** Accepts a *Value* parameter specifying an unsigned integer and converts it to unsigned hexadecimal notation. The letters **abcdef** are used for the **x** conversion and the letters **ABCDEF** are used for the **X** conversion. The precision specifies the minimum number of digits to appear. If the value being converted can be represented in fewer digits, it is expanded with leading 0's. The default precision is 1. The result of converting a value of 0 with a precision of 0 is a null string. Specifying a field width with a 0 as a leading character causes the field–width value to be padded with leading 0's.

f Accepts a *Value* parameter specifying a double and converts it to decimal notation in the format [–]*ddd***.***ddd*. The number of digits after the decimal point is equal to the precision specification. If no precision is specified, six digits are output. If the precision is 0, no decimal point appears.

e or **E** Accepts a *Value* parameter specifying a double and converts it to the exponential form [–]*d.ddd***e**+/–*dd*. One digit exists before the decimal point, and the number of digits after the decimal point is equal to the precision specification. The precision specification can be in the range of 0–17 digits. If no precision is specified, six digits are output. If the precision is 0, no decimal point appears. The **E** conversion character produces a number with **E** instead of **e** before the exponent. The exponent always contains at least two digits.

g or **G**	Accepts a *Value* parameter specifying a double and converts it in the style of the **e**, **E**, or **f** conversion characters, with the precision specifying the number of significant digits. Trailing 0's are removed from the result. A decimal point appears only if it is followed by a digit. The style used depends on the value converted. Style **e** (**E**, if **G** is the flag used) results only if the exponent resulting from the conversion is less than –4, or if it is greater or equal to the precision. If an explicit precision is 0, it is taken as 1.
c	Accepts and prints a *Value* parameter specifying an integer converted to an **unsigned char** data type.
C	Accepts and prints a *Value* parameter specifying a **wchar_t** wide character code. The **wchar_t** wide character code specified by the *Value* parameter is converted to an array of bytes representing a character and that character is written; the *Value* parameter is written without conversion when using the **wsprintf** subroutine.
s	Accepts a *Value* parameter as a string (character pointer), and characters from the string are printed until a null character (\0) is encountered or the number of bytes indicated by the precision is reached. If no precision is specified, all bytes up to the first null character are printed. If the string pointer specified by the *Value* parameter has a null value, the results are unreliable.
S	Accepts a corresponding *Value* parameter as a pointer to a **wchar_t** string. Characters from the string are printed (without conversion) until a null character (\0) is encountered or the number of wide characters indicated by the precision is reached. If no precision is specified, all characters up to the first null character are printed. If the string pointer specified by the *Value* parameter has a value of null, the results are unreliable.
p	Accepts a pointer to void. The value of the pointer is converted to a sequence of printable characters, the same as an unsigned hexadecimal (x).
n	Accepts a pointer to an integer into which is written the number of characters (wide–character codes in the case of the **wsprintf** subroutine) written to the output stream by this call. No argument is converted.

A field width or precision can be indicated by an * (asterisk) instead of a digit string. In this case, an integer *Value* parameter supplies the field width or precision. The *Value* parameter converted for output is not retrieved until the conversion letter is reached, so the parameters specifying field width or precision must appear before the value (if any) to be converted.

If the result of a conversion is wider than the field width, the field is expanded to contain the converted result and no truncation occurs. However, a small field width or precision can cause truncation on the right.

The **printf**, **fprintf**, **sprintf**, **wsprintf**, **vprintf**, **vfprintf**, **vsprintf**, or **vwsprintf** subroutine allows the insertion of a language–dependent radix character in the output string. The radix character is defined by language–specific data in the **LC_NUMERIC** category of the program's locale. In the C locale, or in a locale where the radix character is not defined, the radix character defaults to a . (dot).

After any of these subroutines runs successfully, and before the next successful completion of a call to the **fclose** or **fflush** subroutine on the same stream or to the **exit** or **abort** subroutine, the `st_ctime` and `st_mtime` fields of the file are marked for update.

The **e**, **E**, **f**, **g**, and **G** conversion specifiers represent the special floating–point values as follows:

Quiet NaN	+NaNQ or –NaNQ
Signaling NaN	+NaNS or –NaNS
+/–INF	+INF or –INF
+/–0	+0 or –0

The representation of the + (plus sign) depends on whether the **+** or space–character formatting option is specified.

These subroutines can handle a format string that enables the system to process elements of the parameter list in variable order. In such a case, the normal conversion character % (percent sign) is replaced by **%***digit***$**, where *digit* is a decimal number in the range from 1 to the **NL_ARGMAX** value. Conversion is then applied to the specified argument, rather than to the next unused argument. This feature provides for the definition of format strings in an order appropriate to specific languages. When variable ordering is used the * (asterisk) specification for field width in precision is replaced by **%***digit***$**. If you use the variable–ordering feature, you must specify it for all conversions.

The following criteria apply:

- The format passed to the NLS extensions can contain either the format of the conversion or the explicit or implicit argument number. However, these forms cannot be mixed within a single format string, except for %% (double percent sign).
- The *n* value must have no leading zeros.
- If **%***n***$** is used, **%1$** to **%***n* – **1$** inclusive must be used.
- The *n* in **%***n***$** is in the range from 1 to the **NL_ARGMAX** value, inclusive. See the **limits.h** file for more information about the **NL_ARGMAX** value.
- Numbered arguments in the argument list can be referenced as many times as required.
- The * (asterisk) specification for field width or precision is not permitted with the variable order **%***n***$** format; instead, the ******m***$** format is used.

Return Values

Upon successful completion, the **printf**, **fprintf**, **vprintf**, and **vfprintf** subroutines return the number of bytes transmitted (not including the null character [\0] in the case of the **sprintf** and **vsprintf** subroutines). If an error was encountered, a negative value is output.

Upon successful completion, the **wsprintf** and **vwsprintf** subroutines return the number of wide characters transmitted (not including the wide character null character [\0]). If an error was encountered, a negative value is output.

Error Codes

The **printf**, **fprintf**, **sprintf**, or **wsprintf** subroutine is unsuccessful if the file specified by the *Stream* parameter is unbuffered or the buffer needs to be flushed and one or more of the following are true:

EAGAIN	The **O_NONBLOCK** flag is set for the file descriptor underlying the file specified by the *Stream* or *String* parameter and the process would be delayed in the write operation.
EBADF	The file descriptor underlying the file specified by the *Stream* or *String* parameter is not a valid file descriptor open for writing.

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

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

Note: Depending upon which library routine the application binds to, this subroutine may return **EINTR**. Refer to the **signal** subroutine regarding **sa_restart**.

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

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

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

The **printf**, **fprintf**, **sprintf**, or **wsprintf** subroutine may be unsuccessful if one or more of the following are true:

EILSEQ An invalid character sequence was detected.

EINVAL The *Format* parameter received insufficient arguments.

ENOMEM Insufficient storage space is available.

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

Examples

The following example demonstrates how the **vfprintf** subroutine can be used to write an error routine:

```
#include <stdio.h>
#include <stdarg.h>
/* The error routine should be called with the
      syntax:       */
/* error(routine_name, Format
     [, value, . . . ]); */
/*VARARGS0*/
void error(char *fmt, . . .);
/* **  Note that the function name and
     Format arguments cannot be **
     separately declared because of the **
     definition of varargs.  */  {
   va_list args;

   va_start(args, fmt);
   /*
   ** Display the name of the function
      that called the error routine   */
   fprintf(stderr, "ERROR in %s: ",
      va_arg(args, char *));   /*
   ** Display the remainder of the message
   */
   fmt = va_arg(args, char *);
   vfprintf(fmt, args);
   va_end(args);
    abort();  }
```

Implementation Specifics

These subroutines are part of Base Operating System (BOS) Runtime.

Related Information

The **abort** subroutine, **conv** subroutine, **ecvt**, **fcvt**, or **gcvt** subroutine, **exit** subroutine, **fclose** or **fflush** subroutine, **putc**, **putchar**, **fputc**, or **putw** subroutine, **putwc**, **putwchar**, or **fputwc** subroutine, **scanf**, **fscanf**, **sscanf**, or **wsscanf** subroutine, **setlocale** subroutine.

Input and Output Handling Programmer's Overview and 128–Bit long Floating Point Data Type in *AIX General Programming Concepts : Writing and Debugging Programs*.

profil Subroutine

Purpose

Starts and stops program address sampling for execution profiling.

Library

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

Syntax #include <mon.h>

void profil (*ShortBuffer***,** *BufferSize***,** *Offset***,** *Scale***)**
OR
void profil (*ProfBuffer***, –1, 0, 0)**

unsigned short ****ShortBuffer*;**
struct prof ****ProfBuffer*;**
unsigned int *Buffersize***,** *Scale***;**
unsigned long *Offset***;**

Description

The **profil** subroutine arranges to record a histogram of periodically sampled values of the calling process program counter. If *BufferSize* is not –1:

- The parameters to the **profil** subroutine are interpreted as shown in the first syntax definition.
- After this call, the program counter (pc) of the process is examined each clock tick if the process is the currently active process. The value of the *Offset* parameter is subtracted from the pc. The result is multiplied by the value of the *Scale* parameter, shifted right 16 bits, and rounded up to the next half–word aligned value. If the resulting number is less than the *BufferSize* value divided by **sizeof(short)**, the corresponding **short** inside the *ShortBuffer* parameter is incremented. If the result of this increment would overflow an unsigned short, it remains USHRT_MAX.
- The least significant 16 bits of the *Scale* parameter are interpreted as an unsigned, fixed–point fraction with a binary point at the left. The most significant 16 bits of the *Scale* parameter are ignored. For example:

Octal	Hex	Meaning
0177777	0xFFFF	Maps approximately each pair of bytes in the instruction space to a unique **short** in the *ShortBuffer* parameter.
077777	0x7FFF	Maps approximately every four bytes to a **short** in the *ShortBuffer* parameter.
02	0x0002	Maps all instructions to the same location, producing a noninterrupting core clock.
01	0x0001	Turns profiling off.
00	0x0000	Turns profiling off.

Note: Mapping each byte of the instruction space to an individual **short** in the *ShortBuffer* parameter is not possible.

- Profiling, using the first syntax definition, is rendered ineffective by giving a value of 0 for the *BufferSize* parameter.

If the value of the *BufferSize* parameter is –1:

- The parameters to the **profil** subroutine are interpreted as shown in the second syntax definition. In this case, the *Offset* and *Scale* parameters are ignored, and the *ProfBuffer* parameter points to an array of **prof** structures. The **prof** structure is defined in the **mon.h** file, and it contains the following members:

```
caddr_t          p_low;
caddr_t          p_high;
HISTCOUNTER     *p_buff;
int              p_bufsize;
uint             p_scale;
```

If the `p_scale` member has the value of –1, a value for it is computed based on `p_low`, `p_high`, and `p_bufsize`; otherwise `p_scale` is interpreted like the scale argument in the first synopsis. The `p_high` members in successive structures must be in ascending sequence. The array of structures is ended with a structure containing a `p_high` member set to 0; all other fields in this last structure are ignored.

The `p_buff` buffer pointers in the array of **prof** structures must point into a single contiguous buffer space.

- Profiling, using the second syntax definition, is turned off by giving a *ProfBuffer* argument such that the `p_high` element of the first structure is equal to 0.

In every case:

- Profiling remains on in both the child process and the parent process after a **fork** subroutine.
- Profiling is turned off when an **exec** subroutine is run.
- A call to the **profil** subroutine is ineffective if profiling has been previously turned on using one syntax definition, and an attempt is made to turn profiling off using the other syntax definition.
- A call to the **profil** subroutine is ineffective if the call is attempting to turn on profiling when profiling is already turned on, or if the call is attempting to turn off profiling when profiling is already turned off.

Parameters

ShortBuffer	Points to an area of memory in the user address space. Its length (in bytes) is given by the *BufferSize* parameter.
BufferSize	Specifies the length (in bytes) of the buffer.
Offset	Specifies the delta of program counter start and buffer; for example, a 0 *Offset* implies that text begins at 0. If the user wants to use the entry point of a routine for the *Offset* parameter, the syntax of the parameter is as follows: ***(long *)***RoutineName*
Scale	Specifies the mapping factor between the program counter and *ShortBuffer.*
ProfBuffer	Points to an array of **prof** structures.

Return Values

The **profil** subroutine always returns a value of 0. Otherwise, the **errno** global variable is set to indicate the error.

Error Codes

The **profil** subroutine is unsuccessful if one or both of the following are true:

EFAULT The address specified by the *ShortBuffer* or *ProfBuffer* parameters is not valid, or the address specified by a `p_buff` field is not valid. EFAULT can also occur if there are not sufficient resources to pin the profiling buffer in real storage.

EINVAL The `p_high` fields in the **prof** structure specified by the *ProfBuffer* parameter are not in ascending order.

Implementation Specifics

This subroutine is part of Base Operating System (BOS) Runtime.

Related Information

The **exec** subroutines, **fork** subroutine, **moncontrol** subroutine, **monitor** subroutine, **monstartup** subroutine.

The **prof** command.

psdanger Subroutine

Purpose

Defines the amount of free paging space available.

Syntax

```
#include <signal.h>

int psdanger (Signal)
int Signal;
```

Description

The **psdanger** subroutine returns the difference between the current number of free paging–space blocks and the paging–space thresholds of the system.

Parameters

Signal	Defines the signal.

Return Values

If the value of the *Signal* parameter is 0, the return value is the total number of paging–space blocks defined in the system.

If the value of the *Signal* parameter is –1, the return value is the number of free paging–space blocks available in the system.

If the value of the *Signal* parameter is **SIGDANGER**, the return value is the difference between the current number of free paging–space blocks and the paging–space warning threshold. If the number of free paging–space blocks is less than the paging–space warning threshold, the return value is negative.

If the value of the *Signal* parameter is **SIGKILL**, the return value is the difference between the current number of free paging–space blocks and the paging–space kill threshold. If the number of free paging–space blocks is less than the paging–space kill threshold, the return value is negative.

Implementation Specifics

This subroutine is part of Base Operating System (BOS) Runtime.

Related Information

The **swapon** subroutine, **swapqry** subroutine.

The **chps** command, **lsps** command, **mkps** command, **rmps** command, **swapon** command.

Paging Space Overview in *AIX 4.3 System Management Guide: Operating System and Devices*.

Subroutines Overview and Understanding Paging Space Programming Requirements in *AIX General Programming Concepts : Writing and Debugging Programs*.

psignal Subroutine or sys_siglist Vector

Purpose

Prints system signal messages.

Library

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

Syntax

```
psignal (Signal, String)
unsigned Signal;
char *String;

char *sys_siglist[ ];
```

Description

The **psignal** subroutine produces a short message on the standard error file describing the indicated signal. First the *String* parameter is printed, then the name of the signal and a new–line character.

To simplify variant formatting of signal names, the **sys_siglist** vector of message strings is provided. The signal number can be used as an index in this table to get the signal name without the new–line character. The **NSIG** defined in the **signal.h** file is the number of messages provided for in the table. It should be checked because new signals may be added to the system before they are added to the table.

Parameters

Signal	Specifies a signal. The signal number should be among those found in the **signal.h** file.
String	Specifies a string that is printed. Most usefully, the *String* parameter is the name of the program that incurred the signal.

Implementation Specifics

These subroutines are part of Base Operating System (BOS) Runtime.

Related Information

The **perror** subroutine, **sigvec** subroutine.

pthread_atfork Subroutine

Purpose

Registers fork handlers.

Library

Threads Library (**libpthreads.a**)

Syntax

```
#include <sys/types.h>
#include <unistd.h>

int pthread_atfork (void (*prepare)(void), void (*parent)(void)
void (*child)(void));
```

Description

The **pthread_atfork** subroutine registers fork cleanup handlers. The *prepare* handler is called before the processing of the **fork** subroutine commences. The *parent* handler is called after the processing of the **fork** subroutine completes in the parent process. The *child* handler is called after the processing of the **fork** subroutine completes in the child process.

When the **fork** subroutine is called, only the calling thread is duplicated in the child process, but all synchronization variables are duplicated. The **pthread_atfork** subroutine provides a way to prevent state inconsistencies and resulting deadlocks. The expected usage is that the *prepare* handler acquires all mutexes, and the two other handlers release them in the parent and child processes.

The prepare handlers are called in LIFO (Last In First Out) order; whereas the parent and child handlers are called in FIFO (first–in first–out) order. Thereafter, the order of calls to the **pthread_atfork** subroutine is significant.

Note: The **pthread.h** header file must be the first included file of each source file using the threads library.

Parameters

prepare	Points to the pre–fork cleanup handler. If no pre–fork handling is desired, the value of this pointer should be set to **NULL**.
parent	Points to the parent post–fork cleanup handler. If no parent post–fork handling is desired, the value of this pointer should be set to **NULL**.
child	Points to the child post–fork cleanup handler. If no child post–fork handling is desired, the value of this pointer should be set to **NULL**.

Return Values

Upon successful completion, pthread_atfork returns a value of zero. Otherwise, an error number is returned to indicate the error.

Error Codes

The pthread_atfork function will fail if:

ENOMEM	Insufficient table space exists to record the fork handler addresses.

The pthread_atfork function will not return an error code of **EINTR**.

Implementation Specifics

This subroutine is part of the Base Operating System (BOS) Runtime.

Related Information

sys/types.h

The **fork** subroutine.

The **atexit** subroutine.

Process Duplication and Termination in *AIX General Programming Concepts : Writing and Debugging Programs*.

Threads Library Quick Reference in *AIX General Programming Concepts : Writing and Debugging Programs*.

pthread_attr_destroy Subroutine

Purpose

Deletes a thread attributes object.

Library

Threads Library (**libpthreads.a**)

Syntax

```
#include <pthread.h>

int pthread_attr_destroy (attr)
pthread_attr_t *attr;
```

Description

The **pthread_attr_destroy** subroutine destroys the thread attributes object *attr*, reclaiming its storage space. It has no effect on the threads previously created with that object.

Parameters

attr	Specifies the thread attributes object to delete.

Return Values

Upon successful completion, 0 is returned. Otherwise, an error code is returned.

Error Codes

The **pthread_attr_destroy** subroutine is unsuccessful if the following is true:

EINVAL	The *attr* parameter is not valid.

This function will not return an error code of [EINTR].

Implementation Specifics

This subroutine is part of the Base Operating System (BOS) Runtime.

Related Information

The **pthread_attr_init** subroutine, **pthread_create** subroutine, the **pthread.h** file.

Creating Threads in *AIX General Programming Concepts : Writing and Debugging Programs*.

Threads Library Quick Reference in *AIX General Programming Concepts : Writing and Debugging Programs*.

pthread_attr_getdetachstate or pthread_attr_setdetachstate Subroutines

Purpose

Sets and returns the value of the detachstate attribute of a thread attributes object.

Library

Threads Library (**libpthreads.a**)

Syntax

```
#include <pthread.h>




int pthread_attr_setdetachstate (pthread_attr_t *attr, int
detachstate)
int pthread_attr_getdetachstate (const pthread_attr_t *attr,
   int * detachstate);
```

Description

The detachstate attribute controls whether the thread is created in a detached state. If the thread is created detached, then use of the ID of the newly created thread by the **pthread_detach** or **pthread_join** function is an error.

The **pthread_attr_setdetachstate** and **pthread_attr_getdetachstate**, respectively, set and get the **detachstate** attribute in the **attr** object.

The detachstate can be set to either PTHREAD_CREATE_DETACHED or PTHREAD_CREATE_JOINABLE. A value of PTHREAD_CREATE_DETACHED causes all threads created with **attr** to be in the detached state, whereas using a value of PTHREAD_CREATE_JOINABLE causes all threads created with **attr** to be in the joinable state. The default value of the **detachstate** attribute is PTHREAD_CREATE_JOINABLE.

Parameters

attr	Specifies the thread attributes object.
detachstate	Points to where the detachstate attribute value will be stored.

Return Values

Upon successful completion, **pthread_attr_setdetachstate** and **pthread_attr_getdetachstate** return a value of **0**. Otherwise, an error number is returned to indicate the error.

The **pthread_attr_getdetachstate** function stores the value of the detachstate attribute in detachstate if successful.

Error Codes

The **pthread_attr_setdetachstate** function will fail if:

EINVAL	The value of *detachstate* was not valid.

The **pthread_attr_getdetachstate** and **pthread_attr_setdetachstate** functions will fail if:

EINVAL	The attribute parameter is invalid.

These functions will not return an error code of EINTR.

Implementation Specifics

This subroutine is part of the Base Operating System (BOS) Runtime.

Related Information

The **pthread_attr_setstackaddr**, **pthread_attr_setstacksize**, **pthread_create**, **pthread_attr_init** subroutines, and **pthread.h** file.

Creating Threads in *AIX General Programming Concepts : Writing and Debugging Programs*.

Threads Library Quick Reference in *AIX General Programming Concepts : Writing and Debugging Programs*.

pthread_attr_getguardsize or pthread_attr_setguardsize Subroutines

Purpose

Gets or sets the thread guardsize attribute.

Library

Threads Library (**libthreads.a**)

Syntax

```
#include <pthread.h>

int pthread_attr_getguardsize (const pthread_attr_t *attr, size_t
*guardsize );
int pthread_attr_setguardsize (pthread_attr_t *attr, size_t
guardsize );
```

Description

The guardsize attribute controls the size of the guard area for the created thread's stack. The guardsize attribute provides protection against overflow of the stack pointer. If a thread's stack is created with guard protection, the implementation allocates extra memory at the overflow end of the stack as a buffer against stack overflow of the stack pointer. If an application overflows into this buffer an error results (possibly in a SIGSEGV signal being delivered to the thread).

The guardsize attribute is provided to the application for two reasons:

- Overflow protection can potentially result in wasted system resources. An application that creates a large number of threads, and which knows its threads will never overflow their stack, can save system resources by turning off guard areas.
- When threads allocate large data structures on the stack, large guard areas may be needed to detect stack overflow.

The **pthread_attr_getguardsize** function gets the guardsize attribute in the attr object. This attribute is returned in the *guardsize* parameter.

The **pthread_attr_setguardsize** function sets the guardsize attribute in the attr object. The new value of this attribute is obtained from the *guardsize* parameter. If *guardsize* is zero, a guard area will not be provided for threads created with attr. If *guardsize* is greater than zero, a guard area of at least size guardsize bytes is provided for each thread created with attr.

A conforming implementation is permitted to round up the value contained in *guardsize* to a multiple of the configurable system variable PAGESIZE (see **sys**/**mman.h**). If an implementation rounds up the value of guardsize to a multiple of PAGESIZE, a call to **pthread_attr_getguardsize** specifying attr will store in the *guardsize* parameter the guard size specified by the previous **pthread_attr_setguardsize** function call. The default value of the guardsize attribute is PAGESIZE bytes. The actual value of PAGESIZE is implementation–dependent and may not be the same on all implementations.

If the stackaddr attribute has been set (that is, the caller is allocating and managing its own thread stacks), the guardsize attribute is ignored and no protection will be provided by the implementation. It is the responsibility of the application to manage stack overflow along with stack allocation and management in this case.

Return Values

If successful, the **pthread_attr_getguardsize** and **pthread_attr_setsguardsize** functions return zero. Otherwise, an error number is returned to indicate the error.

Error Codes

The **pthread_attr_getguardsize** and **pthread_attr_setguardsize** functions will fail if:

EINVAL	The attribute attr is invalid.
EINVAL	The *guardsize* parameter is invalid.
EINVAL	The *guardsize* parameter contains an invalid value.

pthread_attr_getschedparam Subroutine

Purpose

Returns the value of the schedparam attribute of a thread attributes object.

Library

Threads Library (**libpthreads.a**)

Syntax

```
#include <pthread.h>
#include <sys/sched.h>

int pthread_attr_getschedparam (attr, schedparam)
const pthread_attr_t *attr;
struct sched_param *schedparam;
```

Description

The **pthread_attr_getschedparam** subroutine returns the value of the schedparam attribute of the thread attributes object *attr*. The schedparam attribute specifies the scheduling parameters of a thread created with this attributes object. The `sched_priority` field of the **sched_param** structure contains the priority of the thread. It is an integer value.

Note: The **pthread.h** header file must be the first included file of each source file using the threads library. Otherwise, the **–D_THREAD_SAFE** compilation flag should be used, or the cc_r compiler used. In this case, the flag is automatically set.

Parameters

attr	Specifies the thread attributes object.
schedparam	Points to where the schedparam attribute value will be stored.

Return Values

Upon successful completion, the value of the schedparam attribute is returned via the *schedparam* parameter, and 0 is returned. Otherwise, an error code is returned.

Error Codes

The **pthread_attr_getschedparam** subroutine is unsuccessful if the following is true:

EINVAL	The *attr* parameter is not valid.

This function does not return EINTR.

Implementation Specifics

This subroutine is part of the Base Operating System (BOS) Runtime.

Related Information

The **pthread_attr_setschedparam** subroutine, **pthread_attr_init** subroutine, **pthread_getschedparam** subroutine, the pthread.h file.

Threads Scheduling in *AIX General Programming Concepts : Writing and Debugging Programs*.

Threads Library Options and Threads Library Quick Reference in *AIX General Programming Concepts : Writing and Debugging Programs*.

pthread_attr_getstackaddr Subroutine

Purpose

Returns the value of the stackaddr attribute of a thread attributes object.

Library

Threads Library (**libpthreads.a**)

Syntax

```
#include <pthread.h>

int pthread_attr_getstackaddr (attr, stackaddr)
const pthread_attr_t *attr;
void **stackaddr;
```

Description

The **pthread_attr_getstackaddr** subroutine returns the value of the stackaddr attribute of the thread attributes object *attr*. This attribute specifies the stack address of the thread created with this attributes object.

Note: The **pthread.h** header file must be the first included file of each source file using the threads library. Otherwise, the **–D_THREAD_SAFE** compilation flag should be used, or the cc_r compiler used. In this case, the flag is automatically set.

Parameters

attr	Specifies the thread attributes object.
stackaddr	Points to where the stackaddr attribute value will be stored.

Return Values

Upon successful completion, the value of the stackaddr attribute is returned via the *stackaddr* parameter, and 0 is returned. Otherwise, an error code is returned.

Error Codes

The **pthread_attr_getstackaddr** subroutine is unsuccessful if the following is true:

EINVAL	The *attr* parameter is not valid.

This function will not return EINTR.

Implementation Specifics

This subroutine is part of the Base Operating System (BOS) Runtime.

Related Information

The **pthread_attr_setstackaddr** subroutine, **pthread_attr_init** subroutine, the pthread.h file.

Advanced Attributes in *AIX General Programming Concepts : Writing and Debugging Programs*.

Threads Library Options and Threads Library Quick Reference in *AIX General Programming Concepts : Writing and Debugging Programs*.

pthread_attr_getstacksize Subroutine

Purpose

Returns the value of the stacksize attribute of a thread attributes object.

Library

Threads Library (**libpthreads.a**)

Syntax

```
#include <pthread.h>

int pthread_attr_getstacksize (attr, stacksize)
const pthread_attr_t *attr;
size_t *stacksize;
```

Description

The **pthread_attr_getstacksize** subroutine returns the value of the stacksize attribute of the thread attributes object *attr*. This attribute specifies the minimum stack size of a thread created with this attributes object. The value is given in bytes. The default stack size is **PTHREAD_STACK_MIN**, *12 defined in **pthread.h**.

Note: The **pthread.h** header file must be the first included file of each source file using the threads library. Otherwise, the **–D_THREAD_SAFE** compilation flag should be used, or the cc_r compiler used. In this case, the flag is automatically set.

Parameters

attr	Specifies the thread attributes object.
stacksize	Points to where the stacksize attribute value will be stored.

Return Values

Upon successful completion, the value of the stacksize attribute is returned via the *stacksize* parameter, and 0 is returned. Otherwise, an error code is returned.

Error Codes

The **pthread_attr_getstacksize** subroutine is unsuccessful if the following is true:

EINVAL	The *attr* or *stacksize* parameters are not valid.

This function will not return an error code of [EINTR].

Implementation Specifics

This subroutine is part of the Base Operating System (BOS) Runtime.

Related Information

The **pthread_attr_setstacksize** subroutine, **pthread_attr_init** subroutine, the **pthread.h** file.

Advanced Attributes in *AIX General Programming Concepts : Writing and Debugging Programs*.

Threads Library Options and Threads Library Quick Reference in *AIX General Programming Concepts : Writing and Debugging Programs*.

pthread_attr_init Subroutine

Purpose

Creates a thread attributes object and initializes it with default values.

Library

Threads Library (**libpthreads.a**)

Syntax

```
#include <pthread.h>


int pthread_attr_init (attr)
pthread_attr_t *attr;
```

Description

The **pthread_attr_init** subroutine creates a new thread attributes object *attr*. The new thread attributes object is initialized with the following default values:

Always initialized	
Attribute	**Default value**
Detachstate	**PTHREAD_CREATE_JOINABLE**

Always Initialized	
Attribute	**Default value**
Contention–scope	**PTHREAD_SCOPE_PROCESS** the default ensures compatibility with implementations that do not support this POSIX option.
Inheritsched	**PTHREAD_INHERITSCHED**
Schedparam	A **sched_param** structure which `sched_prio` field is set to 1, the least favored priority.
Schedpolicy	**SCHED_OTHER**

Always Initialized	
Attribute	**Default value**
Stacksize	**PTHREAD_STACK_MIN**
Guardsize	**PAGESIZE**

The resulting attribute object (possibly modified by setting individual attribute values), when used by **pthread_create**, defines the attributes of the thread created. A single attributes object can be used in multiple simultaneous calls to **pthread_create**.

Parameters

attr	Specifies the thread attributes object to be created.

Return Values

Upon successful completion, the new thread attributes object is filled with default values and returned via the *attr* parameter, and 0 is returned. Otherwise, an error code is returned.

Error Codes

The **pthread_attr_init** subroutine is unsuccessful if the following is true:

EINVAL	The *attr* parameter is not valid.
ENOMEM	There is not sufficient memory to create the thread attribute object.

This function will not return an error code of [EINTR].

Implementation Specifics

This subroutine is part of the Base Operating System (BOS) Runtime.

Related Information

The **pthread_attr_setdetachstate** subroutine, **pthread_attr_setstackaddr** subroutine, **pthread_attr_setstacksize** subroutine, **pthread_create** subroutine, **pthread_attr_destroy** and **pthread_attr_setguardsize** subroutine.

The **pthread.h** file.

Creating Threads in *AIX General Programming Concepts : Writing and Debugging Programs*.

Threads Library Options and Threads Library Quick Reference in *AIX General Programming Concepts : Writing and Debugging Programs*.

pthread_attr_setschedparam Subroutine

Purpose

Sets the value of the schedparam attribute of a thread attributes object.

Library

Threads Library (**libpthreads.a**)

Syntax

```
#include <pthread.h>
#include <sys/sched.h>

int pthread_attr_setschedparam (attr, schedparam)
pthread_attr_t *attr;
const struct sched_param *schedparam;
```

Description

The **pthread_attr_setschedparam** subroutine sets the value of the schedparam attribute of the thread attributes object *attr*. The schedparam attribute specifies the scheduling parameters of a thread created with this attributes object. The `sched_priority` field of the **sched_param** structure contains the priority of the thread.

Note: The **pthread.h** header file must be the first included file of each source file using the threads library. Otherwise, the **–D_THREAD_SAFE** compilation flag should be used, or the cc_r compiler used. In this case, the flag is automatically set.

Parameters

attr	Specifies the thread attributes object.
schedparam	Points to where the scheduling parameters to set are stored. The `sched_priority` field must be in the range from 1 to 127, where 1 is the least favored priority, and 127 the most favored.

Return Values

Upon successful completion, 0 is returned. Otherwise, an error code is returned.

Error Codes

The **pthread_attr_setschedparam** subroutine is unsuccessful if the following is true:

EINVAL	The *attr* parameter is not valid.
ENOSYS	The priority scheduling POSIX option is not implemented.
ENOTSUP	The value of the schedparam attribute is not supported.

Implementation Specifics

This subroutine is part of the Base Operating System (BOS) Runtime.

Related Information

The **pthread_attr_getschedparam** subroutine, **pthread_attr_init** subroutine, **pthread_create** subroutine, the pthread.h file.

Threads Scheduling in *AIX General Programming Concepts : Writing and Debugging Programs*.

Threads Library Options and Threads Library Quick Reference in *AIX General Programming Concepts : Writing and Debugging Programs*.

pthread_attr_setstackaddr Subroutine

Purpose

Sets the value of the stackaddr attribute of a thread attributes object.

Library

Threads Library (**libpthreads.a**)

Syntax

```
#include <pthread.h>

int pthread_attr_setstackaddr (attr, stackaddr)
pthread_attr_t *attr;
void *stackaddr;
```

Description

The **pthread_attr_setstackaddr** subroutine sets the value of the stackaddr attribute of the thread attributes object *attr*. This attribute specifies the stack address of a thread created with this attributes object.

Note: The **pthread.h** header file must be the first included file of each source file using the threads library. Otherwise, the **–D_THREAD_SAFE** compilation flag should be used, or the cc_r compiler used. In this case, the flag is automatically set.

Parameters

attr	Specifies the thread attributes object.
stackaddr	Specifies the stack address to set. It is a void pointer.

Return Values

Upon successful completion, 0 is returned. Otherwise, an error code is returned.

Error Codes

The **pthread_attr_setstackaddr** subroutine is unsuccessful if the following is true:

EINVAL	The *attr* parameter is not valid.
ENOSYS	The stack address POSIX option is not implemented.

Implementation Specifics

This subroutine is part of the Base Operating System (BOS) Runtime.

Related Information

The **pthread_attr_getstackaddr** subroutine, **pthread_attr_init** subroutine, the **pthread.h** file.

Advanced Attributes in *AIX General Programming Concepts : Writing and Debugging Programs*.

Threads Library Options and Threads Library Quick Reference in *AIX General Programming Concepts : Writing and Debugging Programs*.

pthread_attr_setstacksize Subroutine

Purpose

Sets the value of the stacksize attribute of a thread attributes object.

Library

Threads Library (**libpthreads.a**)

Syntax

```
#include <pthread.h>

int pthread_attr_setstacksize (attr, stacksize)
pthread_attr_t *attr;
size_t stacksize;
```

Description

The **pthread_attr_setstacksize** subroutine sets the value of the stacksize attribute of the thread attributes object *attr*. This attribute specifies the minimum stack size, in bytes, of a thread created with this attributes object.

The allocated stack size is always a multiple of 8K bytes, greater or equal to the required minimum stack size of 56K bytes (**PTHREAD_STACK_MIN**). The following formula is used to calculate the allocated stack size: if the required stack size is lower than 56K bytes, the allocated stack size is 56K bytes; otherwise, if the required stack size belongs to the range from (56 + (*n* – 1) * 16) K bytes to (56 + *n* * 16) K bytes, the allocated stack size is (56 + *n* * 16) K bytes.

Note: The **pthread.h** header file must be the first included file of each source file using the threads library. Otherwise, the **–D_THREAD_SAFE** compilation flag should be used, or the cc_r compiler used. In this case, the flag is automatically set.

Parameters

attr	Specifies the thread attributes object.
stacksize	Specifies the minimum stack size, in bytes, to set. The default stack size is **PTHREAD_STACK_MIN**. The minimum stack size should be greater or equal than this value.

Return Values

Upon successful completion, 0 is returned. Otherwise, an error code is returned.

Error Codes

The **pthread_attr_setstacksize** subroutine is unsuccessful if the following is true:

EINVAL	The *attr* parameter is not valid, or the value of the *stacksize* parameter exceeds a system imposed limit.
ENOSYS	The stack size POSIX option is not implemented.

Implementation Specifics

This subroutine is part of the Base Operating System (BOS) Runtime.

Related Information

The **pthread_attr_getstacksize** subroutine, **pthread_attr_init** subroutine, **pthread_create** subroutine, the **pthread.h** file.

Advanced Attributes in *AIX General Programming Concepts : Writing and Debugging Programs*.

Threads Library Options and Threads Library Quick Reference in *AIX General Programming Concepts : Writing and Debugging Programs*.

pthread_attr_setsuspendstate_np and pthread_attr_getsuspendstate_np Subroutine

Purpose

Controls whether a thread is created in a suspended state.

Library

Threads Library (**libpthreads.a**)

Syntax

```
#include <pthread.h>

int pthread_attr_setsuspendstate_np(pthread_attr_t, *attr, int
suspendstate);

int pthread_attr_getsuspendstate_np(pthread_attr_t, *attr, int
*suspendstate);
```

Description

The *suspendstate* attribute controls whether the thread is created in a suspended state. If the thread is created suspended, the thread start routine will not execute until **pthread_continue_np** is run on the thread. The **pthread_attr_setsuspendstate_np** and **pthread_attr_getsuspendstate_np** routines, respectively, set and get the *suspendstate* attribute in the *attr* object.

The *suspendstate* attribute can be set to either **PTHREAD_CREATE_SUSPENDED_NP** or **PTHREAD_CREATE_UNSUSPENDED_NP**. A value of **PTHREAD_CREATE_SUSPENDED_NP** causes all threads created with *attr* to be in the suspended state, whereas using a value of **PTHREAD_CREATE_UNSUSPENDED_NP** causes all threads created with *attr* to be in the unsuspended state. The default value of the *suspendstate* attribute is **PTHREAD_CREATE_UNSUSPENDED_NP**.

Parameters

attr	Specifies the thread attributes object.
suspendstate	Points to where the *suspendstate* attribute value will be stored.

Return Values

Upon successful completion, **pthread_attr_setsuspendstate_np** and **pthread_attr_getsuspendstate_np** return a value of 0. Otherwise, an error number is returned to indicate the error.

The **pthread_attr_getsuspendstate_np** function stores the value of the *suspendstate* attribute in *suspendstate* if successful.

Error Codes

The **pthread_attr_setsuspendstate_np** function will fail if:

EINVAL	The value of *suspendstate* is not valid.

Implementation Specifics

This subroutine is part of the Base Operating System (BOS) Runtime.

pthread_cancel Subroutine

Purpose

Requests the cancellation of a thread.

Library

Threads Library (**libpthreads.a**)

Syntax

```
#include <pthread.h>
int pthread_cancel (pthread_t thread);
```

Description

The **pthread_cancel** subroutine requests the cancellation of the thread *thread*. The action depends on the cancelability of the target thread:

- If its cancelability is disabled, the cancellation request is set pending.
- If its cancelability is deferred, the cancellation request is set pending till the thread reaches a cancellation point.
- If its cancelability is asynchronous, the cancellation request is acted upon immediately; in some cases, it may result in unexpected behaviour.

The cancellation of a thread terminates it safely, using the same termination procedure as the **pthread_exit** subroutine.

Note: The **pthread.h** header file must be the first included file of each source file using the threads library. Otherwise, the **–D_THREAD_SAFE** compilation flag should be used, or the cc_r compiler used. In this case, the flag is automatically set.

Parameters

thread	Specifies the thread to be canceled.

Return Values

If successful, the **pthread_cancel** function returns zero. Otherwise, an error number is returned to indicate the error.

Error Codes

The **ptread_cancel** function may fail if:

ESRCH	No thread could be found corresponding to that specified by the given thread ID.

The **pthread_cancel** function will not return an error code of EINTR.

Implementation Specifics

This subroutine is part of the Base Operating System (BOS) Runtime.

Related Information

The **pthread_kill** subroutine, **pthread_exit** subroutine, **pthread_join** subroutine, **pthread_cond_wait**, and **pthread_cond_timedwait** subroutines.

The **pthread.h** file.

Terminating Threads in *AIX General Programming Concepts : Writing and Debugging Programs*.

Threads Library Quick Reference in *AIX General Programming Concepts : Writing and Debugging Programs*.

pthread_cleanup_pop or pthread_cleanup_push Subroutine

Purpose

Establishes cancellation handlers.

Library

Threads Library (**libpthreads.a**)

Syntax

```
#include <pthread.h>

void pthread_cleanup_pop (int execute);
void pthread_cleanup_push (void (*routine)(void *), void *arg);
```

Description

The **pthread_cleanup_push** function pushes the specified cancellation cleanup handler *routine* onto the calling thread's cancellation cleanup stack. The cancellation cleanup handler is popped from the cancellation cleanup stack and invoked with the argument *arg* when: (a) the thread exits (that is, calls **pthread_exit**, (b) the thread acts upon a cancellation request, or (c) the thread calls **pthread_cleanup_pop** with a non–zero *execute* argument.

The **pthread_cleanup_pop** function removes the routine at the top of the calling thread's cancellation cleanup stack and optionally invokes it (if *execute* is non–zero).

These functions may be implemented as macros and will appear as statements and in pairs within the same lexical scope (that is, the **pthread_cleanup_push** macro may be thought to expand to a token list whose first token is **'{'** with **pthread_cleanup_pop** expanding to a token list whose last token is the corresponding **'}'**).

The effect of calling **longjmp** or **siglongjmp** is undefined if there have been any calls to **pthread_cleanup_push** or **pthread_cleanup_pop** made without the matching call since the jump buffer was filled. The effect of calling **longjmp** or **siglongjmp** from inside a cancellation cleanup handler is also undefined unless the jump buffer was also filled in the cancellation cleanup handler.

Parameters

execute	Specifies if the popped routine will be executed.

Implementation Specifics

This subroutine is part of the Base Operating System (BOS) Runtime.

Related Information

The **pthread_cancel**, **pthread_setcancelstate** subroutines, the **pthread.h** file.

Terminating Threads in *AIX General Programming Concepts : Writing and Debugging Programs*.

Threads Library Quick Reference in *AIX General Programming Concepts : Writing and Debugging Programs*.

pthread_cond_destroy or pthread_cond_init Subroutine

Purpose

Initialise and destroys condition variables.

Library

Threads Library (**libpthreads.a**)

Syntax

```
#include <pthread.h>

int pthread_cond_init (pthread_cond_t *cond, const
pthread_condattr_t *attr);

int pthread_cond_destroy (pthread_cond_t *cond);
pthread_cond_t cond = PTHREAD_COND_INTITIALIZER;
```

Description

The function **pthread_cond_init** initialises the condition variable referenced by cond with attributes referenced by attr.Ifattr is NULL, the default condition variable attributes are used; the effect is the same as passing the address of a default condition variable attributes object. Upon successful initialisation, the state of the condition variable becomes initialised.

Attempting to initialise an already initialised condition variable results in undefined behaviour.

The function **pthread_cond_destroy** destroys the given condition variable specified by cond; the object becomes, in effect, uninitialised. An implementation may cause **pthread_cond_destroy** to set the object referenced by cond to an invalid value. A destroyed condition variable object can be re–initialised using **pthread_cond_init**; the results of otherwise referencing the object after it has been destroyed are undefined.

It is safe to destroy an initialised condition variable upon which no threads are currently blocked. Attempting to destroy a condition variable upon which other threads are currently blocked results in undefined behaviour.

In cases where default condition variable attributes are appropriate, the macro PTHREAD_COND_INITIALIZER can be used to initialise condition variables that are statically allocated. The effect is equivalent to dynamic initialisation by a call to **pthread_cond_init** with parameter *attr* specified as NULL, except that no error checks are performed.

Return Values

If successful, the **pthread_cond_init** and **pthread_cond_destroy** functions return zero. Otherwise, an error number is returned to indicate the error. The EBUSY and EINVAL error checks, if implemented, act as if they were performed immediately at the beginning of processing for the function and caused an error return prior to modifying the state of the condition variable specified by *cond*.

Error Codes

The **pthread_cond_init** function will fail if:

EAGAIN	The system lacked the necessary resources (other than memory) to initialise another condition variable.
ENOMEM	Insufficient memory exists to initialise the condition variable.

The **pthread_cond_init** function may fail if:

EINVAL	The value specified by *attr* is invalid.

The **pthread_cond_destroy** function may fail if:

EBUSY	The implementation has detected an attempt to destroy the object referenced by cond while it is referenced (for example, while being used in a **pthread_cond_wait** or **pthread_cond_timedwait** by another thread.
EINVAL	The value specified by *cond* is invalid.

These functions will not return an error code of EINTR.

Implementation Specifics

This subroutine is part of the Base Operating System (BOS) Runtime.

Related Information

The **pthread_cond_signal**, **pthread_cond_broadcast**, **pthread_cond_wait**, and **pthread_cond_timewait** subroutines.

The **pthread.h** file.

Using Condition Variables in *AIX General Programming Concepts : Writing and Debugging Programs*.

Threads Library Quick Reference in *AIX General Programming Concepts : Writing and Debugging Programs*.

PTHREAD_COND_INITIALIZER Macro

Purpose

Initializes a static condition variable with default attributes.

Library

Threads Library (**libpthreads.a**)

Syntax

```
#include <pthread.h>

static pthread_cond_t cond = PTHREAD_COND_INITIALIZER;
```

Description

The **PTHREAD_COND_INITIALIZER** macro initializes the static condition variable *cond*, setting its attributes to default values. This macro should only be used for static condition variables, since no error checking is performed.

Note: The **pthread.h** header file must be the first included file of each source file using the threads library. Otherwise, the **–D_THREAD_SAFE** compilation flag should be used, or the cc_r compiler used. In this case, the flag is automatically set.

Implementation Specifics

This macro is part of the Base Operating System (BOS) Runtime.

Related Information

The **pthread_cond_init** subroutine.

Using Condition Variables and Threads Library Quick Reference in *AIX General Programming Concepts : Writing and Debugging Programs*.

pthread_cond_signal or pthread_cond_broadcast Subroutine

Purpose

Unblocks one or more threads blocked on a condition.

Library

Threads Library (**libpthreads.a**)

Syntax

```
#include <pthread.h>

int pthread_cond_signal (condition)
pthread_cond_t *condition;

int pthread_cond_broadcast (condition)
pthread_cond_t *condition;
```

Description

These subroutines unblock one or more threads blocked on the condition specified by *condition*. The **pthread_cond_signal** subroutine unblocks at least one blocked thread, while the **pthread_cond_broadcast** subroutine unblocks all the blocked threads.

If more than one thread is blocked on a condition variable, the scheduling policy determines the order in which threads are unblocked. When each thread unblocked as a result of a **pthread_cond_signal** or **pthread_cond_broadcast** returns from its call to **pthread_cond_wait** or **pthread_cond_timedwait**, the thread owns the mutex with which it called **pthread_cond_wait**or **pthread_cond_timedwait**. The thread(s) that are unblocked contend for the mutex according to the scheduling policy (if applicable), and as if each had called **pthread_mutex_lock**.

The **pthread_cond_signal** or **pthread_cond_broadcast** functions may be called by a thread whether or not it currently owns the mutex that threads calling **pthread_cond_wait** or **pthread_cond_timedwait** have associated with the condition variable during their waits; however, if predictable scheduling behaviour is required, then that mutex is locked by the thread calling **pthread_cond_signal** or **pthread_cond_broadcast**.

If no thread is blocked on the condition, the subroutine succeeds, but the signalling of the condition is not held. The next thread calling **pthread_cond_wait** will be blocked.

Note: The **pthread.h** header file must be the first included file of each source file using the threads library. Otherwise, the **–D_THREAD_SAFE** compilation flag should be used, or the cc_r compiler used. In this case, the flag is automatically set.

Parameter

condition	Specifies the condition to signal.

Return Values

Upon successful completion, 0 is returned. Otherwise, an error code is returned.

Error Code

The **pthread_cond_signal** and **pthread_cond_broadcast** subroutines are unsuccessful if the following is true:

EINVAL	The *condition* parameter is not valid.

Implementation Specifics

These subroutines are part of the Base Operating System (BOS) Runtime.

Related Information

The **pthread_cond_wait** or **pthread_cond_timedwait** subroutine.

Using Condition Variables in *AIX General Programming Concepts : Writing and Debugging Programs*.

Threads Library Quick Reference in *AIX General Programming Concepts : Writing and Debugging Programs*.

pthread_cond_wait or pthread_cond_timedwait Subroutine

Purpose

Blocks the calling thread on a condition.

Library

Threads Library (**libpthreads.a**)

Syntax

```
#include <pthread.h>int pthread_cond_wait (pthread_cond_t *cond);
int pthread_cond_timedwait ( pthread_cond_t *cond,
pthread_mutex_t * mutex, const struct timespec *abstime);
```

Description

The **pthread_cond_wait** and **pthread_cond_timedwait** functions are used to block on a condition variable. They are called with *mutex* locked by the calling thread or undefined behaviour will result.

These functions atomically release *mutex* and cause the calling thread to block on the condition variable cond; atomically here means"atomically with respect to access by another thread to the mutex and then the condition variable''. That is, if another thread is able to acquire the mutex after the about–to–block thread has released it, then a subsequent call to **pthread_cond_signal** or **pthread_cond_broadcast** in that thread behaves as if it were issued after the about–to–block thread has blocked.

Upon successful return, the mutex has been locked and is owned by the calling thread.

When using condition variables there is always a boolean predicate involving shared variables associated with each condition wait that is true if the thread should proceed. Spurious wakeups from the **pthread_cond_wait** or **pthread_cond_timedwait** functions may occur. Since the return from **pthread_cond_wait** or **pthread_cond_timedwait** does not imply anything about the value of this predicate, the predicate should be re–evaluated upon such return.

The effect of using more than one mutex for concurrent **pthread_cond_wait** or **pthread_cond_timedwait** operations on the same condition variable is undefined; that is, a condition variable becomes bound to a unique mutex when a thread waits on the condition variable, and this (dynamic) binding ends when the wait returns.

A condition wait (whether timed or not) is a cancellation point. When the cancelability enable state of a thread is set to PTHREAD_CANCEL_DEFERRED, a side effect of acting upon a cancellation request while in a condition wait is that the mutex is (in effect) re–acquired before calling the first cancellation cleanup handler. The effect is as if the thread were unblocked, allowed to execute up to the point of returning from the call to **pthread_cond_wait** or **pthread_cond_timedwait**, but at that point notices the cancellation request and instead of returning to the caller of **pthread_cond_wait** or **pthread_cond_timedwait**, starts the thread cancellation activities, which includes calling cancellation cleanup handlers.

A thread that has been unblocked because it has been canceled while blocked in a call to **pthread_cond_wait** or **pthread_cond_timedwait** does not consume any condition signal that may be directed concurrently at the condition variable if there are other threads blocked on the condition variable.

The **pthread_cond_timedwait** function is the same as **pthread_cond_wait** except that an error is returned if the absolute time specified by *abstime* passes (that is, system time equals or exceeds *abstime*) before the condition *cond* is signaled or broadcasted, or if the absolute time specified by abstime has already been passed at the time of the call. When such time–outs occur, **pthread_cond_timedwait** will nonetheless release and reacquire the mutex referenced by *mutex*. The function **pthread_cond_timedwait** is also a cancellation point.

If a signal is delivered to a thread waiting for a condition variable, upon return from the signal handler the thread resumes waiting for the condition variable as if it was not interrupted, or it returns zero due to spurious wakeup.

Parameters

condition	Specifies the condition variable to wait on.
mutex	Specifies the mutex used to protect the condition variable. The mutex must be locked when the subroutine is called.
timeout	Points to the absolute time structure specifying the blocked state timeout.

Return Values

Except in the case of ETIMEDOUT, all these error checks act as if they were performed immediately at the beginning of processing for the function and cause an error return, in effect, prior to modifying the state of the mutex specified by mutex or the condition variable specified by cond.

Upon successful completion, a value of zero is returned. Otherwise, an error number is returned to indicate the error.

Error Codes

The **pthread_cond_timedwait** function will fail if:

ETIMEDOUT	The time specified by *abstime* to **pthread_cond_timedwait** has passed.

The **pthread_cond_wait** and **pthread_cond_timedwait** functions may fail if:

EINVAL	The value specified by cond, mutex,orabstime is invalid.
EINVAL	Different mutexes were supplied for concurrent **pthread_cond_wait** or **pthread_cond_timedwait** operations on the same condition variable.
EINVAL	The mutex was not owned by the current thread at the time of the call.

These functions will not return an error code of EINTR.

Implementation Specifics

These subroutines are part of the Base Operating System (BOS) Runtime.

Related Information

The **pthread_cond_signal** or **pthread_cond_broadcast** subroutine, the **pthread.h** file.

Using Condition Variables in *AIX General Programming Concepts : Writing and Debugging Programs*.

Threads Library Quick Reference in *AIX General Programming Concepts : Writing and Debugging Programs*.

pthread_condattr_destroy or pthread_condattr_init Subroutine

Purpose

Initialises and destroys condition variable.

Library

Threads Library (**libpthreads.a**)

Syntax

```
#include <pthread.h>

int pthread_condattr_destroy (pthread_condattr_t *attr);
int pthread_condattr_init (pthread_condattr_t *attr);
```

Description

The function **pthread_condattr_init** initialises a condition variable attributes object *attr* with the default value for all of the attributes defined by the implementation. Attempting to initialise an already initialised condition variable attributes object results in undefined behaviour.

After a condition variable attributes object has been used to initialise one or more condition variables, any function affecting the attributes object (including destruction) does not affect any previously initialised condition variables.

The **pthread_condattr_destroy** function destroys a condition variable attributes object; the object becomes, in effect, uninitialised. The **pthread_condattr_destroy** subroutine may set the object referenced by *attr* to an invalid value. A destroyed condition variable attributes object can be re–initialised using **pthread_condattr_init**; the results of otherwise referencing the object after it has been destroyed are undefined.

Parameter

attr	Specifes the condition attributes object to delete.

Return Values

If successful, the **pthread_condattr_init** and **pthread_condattr_destroy** functions return zero. Otherwise, an error number is returned to indicate the error.

Error Code

The **pthread_condattr_init** function will fail if:

ENOMEM	Insufficient memory exists to initialise the condition variable attributes object.

The **pthread_condattr_destroy** function may fail if:

EINVAL	The value specified by attr is invalid.

These functions will not return an error code of EINTR.

Implementation Specifics

This subroutine is part of the Base Operating System (BOS) Runtime.

Related Information

The **pthread_cond_init** subroutine, the **pthread_condattr_getpshared**, the **pthread_create**, the **pthread_mutex_init**, the **pthread.h** file.

Using Condition Variables in *AIX General Programming Concepts : Writing and Debugging Programs*.

Threads Library Quick Reference in *AIX General Programming Concepts : Writing and Debugging Programs*.

pthread_condattr_getpshared Subroutine

Purpose

Returns the value of the pshared attribute of a condition attributes object.

Library

Threads Library (**libpthreads.a**)

Syntax

```
#include <pthread.h>

int pthread_condattr_getpshared (attr, pshared)
const pthread_condattr_t *attr;
int *pshared;
```

Description

The **pthread_condattr_getpshared** subroutine returns the value of the pshared attribute of the condition attribute object *attr*. This attribute specifies the process sharing of the condition variable created with this attributes object. It may have one of the following values:

PTHREAD_PROCESS_ SHARED	Specifies that the condition variable can be used by any thread that has access to the memory where it is allocated, even if these threads belong to different processes.
PTHREAD_PROCESS_ PRIVATE	Specifies that the condition variable shall only be used by threads within the same process as the thread that created it. This is the default value.

Note: The **pthread.h** header file must be the first included file of each source file using the threads library. Otherwise, the **–D_THREAD_SAFE** compilation flag should be used, or the cc_r compiler used. In this case, the flag is automatically set.

Parameters

attr	Specifies the condition attributes object.
pshared	Points to where the pshared attribute value will be stored.

Return Values

Upon successful completion, the value of the pshared attribute is returned via the *pshared* parameter, and 0 is returned. Otherwise, an error code is returned.

Error Codes

The **pthread_condattr_getpshared** subroutine is unsuccessful if the following is true:

EINVAL	The *attr* parameter is not valid.
ENOSYS	The process sharing POSIX option is not implemented.

Implementation Specifics

This subroutine is part of the Base Operating System (BOS) Runtime.

Related Information

The **pthread_condattr_setpshared** subroutine, **pthread_condattr_init** subroutine.

Advanced Attributes in *AIX General Programming Concepts : Writing and Debugging Programs*.

Threads Library Options in *AIX General Programming Concepts : Writing and Debugging Programs*.

Threads Library Quick Reference in *AIX General Programming Concepts : Writing and Debugging Programs*.

pthread_condattr_setpshared Subroutine

Purpose

Sets the value of the pshared attribute of a condition attributes object.

Library

Threads Library (**libpthreads.a**)

Syntax

```
#include <pthread.h>

int pthread_condattr_setpshared (attr, pshared)
pthread_condattr_t *attr;
int pshared;
```

Description

The **pthread_condattr_setpshared** subroutine sets the value of the pshared attribute of the condition attributes object *attr*. This attribute specifies the process sharing of the condition variable created with this attributes object.

Note: The **pthread.h** header file must be the first included file of each source file using the threads library. Otherwise, the **–D_THREAD_SAFE** compilation flag should be used, or the cc_r compiler used. In this case, the flag is automatically set.

Parameters

attr	Specifies the condition attributes object.
pshared	Specifies the process sharing to set. It must have one of the following values:

PTHREAD_PROCESS_SHARED
Specifies that the condition variable can be used by any thread that has access to the memory where it is allocated, even if these threads belong to different processes.

PTHREAD_PROCESS_PRIVATE
Specifies that the condition variable shall only be used by threads within the same process as the thread that created it. This is the default value.

Return Values

Upon successful completion, 0 is returned. Otherwise, an error code is returned.

Error Codes

The **pthread_condattr_setpshared** subroutine is unsuccessful if the following is true:

EINVAL	The *attr* or *pshared* parameters are not valid.

Implementation Specifics

This subroutine is part of the Base Operating System (BOS) Runtime.

Related Information

The **pthread_condattr_getpshared** subroutine, **pthread_condattr_init** subroutine, **pthread_cond_init** subroutine.

Advanced Attributes in *AIX General Programming Concepts : Writing and Debugging Programs*.

Threads Library Options in *AIX General Programming Concepts : Writing and Debugging Programs*.

Threads Library Quick Reference in *AIX General Programming Concepts : Writing and Debugging Programs*.

pthread_create Subroutine

Purpose

Creates a new thread, initializes its attributes, and makes it runnable.

Library

Threads Library (**libpthreads.a**)

Syntax

```
#include <pthread.h>int pthread_create( pthread_t * thread, const
pthread_attr_t * attr, void *(* start_routine) (void), void *
arg);
```

Description

The **pthread_create** subroutine creates a new thread and initializes its attributes using the thread attributes object specified by the *attr* parameter. The new thread inherits its creating thread's signal mask; but any pending signal of the creating thread will be cleared for the new thread.

Note: The number of threads per process is defined in the **pthread.h** file as 512.

The new thread is made runnable, and will start executing the *start_routine* routine, with the parameter specified by the *arg* parameter. The *arg* parameter is a void pointer; it can reference any kind of data. It is not recommended to cast this pointer into a scalar data type (**int** for example), because the casts may not be portable.

After thread creation, the thread attributes object can be reused to create another thread, or deleted.

The thread terminates in the following cases:

- The thread returned from its starting routine (the **main** routine for the initial thread)
- The thread called the **pthread_exit** subroutine
- The thread was canceled
- The thread received a signal that terminated it
- The entire process is terminated due to a call to either the **exec** or **exit** subroutines.

Note: The **pthread.h** header file must be the first included file of each source file using the threads library. Otherwise, the **–D_THREAD_SAFE** compilation flag should be used, or the cc_r compiler used. In this case, the flag is automatically set.

Parameters

thread	Points to where the thread ID will be stored.
attr	Specifies the thread attributes object to use in creating the thread. If the value is **NULL**, the default attributes values will be used.
start_routine	Points to the routine to be executed by the thread.
arg	Points to the single argument to be passed to the *start_routine* routine.

Return Values

If successful, the **pthread_create** function returns zero. Otherwise, an error number is returned to indicate the error.

Error Codes

The **pthread_create** function will fail if:

EAGAIN	The system lacked the necessary resources to create another thread, or the system–imposed limit on the total number of threads in a process PTHREAD_THREADS_MAX would be exceeded.
EINVAL	The value specified by **attr** is invalid.
EPERM	The caller does not have appropriate permission to set the required scheduling parameters or scheduling policy.

The **pthread_create** function will not return an error code of **EINTR**.

Implementation Specifics

This subroutine is part of the Base Operating System (BOS) Runtime.

Related Information

The **pthread_attr_init** subroutine, **pthread_attr_destroy** subroutine, **pthread_exit** subroutine, **pthread_cancel** subroutine, **pthread_kill** subroutine, **pthread_self** subroutine, **pthread_once** subroutine, **pthread_join** subroutine, **fork** subroutine, and the **pthread.h** file.

Creating Threads in *AIX General Programming Concepts : Writing and Debugging Programs*.

Threads Library Quick Reference in *AIX General Programming Concepts : Writing and Debugging Programs*.

pthread_delay_np Subroutine

Purpose

Causes a thread to wait for a specified period.

Library

Threads Library (**libpthreads.a**)

Syntax

```
#include <pthread.h>

int pthread_delay_np (interval)
struct timespec *interval;
```

Description

The **pthread_delay_np** subroutine causes the calling thread to delay execution for a specified period of elapsed wall clock time. The period of time the thread waits is at least as long as the number of seconds and nanoseconds specified in the *interval* parameter.

Notes:

1. The **pthread.h** header file must be the first included file of each source file using the threads library. Otherwise, the **–D_THREAD_SAFE** compilation flag should be used, or the cc_r compiler used. In this case, the flag is automatically set.
2. The **pthread_delay_np** subroutine is not portable.

Parameters

interval	Points to the time structure specifying the wait period.

Return Values

Upon successful completion, 0 is returned. Otherwise, an error code is returned.

Error Codes

The **pthread_delay_np** subroutine is unsuccessful if the following is true:

EINVAL	The *interval* parameter is not valid.

Implementation Specifics

This subroutine is part of the Base Operating System (BOS) Runtime.

This subroutine is not POSIX compliant and is provided only for compatibility with DCE threads. It should not be used when writing new applications.

Related Information

The **sleep**, **nsleep**, or **usleep** subroutine.

pthread_equal Subroutine

Purpose

Compares two thread IDs.

Library

Threads Library (**libpthreads.a**)

Syntax

```
#include <pthread.h>

int pthread_equal (pthread_t t1, pthread_t t2);
```

Description

The **pthread_equal** subroutine compares the thread IDs *thread1* and *thread2*. Since the thread IDs are opaque objects, it should not be assumed that they can be compared using the equality operator (==).

Note: The **pthread.h** header file must be the first included file of each source file using the threads library. Otherwise, the **–D_THREAD_SAFE** compilation flag should be used, or the cc_r compiler used. In this case, the flag is automatically set.

Parameters

thread1	Specifies the first ID to be compared.
thread2	Specifies the second ID to be compared.

Return Values

The **pthread_equal** function returns a non–zero value if *t1* and t2 are equal; otherwise, zero is returned.

If either *t1* or *t2* are not valid thread IDs, the behaviour is undefined.

Implementation Specifics

This subroutine is part of the Base Operating System (BOS) Runtime.

Related Information

The **pthread_self** subroutine, the **pthread_create** subroutine, the **pthread.h** file.

Creating Threads in *AIX General Programming Concepts : Writing and Debugging Programs*.

Threads Library Quick Reference in *AIX General Programming Concepts : Writing and Debugging Programs*.

pthread_exit Subroutine

Purpose

Terminates the calling thread.

Library

Threads Library (**libpthreads.a**)

Syntax

```
#include <pthread.h>

void pthread_exit (void *value_ptr);
```

Description

The **pthread_exit** subroutine terminates the calling thread safely, and stores a termination status for any thread that may join the calling thread. The termination status is always a void pointer; it can reference any kind of data. It is not recommended to cast this pointer into a scalar data type (**int** for example), because the casts may not be portable. This subroutine never returns.

Unlike the **exit** subroutine, the **pthread_exit** subroutine does not close files. Thus any file opened and used only by the calling thread must be closed before calling this subroutine. It is also important to note that the **pthread_exit** subroutine frees any thread–specific data, including the thread's stack. Any data allocated on the stack becomes invalid, since the stack is freed and the corresponding memory may be reused by another thread. Therefore, thread synchronization objects (mutexes and condition variables) allocated on a thread's stack must be destroyed before the thread calls the **pthread_exit** subroutine.

Returning from the initial routine of a thread implicitly calls the **pthread_exit** subroutine, using the return value as parameter.

If the thread is not detached, its resources, including the thread ID, the termination status, the thread–specific data, and its storage, are all maintained until the thread is detached or the process terminates.

If another thread joins the calling thread, that thread wakes up immediately, and the calling thread is automatically detached.

If the thread is detached, the cleanup routines are popped from their stack and executed. Then the destructor routines from the thread–specific data are executed. Finally, the storage of the thread is reclaimed and its ID is freed for reuse.

Terminating the initial thread by calling this subroutine does not terminate the process, it just terminates the initial thread. However, if all the threads in the process are terminated, the process is terminated by implicitly calling the **exit** subroutine with a return code of 0 if the last thread is detached, or 1 otherwise.

Note: The **pthread.h** header file must be the first included file of each source file using the threads library. Otherwise, the **–D_THREAD_SAFE** compilation flag should be used, or the cc_r compiler used. In this case, the flag is automatically set.

Parameters

status	Points to an optional termination status, used by joining threads. If no termination status is desired, its value should be **NULL**.

Return Values

The **pthread_exit** function cannot return to its caller.

Errors No errors are defined.

The **pthread_exit** function will not return an error code of EINTR.

Implementation Specifics

This subroutine is part of the Base Operating System (BOS) Runtime.

Related Information

The **pthread_cleanup_push** subroutine, **pthread_cleanup_pop** subroutine, **pthread_key_create** subroutine, **pthread_create** subroutine, **pthread_join** subroutine, **pthread_cancel** subroutine, **exit** subroutine, the **pthread.h** file.

Terminating Threads and Threads Library Quick Reference in *AIX General Programming Concepts : Writing and Debugging Programs*.

pthread_get_expiration_np Subroutine

Purpose

Obtains a value representing a desired expiration time.

Library

Threads Library (**libpthreads.a**)

Syntax

```
#include <pthread.h>

int pthread_get_expiration_np (delta, abstime)
struct timespec *delta;
struct timespec *abstime;
```

Description

The **pthread_get_expiration_np** subroutine adds the interval *delta* to the current absolute system time and returns a new absolute time. This new absolute time can be used as the expiration time in a call to the **pthread_cond_timedwait** subroutine.

Notes:

1. The **pthread.h** header file must be the first included file of each source file using the threads library. Otherwise, the **–D_THREAD_SAFE** compilation flag should be used, or the cc_r compiler used. In this case, the flag is automatically set.
2. The **pthread_get_expiration_np** subroutine is not portable.

Parameters

delta	Points to the time structure specifying the interval.
abstime	Points to where the new absolute time will be stored.

Return Values

Upon successful completion, the new absolute time is returned via the *abstime* parameter, and 0 is returned. Otherwise, an error code is returned.

Error Codes

The **pthread_get_expiration_np** subroutine is unsuccessful if the following is true:

EINVAL	The *delta* or *abstime* parameters are not valid.

Implementation Specifics

This subroutine is part of the Base Operating System (BOS) Runtime.

This subroutine is not POSIX compliant and is provided only for compatibility with DCE threads. It should not be used when writing new applications.

Related Information

The **pthread_cond_timedwait** subroutine.

pthread_getconcurrency or pthread_setconcurrency Subroutine

Purpose

Gets or sets level of concurrency.

Library

Threads Library (**libthreads.a**)

Syntax

```
#include <pthread.h>

int pthread_getconcurrency (void);
int pthread_setconcurrency (int new_level);
```

Description

The pthread_setconcurrency function allows an application to inform the threads implementation of its desired concurrency level, new_level. The actual level of concurrency provided by the implementation as a result of this function call is unspecified.

If new_level is zero, it causes the implementation to maintain the concurrency level at its discretion as if pthread_setconcurrency was never called.

The pthread_getconcurrency function returns the value set by a previous call to the pthread_setconcurrency function. If the pthread_setconcurrency function was not previously called, this function returns zero to indicate that the implementation is maintaining the concurrency level.

When an application calls pthread_setconcurrency it is informing the implementation of its desired concurrency level. The implementation uses this as a hint, not a requirement.

Return Value

If successful, the pthread_setconcurrency function returns zero. Otherwise, an error number is returned to indicate the error.

The pthread_getconcurrency function always returns the concurrency level set by a previous call to pthread_setconcurrency. If the pthread_setconcurrency function has never been called, pthread_getconcurrency returns zero.

Error Codes

The pthread_setconcurrency function will fail if:

EINVAL	The value specified by new_level is negative.
EAGAIN	The value specific by new_level would cause a system resource to be exceeded.

Implementation Specifics

Use of these functions changes the state of the underlying concurrency upon which the application depends. Library developers are advised to not use the pthread_getconcurrency and pthread_setconcurrency functions since their use may conflict with an applications use of these functions.

Related Information

The **pthread.h** file.

pthread_getschedparam Subroutine

Purpose

Returns the current schedpolicy and schedparam attributes of a thread.

Library

Threads Library (**libpthreads.a**)

Syntax

```
#include <pthread.h>
#include <sys/sched.h>

int pthread_getschedparam (thread, schedpolicy, schedparam)
pthread_t thread;
int *schedpolicy;
struct sched_param *schedparam;
```

Description

The **pthread_getschedparam** subroutine returns the current schedpolicy and schedparam attributes of the thread *thread*. The schedpolicy attribute specifies the scheduling policy of a thread. It may have one of the following values:

SCHED_FIFO	Denotes first–in first–out scheduling.
SCHED_RR	Denotes round–robin scheduling.
SCHED_OTHER	Denotes the default AIX scheduling policy. It is the default value.

The schedparam attribute specifies the scheduling parameters of a thread created with this attributes object. The `sched_priority` field of the **sched_param** structure contains the priority of the thread. It is an integer value.

Note: The **pthread.h** header file must be the first included file of each source file using the threads library. Otherwise, the **–D_THREAD_SAFE** compilation flag should be used, or the cc_r compiler used. In this case, the flag is automatically set.

Parameters

thread	Specifies the target thread.
schedpolicy	Points to where the schedpolicy attribute value will be stored.
schedparam	Points to where the schedparam attribute value will be stored.

Return Values

Upon successful completion, the current value of the schedpolicy and schedparam attributes are returned via the *schedpolicy* and *schedparam* parameters, and 0 is returned. Otherwise, an error code is returned.

Error Codes

The **pthread_getschedparam** subroutine is unsuccessful if the following is true:

ESRCH	The thread *thread* does not exist.

Implementation Specifics

This subroutine is part of the Base Operating System (BOS) Runtime.

The implementation of this subroutine is dependent on the priority scheduling POSIX option. The priority scheduling POSIX option is implemented in AIX.

Related Information

The **pthread_attr_getschedparam** subroutine.

Threads Scheduling in *AIX General Programming Concepts : Writing and Debugging Programs*.

Threads Library Options in *AIX General Programming Concepts : Writing and Debugging Programs*.

Threads Library Quick Reference in *AIX General Programming Concepts : Writing and Debugging Programs*.

pthread_getspecific or pthread_setspecific Subroutine

Purpose

Returns and sets the thread–specific data associated with the specified key.

Library

Threads Library (**libpthreads.a**)

Syntax

```
#include <pthread.h>

void *pthread_getspecific (key)
pthread_key_t key;

void *pthread_setspecific (key, value)
pthread_key_t key;
const void *value;
```

Description

The **pthread_setspecific** function associates a thread–specific *value* with a *key* obtained via a previous call to **pthread_key_create**. Different threads may bind different values to the same key. These values are typically pointers to blocks of dynamically allocated memory that have been reserved for use by the calling thread.

The **pthread_getspecific** function returns the value currently bound to the specified *key* on behalf of the calling thread.

The effect of calling **pthread_setspecific** or **pthread_getspecific** with a *key* value not obtained from **pthread_key_create** or after key has been deleted with **pthread_key_delete** is undefined.

Both **pthread_setspecific** and **pthread_getspecific** may be called from a thread–specific data destructor function. However, calling **pthread_setspecific** from a destructor may result in lost storage or infinite loops.

Parameters

key	Specifies the key to which the value is bound.
value	Specifies the new thread–specific value.

Return Values

The function **pthread_getspecific** returns the thread–specific data value associated with the given key. If no thread–specific data value is associated with key, then the value NULL is returned. If successful, the **pthread_setspecific** function returns zero. Otherwise, an error number is returned to indicate the error.

Error Codes

The **pthread_setspecific** function will fail if:

ENOMEM	Insufficient memory exists to associate the value with the key.

The **pthread_setspecific** function may fail if:

EINVAL	The key value is invalid.

No errors are returned from **pthread_getspecific**.

These functions will not return an error code of EINTR.

Implementation Specifics

These subroutines are part of the Base Operating System (BOS) Runtime.

Related Information

The **pthread_key_create** subroutine, the **pthread.h** file.

Thread–Specific Data in *AIX General Programming Concepts : Writing and Debugging Programs*.

Threads Library Quick Reference in *AIX General Programming Concepts : Writing and Debugging Programs*.

pthread_getunique_np Subroutine

Purpose

Returns the sequence number of a thread

Library

Threads Library (**libpthreads.a**)

Syntax

```
#include <pthread.h>

int pthread_getunique_np (thread, sequence)
pthread_t *thread;
int *sequence;
```

Description

The **pthread_getunique_np** subroutine returns the sequence number of the thread *thread*. The sequence number is a number, unique to each thread, associated with the thread at creation time.

Notes:

1. The **pthread.h** header file must be the first included file of each source file using the threads library. Otherwise, the **–D_THREAD_SAFE** compilation flag should be used, or the cc_r compiler used. In this case, the flag is automatically set.
2. The **pthread_getunique_np** subroutine is not portable.

Parameters

thread	Specifies the thread.
sequence	Points to where the sequence number will be stored.

Return Values

Upon successful completion, the sequence number is returned via the *sequence* parameter, and 0 is returned. Otherwise, an error code is returned.

Error Codes

The **pthread_getunique_np** subroutine is unsuccessful if the following is true:

EINVAL	The *thread* or *sequence* parameters are not valid.
ESRCH	The thread *thread* does not exist.

Implementation Specifics

This subroutine is part of the Base Operating System (BOS) Runtime.

This subroutine is not POSIX compliant and is provided only for compatibility with DCE threads. It should not be used when writing new applications.

Related Information

The **pthread_self** subroutine.

pthread_join, or pthread_detach Subroutine

Purpose

Blocks the calling thread until the specified thread terminates.

Library

Threads Library (**libpthreads.a**)

Syntax

```
#include <pthread.h>

int pthread_join (pthread_t thread, void
**value_ptr);
int pthread_detach (pthread_t thread;
**value_ptr);
```

Description

The **pthread_join** subroutine blocks the calling thread until the thread *thread* terminates. The target thread's termination status is returned in the *status* parameter.

If the target thread is already terminated, but not yet detached, the subroutine returns immediately. It is impossible to join a detached thread, even if it is not yet terminated. The target thread is automatically detached after all joined threads have been woken up.

This subroutine does not itself cause a thread to be terminated. It acts like the **pthread_cond_wait** subroutine to wait for a special condition.

Note: The **pthread.h** header file must be the first included file of each source file using the threads library. Otherwise, the **–D_THREAD_SAFE** compilation flag should be used, or the cc_r compiler used. In this case, the flag is automatically set.

The **pthread_detach** subroutine is used to indicate to the implementation that storage for the thread whose thread ID is in the location *thread* can be reclaimed when that thread terminates. This storage shall be reclaimed on process exit, regardless of whether the thread has been detached or not, and may include storage for *thread* return value. If *thread* has not yet terminated, **pthread_detach** shall not cause it to terminate. Multiple **pthread_detach** calls on the same target thread causes an error.

Parameters

thread	Specifies the target thread.
status	Points to where the termination status of the target thread will be stored. If the value is **NULL**, the termination status is not returned.

Return Values

If successful, the **pthread_join** function returns zero. Otherwise, an error number is returned to indicate the error.

Error Codes

The **pthread_join** and **pthread_detach** functions will fail if:

EINVAL	The implementation has detected that the value specified by thread does not refer to a joinable thread.
ESRCH	No thread could be found corresponding to that specified by the given thread ID.

The **pthread_join** function will fail if:

EDEADLK	The value of thread specifies the calling thread.

The **pthread_join** function will not return an error code of EINTR.

Implementation Specifics

This subroutine is part of the Base Operating System (BOS) Runtime.

Related Information

The **pthread_exit** subroutine, **pthread_create** subroutine, **wait** subroutine, **pthread_cond_wait** or **pthread_cond_timedwait** subroutines, the **pthread.h** file.

Joining Threads in *AIX General Programming Concepts : Writing and Debugging Programs*.

Threads Library Quick Reference in *AIX General Programming Concepts : Writing and Debugging Programs*.

pthread_key_create Subroutine

Purpose

Creates a thread–specific data key.

Library

Threads Library (**libpthreads.a**)

Syntax#include <pthread.h>

int pthread_key_create (*key*,*destructor* **)**
pthread_key_t * *key***;**
void (* *destructor***) (void *);**

Description

The **pthread_key_create** subroutine creates a thread–specific data key. The key is shared among all threads within the process, but each thread has specific data associated with the key. The thread–specific data is a void pointer, initially set to **NULL**.

The application is responsible for ensuring that this subroutine is called only once for each requested key. This can be done, for example, by calling the subroutine before creating other threads, or by using the one–time initialization facility.

Typically, thread–specific data are pointers to dynamically allocated storage. When freeing the storage, the value should be set to **NULL**. It is not recommended to cast this pointer into scalar data type (**int** for example), because the casts may not be portable, and because the value of **NULL** is implementation dependent.

An optional destructor routine can be specified. It will be called for each thread when it is terminated and detached, after the call to the cleanup routines, if the specific value is not **NULL**. Typically, the destructor routine will release the storage thread–specific data. It will receive the thread–specific data as a parameter.

Note: The **pthread.h** header file must be the first included file of each source file using the threads library. Otherwise, the **–D_THREAD_SAFE** compilation flag should be used, or the cc_r compiler used. In this case, the flag is automatically set.

Parameters

key	Points to where the key will be stored.
destructor	Points to an optional destructor routine, used to cleanup data on thread termination. If no cleanup is desired, this pointer should be **NULL**.

Return Values

If successful, the **pthread_key_create** function stores the newly created key value at **key* and returns zero. Otherwise, an error number is returned to indicate the error.

Error Codes

The **pthread_key_create** function will fail if:

EAGAIN	The system lacked the necessary resources to create another thread–specific data key, or the system–imposed limit on the total number of keys per process PTHREAD_KEYS_MAX has been exceeded.
ENOMEM	Insufficient memory exists to create the key.

The **pthread_key_create** function will not return an error code of EINTR.

Implementation Specifics

This subroutine is part of the Base Operating System (BOS) Runtime.

Related Information

The **pthread_exit** subroutine, **pthread_key_delete** subroutine, **pthread_getspecific** subroutne, **pthread_once** subroutine, **pthread.h** file.

Thread–Specific Data in *AIX General Programming Concepts : Writing and Debugging Programs*.

Threads Library Quick Reference in *AIX General Programming Concepts : Writing and Debugging Programs*.

pthread_key_delete Subroutine

Purpose

Deletes a thread–specific data key.

Library

Threads Library (**libpthreads.a**)

Syntax

```
#include <pthread.h>

int pthread_key_delete (pthread_key_t key);
```

Description

The **pthread_key_delete** subroutine deletes the thread–specific data key *key*, previously created with the **pthread_key_create** subroutine. The application must ensure that no thread–specific data is associated with the key. No destructor routine is called.

Note: The **pthread.h** header file must be the first included file of each source file using the threads library. Otherwise, the **–D_THREAD_SAFE** compilation flag should be used, or the cc_r compiler used. In this case, the flag is automatically set.

Parameters

key	Specifies the key to delete.

Return Values

If successful, the **pthread_key_delete** function returns zero. Otherwise, an error number is returned to indicate the error.

Error Codes

The **pthread_key_delete** function will fail if:

EINVAL	The key value is invalid.

The **pthread_key_delete** function will not return an error code of EINTR.

Implementation Specifics

This subroutine is part of the Base Operating System (BOS) Runtime.

Related Information

The **pthread_key_create** subroutine, **pthread.h** file.

Thread–Specific Data in *AIX General Programming Concepts : Writing and Debugging Programs*.

Threads Library Quick Reference in *AIX General Programming Concepts : Writing and Debugging Programs*.

pthread_kill Subroutine

Purpose

Sends a signal to the specified thread.

Library

Threads Library (**libpthreads.a**)

Syntax

```
#include <signal.h>

int pthread_kill (pthread_t thread, int sig);
```

Description

The **pthread_kill** subroutine sends the signal *signal* to the thread *thread*. It acts with threads like the **kill** subroutine with single–threaded processes.

If the receiving thread has blocked delivery of the signal, the signal remains pending on the thread until the thread unblocks delivery of the signal or the action associated with the signal is set to ignore the signal.

Note: The **pthread.h** header file must be the first included file of each source file using the threads library. Otherwise, the **–D_THREAD_SAFE** compilation flag should be used, or the cc_r compiler used. In this case, the flag is automatically set.

Parameters

thread	Specifies the target thread for the signal.
signal	Specifies the signal to be delivered. If the signal value is 0, error checking is performed, but no signal is delivered.

Return Values

Upon successful completion, the function returns a value of zero. Otherwise the function returns an error number. If the **pthread_kill** function fails, no signal is sent.

Error Codes

The **pthread_kill** function will fail if:

ESRCH	No thread could be found corresponding to that specified by the given thread ID.
EINVAL	The value of the sig argument is an invalid or unsupported signal number.

The **pthread_kill** function will not return an error code of EINTR.

Implementation Specifics

This subroutine is part of the Base Operating System (BOS) Runtime.

Related Information

The **kill** subroutine, **pthread_cancel** subroutine, **pthread_create** subroutine, **sigaction** subroutine, **pthread_self** subroutine, **raise** subroutine, **pthread.h** file.

Signal Management in *AIX General Programming Concepts : Writing and Debugging Programs*.

Threads Library Quick Reference in *AIX General Programming Concepts : Writing and Debugging Programs*.

pthread_lock_global_np Subroutine

Purpose

Locks the global mutex.

Library

Threads Library (**libpthreads.a**)

Syntax

```
#include <pthread.h>

void pthread_lock_global_np ()
```

Description

The **pthread_lock_global_np** subroutine locks the global mutex. If the global mutex is currently held by another thread, the calling thread waits until the global mutex is unlocked. The subroutine returns with the global mutex locked by the calling thread.

Use the global mutex when calling a library package that is not designed to run in a multithreaded environment. (Unless the documentation for a library function specifically states that it is compatible with multithreading, assume that it is not compatible; in other words, assume it is nonreentrant.)

The global mutex is one lock. Any code that calls any function that is not known to be reentrant uses the same lock. This prevents dependencies among threads calling library functions and those functions calling other functions, and so on.

The global mutex is a recursive mutex. A thread that has locked the global mutex can relock it without deadlocking. The thread must then call the **pthread_unlock_global_np** subroutine as many times as it called this routine to allow another thread to lock the global mutex.

Notes:

1. The **pthread.h** header file must be the first included file of each source file using the threads library. Otherwise, the **–D_THREAD_SAFE** compilation flag should be used, or the cc_r compiler used. In this case, the flag is automatically set.
2. The **pthread_lock_global_np** subroutine is not portable.

Implementation Specifics

This subroutine is part of the Base Operating System (BOS) Runtime.

This subroutine is not POSIX compliant and is provided only for compatibility with DCE threads. It should not be used when writing new applications.

Related Information

The **pthread_mutex_lock** subroutine, **pthread_unlock_global_np** subroutine.

Using Mutexes in *AIX General Programming Concepts : Writing and Debugging Programs*.

pthread_mutex_init or pthread_mutex_destroy Subroutine

Purpose

Initialises or destroys a mutex.

Library

Threads Library (**libpthreads.a**)

Syntax

```
#include <pthread.h>
int pthread_mutex_init (pthread_mutex_t *mutex, const
pthread_mutexattr_t *attr);
int pthread_mutex_destroy (pthread_mutex_t *mutex);
pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;
```

Description

The **pthread_mutex_init** function initialises the mutex referenced by *mutex* with attributes specified by *attr*. If *attr* is NULL, the default mutex attributes are used; the effect is the same as passing the address of a default mutex attributes object. Upon successful initialisation, the state of the mutex becomes initialised and unlocked.

Attempting to initialise an already initialised mutex results in undefined behaviour.

The **pthread_mutex_destroy** function destroys the mutex object referenced by mutex; the mutex object becomes, in effect, uninitialised. An implementation may cause **pthread_mutex_destroy** to set the object referenced by *mutex* to an invalid value. A destroyed mutex object can be re–initialised using **pthread_mutex_init**; the results of otherwise referencing the object after it has been destroyed are undefined.

It is safe to destroy an initialised mutex that is unlocked. Attempting to destroy a locked mutex results in undefined behaviour.

In cases where default mutex attributes are appropriate, the macro PTHREAD_MUTEX_INITIALIZER can be used to initialise mutexes that are statically allocated. The effect is equivalent to dynamic initialisation by a call to **pthread_mutex_init** with parameter *attr* specified as NULL, except that no error checks are performed.

Parameters

mutex	Specifies the mutex to delete.

Return Values

If successful, the **pthread_mutex_init** and **pthread_mutex_destroy** functions return zero. Otherwise, an error number is returned to indicate the error. The EBUSY and EINVAL error checks act as if they were performed immediately at the beginning of processing for the function and cause an error return prior to modifying the state of the mutex specified by *mutex*.

Error Codes

The **pthread_mutex_init** function will fail if:

ENOMEM	Insufficient memory exists to initialise the mutex.
EINVAL	The value specified by *attr* is invalid.

The **pthread_mutex_destroy** function will fail if:

EBUSY	The implementation has detected an attempt to destroy the object referenced by *mutex* while it is locked or referenced (for example, while being used in a **pthread_cond_wait** or **pthread_cond_timedwait** by another thread.
EINVAL	The value specified by *mutex* is invalid.

These functions will not return an error code of EINTR.

Implementation Specifics

This subroutine is part of the Base Operating System (BOS) Runtime.

Related Information

The **pthread_mutex_lock**, **pthread_mutex_unlock**, **pthread_mutex_trylock**, **pthread_mutexattr_setpshared** subroutines, the **pthread.h** file.

PTHREAD_MUTEX_INITIALIZER Macro

Purpose

Initializes a static mutex with default attributes.

Library

Threads Library (**libpthreads.a**)

Syntax

```
#include <pthread.h>

static pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;
```

Description

The **PTHREAD_MUTEX_INITIALIZER** macro initializes the static mutex *mutex*, setting its attributes to default values. This macro should only be used for static mutexes, as no error checking is performed.

Note: The **pthread.h** header file must be the first included file of each source file using the threads library. Otherwise, the **–D_THREAD_SAFE** compilation flag should be used, or the cc_r compiler used. In this case, the flag is automatically set.

Implementation Specifics

This macro is part of the Base Operating System (BOS) Runtime.

Related Information

The **pthread_mutex_init** subroutine.

Using Mutexes in *AIX General Programming Concepts : Writing and Debugging Programs*.

Threads Library Quick Reference in *AIX General Programming Concepts : Writing and Debugging Programs*.

pthread_mutex_lock, pthread_mutex_trylock, or pthread_mutex_unlock Subroutine

Purpose

Locks and unlocks a mutex.

Library

Threads Library (**libpthreads.a**)

Syntax

```
#include <pthread.h>
```

```
int pthread_mutex_lock (mutex)
pthread_mutex_t *mutex;
```

```
int pthread_mutex_trylock (mutex)
pthread_mutex_t *mutex;
```

```
int pthread_mutex_unlock (mutex)
pthread_mutex_t *mutex;
```

Description

The mutex object referenced by *mutex* is locked by calling **pthread_mutex_lock**. If the mutex is already locked, the calling thread blocks until the mutex becomes available. This operation returns with the mutex object referenced by mutex in the locked state with the calling thread as its owner.

If the mutex type is PTHREAD_MUTEX_NORMAL, deadlock detection is not provided. Attempting to relock the mutex causes deadlock. If a thread attempts to unlock a mutex that it has not locked or a mutex which is unlocked, undefined behaviour results.

If the mutex type is PTHREAD_MUTEX_ERRORCHECK, then error checking is provided. If a thread attempts to relock a mutex that it has already locked, an error will be returned. If a thread attempts to unlock a mutex that it has not locked or a mutex which is unlocked, an error will be returned.

If the mutex type is PTHREAD_MUTEX_RECURSIVE, then the mutex maintains the concept of a lock count. When a thread successfully acquires a mutex for the first time, the lock count is set to one. Every time a thread relocks this mutex, the lock count is incremented by one. Each time the thread unlocks the mutex, the lock count is decremented by one. When the lock count reaches zero, the mutex becomes available for other threads to acquire. If a thread attempts to unlock a mutex that it has not locked or a mutex which is unlocked, an error will be returned.

If the mutex type is PTHREAD_MUTEX_DEFAULT, attempting to recursively lock the mutex results in undefined behaviour. Attempting to unlock the mutex if it was not locked by the calling thread results in undefined behaviour. Attempting to unlock the mutex if it is not locked results in undefined behaviour.

The function **pthread_mutex_trylock** is identical to **pthread_mutex_lock** except that if the mutex object referenced by *mutex* is currently locked (by any thread, including the current thread), the call returns immediately.

The **pthread_mutex_unlock** function releases the mutex object referenced by mutex. The manner in which a mutex is released is dependent upon the mutex's type attribute. If there are threads blocked on the mutex object referenced by *mutex* when **pthread_mutex_unlock** is called, resulting in the mutex becoming available, the scheduling policy is used to determine which thread shall acquire the mutex. (In the case of PTHREAD_MUTEX_RECURSIVE mutexes, the mutex becomes available when the count reaches zero and the calling thread no longer has any locks on this mutex).

If a signal is delivered to a thread waiting for a mutex, upon return from the signal handler the thread resumes waiting for the mutex as if it was not interrupted.

Parameter

mutex	Specifies the mutex to lock.

Return Values

If successful, the **pthread_mutex_lock** and **pthread_mutex_unlock** functions return zero. Otherwise, an error number is returned to indicate the error.

The function **pthread_mutex_trylock** returns zero if a lock on the mutex object referenced by *mutex* is acquired. Otherwise, an error number is returned to indicate the error.

Error Codes

The **pthread_mutex_trylock** function will fail if:

EBUSY	The mutex could not be acquired because it was already locked.

The **pthread_mutex_lock**, **pthread_mutex_trylock** and **pthread_mutex_unlock** functions will fail if:

EINVAL	The value specified by mutex does not refer to an initialised mutex object.

The **pthread_mutex_lock** function will fail if:

EDEADLK	The current thread already owns the mutex and the mutex type is **pthread_mutex_errorcheck**.

The **pthread_mutex_unlock** function will fail if:

EPERM	The current thread does not own the mutex and the mutex type is not **pthread_mutex_normal**.

These functions will not return an error code of EINTR.

Implementation Specifics

These subroutines are part of the Base Operating System (BOS) Runtime.

Related Information

The **pthread_mutex_init** and **pthread_mutex_destroy** subroutines, **pthread.h** file.

Using Mutexes and Threads Library Quick Reference in *AIX General Programming Concepts : Writing and Debugging Programs*.

pthread_mutexattr_destroy or pthread_mutexattr_init Subroutine

Purpose

Initialises and destroys mutex attributes.

Library

Threads Library **(libpthreads.a)**

Syntax

```
#include <pthread.h>
int pthread_mutexattr_init (pthread_mutexattr_t *attr);
int pthread_mutexattr_destroy (pthread_mutexattr_t *attr);
```

Description

The function **pthread_mutexattr_init** initialises a mutex attributes object *attr* with the default value for all of the attributes defined by the implementation.

The effect of initialising an already initialised mutex attributes object is undefined.

After a mutex attributes object has been used to initialise one or more mutexes, any function affecting the attributes object (including destruction) does not affect any previously initialised mutexes.

The **pthread_mutexattr_destroy** function destroys a mutex attributes object; the object becomes, in effect, uninitialised. An implementation may cause **pthread_mutexattr_destroy** to set the object referenced by attr to an invalid value. A destroyed mutex attributes object can be re–initialised using **pthread_mutexattr_init**; the results of otherwise referencing the object after it has been destroyed are undefined.

Parameters

attr	Specifies the mutex attributes object to delete.

Return Values

Upon successful completion, **pthread_mutexattr_init** and **pthread_mutexattr_destroy** return zero. Otherwise, an error number is returned to indicate the error.

Error Codes

The **pthread_mutexattr_init** function will fail if:

ENOMEM	Insufficient memory exists to initialise the mutex attributes object.

The **pthread_mutexattr_destroy** function will fail if:

EINVAL	The value specified by *attr* is invalid.

These functions will not return EINTR.

Implementation Specifics

This subroutine is part of the Base Operating System (BOS) Runtime.

Related Information

The **pthread_create** subroutine, **pthread_mutex_init** subroutine, **pthread_cond_init** subroutine, **pthread.h** file.

Using Mutexes in *AIX General Programming Concepts : Writing and Debugging Programs*.

Threads Library Options and Threads Library Quick Reference in *AIX General Programming Concepts : Writing and Debugging Programs*.

pthread_mutexattr_getkind_np Subroutine

Purpose

Returns the value of the kind attribute of a mutex attributes object.

Library

Threads Library (**libpthreads.a**)

Syntax

```
#include <pthread.h>

int pthread_mutexattr_getkind_np (attr, kind)
pthread_mutexattr_t *attr;
int *kind;
```

Description

The **pthread_mutexattr_getkind_np** subroutine returns the value of the kind attribute of the mutex attributes object *attr*. This attribute specifies the kind of the mutex created with this attributes object. It may have one of the following values:

MUTEX_FAST_NP	Denotes a fast mutex. A fast mutex can be locked only once. If the same thread unlocks twice the same fast mutex, the thread will deadlock. Any thread can unlock a fast mutex. A fast mutex is not compatible with the priority inheritance protocol.
MUTEX_RECURSIVE_NP	Denotes a recursive mutex. A recursive mutex can be locked more than once by the same thread without causing that thread to deadlock. The thread must then unlock the mutex as many times as it locked it. Only the thread that locked a recursive mutex can unlock it. A recursive mutex must not be used with condition variables.
MUTEX_NONRECURSIVE_NP	Denotes the default non–recursive POSIX compliant mutex.

Notes:

1. The **pthread.h** header file must be the first included file of each source file using the threads library. Otherwise, the **–D_THREAD_SAFE** compilation flag should be used, or the cc_r compiler used. In this case, the flag is automatically set.
2. The **pthread_mutexattr_getkind_np** subroutine is not portable.

Parameters

attr	Specifies the mutex attributes object.
kind	Points to where the kind attribute value will be stored.

Return Values

Upon successful completion, the value of the kind attribute is returned via the *kind* parameter, and 0 is returned. Otherwise, an error code is returned.

Error Codes

The **pthread_mutexattr_getkind_np** subroutine is unsuccessful if the following is true:

EINVAL The *attr* parameter is not valid.

Implementation Specifics

This subroutine is part of the Base Operating System (BOS) Runtime.

This subroutine is not POSIX compliant and is provided only for compatibility with DCE threads. It should not be used when writing new applications.

Related Information

The **pthread_mutexattr_setkind_np** subroutine.

Using Mutexes in *AIX General Programming Concepts : Writing and Debugging Programs*.

pthread_mutexattr_getpshared or pthread_mutexattr_setpshared Subroutine

Purpose

Sets and gets process–shared attribute.

Library

Threads Library (**libpthreads.a**)

Syntax

```
#include <pthread.h>

int pthread_mutexattr_getpshared (attr, pshared)
const pthread_mutexattr_t *attr;
int *pshared;

int pthread_mutexattr_setpshared (attr, pshared)
pthread_mutexattr_t *attr;
int pshared;
```

Description

The **pthread_mutexattr_getpshared** function obtains the value of the process–shared attribute from the attributes object referenced by attr. The **pthread_mutexattr_setpshared** function is used to set the process–shared attribute in an initialised attributes object referenced by attr.

The process–shared attribute is set to PTHREAD_PROCESS_SHARED to permit a mutex to be operated upon by any thread that has access to the memory where the mutex is allocated, even if the mutex is allocated in memory that is shared by multiple processes. If the **process–shared** attribute is PTHREAD_PROCESS_PRIVATE, the mutex will only be operated upon by threads created within the same process as the thread that initialised the mutex; if threads of differing processes attempt to operate on such a mutex, the behaviour is undefined. The default value of the attribute is PTHREAD_PROCESS_PRIVATE.

Parameters

attr	Specifies the mutex attributes object.
pshared	Points to where the pshared attribute value will be stored.

Return Values

Upon successful completion, **pthread_mutexattr_setpshared** returns zero. Otherwise, an error number is returned to indicate the error.

Upon successful completion, **pthread_mutexattr_getpshared** returns zero and stores the value of the process–shared attribute of *attr* into the object referenced by the *pshared* parameter. Otherwise, an error number is returned to indicate the error.

Error Codes

The **pthread_mutexattr_getpshared** and **pthread_mutexattr_setpshared** functions will fail if:

EINVAL	The value specified by *attr* is invalid.

The **pthread_mutexattr_setpshared** function willfail if:

EINVAL	The new value specified for the attribute is outside the range of legal values for that attribute.

These functions will not return an error code of EINTR.

Implementation Specifics

This subroutine is part of the Base Operating System (BOS) Runtime.

Related Information

The **pthread_mutexattr_init** subroutine.

Advanced Attributes in *AIX General Programming Concepts : Writing and Debugging Programs*.

Threads Library Options and Threads Library Quick Reference in *AIX General Programming Concepts : Writing and Debugging Programs*.

pthread_mutexattr_gettype or pthread_mutexattr_settype Subroutines

Purpose

Gets or sets a mutex type.

Library

Threads Library (**libthreads.a**)

Syntax

```
#include <pthread.h>

int pthread_mutexattr_gettype (pthread_mutexattr_t *attr, int
*type);
int pthread_mutexattr_settype (pthread_mutexattr_t *attr, int
type);
```

Description

The pthread_mutexattr_gettype and pthread_mutexattr_settype functions respectively get and set the mutex type attribute. This attribute is set in the type parameter to these functions. The default value of the type attribute is PTHREAD_MUTEX_DEFAULT. The type of mutex is contained in the type attribute of the mutex attributes. Valid mutex types include:

PTHREAD_MUTEX_NORMAL
This type of mutex does not detect deadlock. A thread attempting to relock this mutex without first unlocking it will deadlock. Attempting to unlock a mutex locked by a different thread results in undefined behaviour. Attempting to unlock an unlocked mutex results in undefined behaviour.

PTHREAD_MUTEX_ERRORCHECK
This type of mutex provides error checking. A thread attempting to relock this mutex without first unlocking it will return with an error. A thread attempting to unlock a mutex which another thread has locked will return with an error. A thread attempting to unlock an unlocked mutex will return with an error.

PTHREAD_MUTEX_RECURSIVE
A thread attempting to relock this mutex without first unlocking it will succeed in locking the mutex. The relocking deadlock which can occur with mutexes of type PTHREAD_MUTEX_NORMAL cannot occur with this type of mutex. Multiple locks of this mutex require the same number of unlocks to release the mutex before another thread can acquire the mutex. A thread attempting to unlock a mutex which another thread has locked will return with an error. A thread attempting to unlock an unlocked mutex will return with | 20103 an error.

PTHREAD_MUTEX_DEFAULT
Attempting to recursively lock a mutex of this type results in undefined behaviour. Attempting to unlock a mutex of this type which was not locked by the calling thread results in undefined behaviour. Attempting to unlock a mutex of this type which is not locked results in undefined behaviour. An implementation is allowed to map this mutex to one of the other mutex types.

Return Values

If successful, the **pthread_mutexattr_settype** function returns zero. Otherwise, an error number is returned to indicate the error. Upon successful completion, the **pthread_mutexattr_gettype** function returns zero and stores the value of the type attribute of attr into the object referenced by the type parameter. Otherwise an error is returned to indicate the error.

Error Codes

The **pthread_mutexattr_gettype** and **pthread_mutexattr_settype** functions will fail if:

EINVAL	The value type is invalid.
EINVAL	The value specified by attr is invalid.

Implementation Specifics

It is advised that an application should not use a PTHREAD_MUTEX_RECURSIVE mutex with condition variables because the implicit unlock performed for a pthread_cond_wait or pthread_cond_timedwait may not actually release the mutex (if it had been locked multiple times). If this happens, no other thread can satisfy the condition of the predicate.

Related Information

The **pthread_cond_wait** and **pthread_cond_timedwait** subroutines.

The **pthread.h** file.

pthread_mutexattr_setkind_np Subroutine

Purpose

Sets the value of the kind attribute of a mutex attributes object.

Library

Threads Library (**libpthreads.a**)

Syntax

```
#include <pthread.h>

int pthread_mutexattr_setkind_np (attr, kind)
pthread_mutexattr_t *attr;
int kind;
```

Description

The **pthread_mutexattr_setkind_np** subroutine sets the value of the kind attribute of the mutex attributes object *attr*. This attribute specifies the kind of the mutex created with this attributes object.

Notes:

1. The **pthread.h** header file must be the first included file of each source file using the threads library. Otherwise, the **–D_THREAD_SAFE** compilation flag should be used, or the cc_r compiler used. In this case, the flag is automatically set.
2. The **pthread_mutexattr_setkind_np** subroutine is not portable.

Parameters

attr	Specifies the mutex attributes object.
kind	Specifies the kind to set. It must have one of the following values:

MUTEX_FAST_NP	Denotes a fast mutex. A fast mutex can be locked only once. If the same thread unlocks twice the same fast mutex, the thread will deadlock. Any thread can unlock a fast mutex. A fast mutex is not compatible with the priority inheritance protocol.
MUTEX_RECURSIVE_NP	Denotes a recursive mutex. A recursive mutex can be locked more than once by the same thread without causing that thread to deadlock. The thread must then unlock the mutex as many times as it locked it. Only the thread that locked a recursive mutex can unlock it. A recursive mutex must not be used with condition variables.
MUTEX_NONRECURSIVE_NP	Denotes the default non–recursive POSIX compliant mutex.

Return Values

Upon successful completion, 0 is returned. Otherwise, an error code is returned.

Error Codes

The **pthread_mutexattr_setkind_np** subroutine is unsuccessful if the following is true:

EINVAL	The *attr* parameter is not valid.
ENOTSUP	The value of the *kind* parameter is not supported.

Implementation Specifics

This subroutine is part of the Base Operating System (BOS) Runtime.

This subroutine is provided only for compatibility with the DCE threads. It should not be used when writing new applications.

Related Information

The **pthread_mutexattr_getkind_np** subroutine.

Using Mutexes in *AIX General Programming Concepts : Writing and Debugging Programs*.

pthread_once Subroutine

Purpose

Executes a routine exactly once in a process.

Library

Threads Library (**libpthreads.a**)

Syntax

```
#include <pthread.h>

int pthread_once (pthread_once_t *once_control, void
(*init_routine)(void));

pthread_once_t once_control = PTHREAD_ONCE_INIT;
```

Description

The **pthread_once** subroutine executes the routine *init_routine* exactly once in a process. The first call to this subroutine by any thread in the process executes the given routine, without parameters. Any subsequent call will have no effect.

The *init_routine* routine is typically an initialization routine. Multiple initializations can be handled by multiple instances of **pthread_once_t** structures. This subroutine is useful when a unique initialization has to be done by one thread among many. It reduces synchronization requirements.

Note: The **pthread.h** header file must be the first included file of each source file using the threads library. Otherwise, the **–D_THREAD_SAFE** compilation flag should be used, or the cc_r compiler used. In this case, the flag is automatically set.

Parameters

once_block	Points to a synchronization control structure. This structure has to be initialized by the static initializer macro **PTHREAD_ONCE_INIT**.
init_routine	Points to the routine to be executed.

Return Values

Upon successful completion, **pthread_once** returns zero. Otherwise, an error number is returned to indicate the error.

Error Codes

No errors are defined. The **pthread_once** function will not return an error code of EINTR.

Implementation Specifics

This subroutine is part of the Base Operating System (BOS) Runtime.

Related Information

The **pthread_create** subroutine, **pthread.h** file, **PTHREAD_ONCE_INIT** macro.

One Time Initializations in *AIX General Programming Concepts : Writing and Debugging Programs*.

Threads Library Quick Reference in *AIX General Programming Concepts : Writing and Debugging Programs*.

PTHREAD_ONCE_INIT Macro

Purpose

Initializes a once synchronization control structure.

Library

Threads Library (**libpthreads.a**)

Syntax

```
#include <pthread.h>

static pthread_once_t once_block = PTHREAD_ONCE_INIT;
```

Description

The **PTHREAD_ONCE_INIT** macro initializes the static once synchronization control structure *once_block*, used for one–time initializations with the **pthread_once** subroutine. The once synchronization control structure must be static to ensure the unicity of the initialization.

Note: The **pthread.h** file header file must be the first included file of each source file using the threads library. Otherwise, the **–D_THREAD_SAFE** compilation flag should be used, or the cc_r compiler used. In this case, the flag is automatically set.

Implementation Specifics

This macro is part of the Base Operating System (BOS) Runtime.

Related Information

The **pthread_once** subroutine.

One Time Initializations in *AIX General Programming Concepts : Writing and Debugging Programs*.

Threads Library Quick Reference in *AIX General Programming Concepts : Writing and Debugging Programs*.

pthread_rwlock_init, pthread_rwlock_destroy Subroutine

Purpose

Initialises or destroys a read–write lock object.

Library

Threads Library (**libthreads.a**)

Syntax

```
#include <pthread.h>

int pthread_rwlock_init (pthread_rwlock_t *rwlock, const
pthread_rwlock attr_t *attr);
int pthread_rwlock_destroy (pthread_rwlock_t *rwlock);
pthread_rwlock_t rwlock=PTHREAD_RWLOCK_INITIALIZER;
```

Description

The **pthread_rwlock_init** function initialises the read–write lock referenced by *rwlock* with the attributes referenced by *attr*. If *attr* is NULL, the default read–write lock attributes are used; the effect is the same as passing the address of a default read–write lock attributes object. Once initialised, the lock can be used any number of times without being re–initialised. Upon successful initialisation, the state of the read–write lock becomes initialised and unlocked. Results are undefined if **pthread_rwlock_init** is called specifying an already initialised read–write lock. Results are undefined if a read–write lock is used without first being initialised.

If the **pthread_rwlock_init** function fails, *rwlock* is not initialised and the contents of *rwlock* are undefined.

The **pthread_rwlock_destroy** function destroys the read–write lock object referenced by rwlock and releases any resources used by the lock. The effect of subsequent use of the lock is undefined until the lock is re–initialised by another call to **pthread_rwlock_init**. An implementation may cause **pthread_rwlock_destroy** to set the object referenced by *rwlock* to an invalid value. Results are undefined if **pthread_rwlock_destroy** is called when any thread holds rwlock. Attempting to destroy an uninitialised read–write lock results in undefined behaviour. A destroyed read–write lock object can be re–initialised using **pthread_rwlock_init**; the results of otherwise referencing the read–write lock object after it has been destroyed are undefined.

In cases where default read–write lock attributes are appropriate, the macro **PTHREAD_RWLOCK_INITIALIZER** can be used to initialise read–write locks that are statically allocated. The effect is equivalent to dynamic initialisation by a call to **pthread_rwlock_init** with the parameter *attr* specified as NULL, except that no error checks are performed.

Return Values

If successful, the pthread_rwlock_init and pthread_rwlock_destroy functions return zero. Otherwise, an error number is returned to indicate the error. The EBUSY and EINVAL error checks, if implemented, will act as if they were performed immediately at the beginning of processing for the function and caused an error return prior to modifying the state of the read–write lock specified by rwlock.

Error Codes

The **pthread_rwlock_init** function will fail if:

ENOMEM	Insufficient memory exists to initialise the read–write lock.
EINVAL	The value specified by *attr* is invalid.

The **pthread_rwlock_destroy** function will fail if:

EBUSY	The implementation has detected an attempt to destroy the object referenced by *rwlock* while it is locked.
EINVAL	The value specified by *attr* is invalid.

Implementation Specifics

Similar functions are being developed by IEEE PASC. In keeping with its objective of ensuring that CAE Specifications are fully aligned with formal standards, The Open Group intends to add any new interfaces adopted by an official IEEE standard in this area.

Related Information

The **pthread.h** file.

The **pthread_rwlock_rdlock**, **pthread_rwlock_wrlock**, **pthread_rwlockattr_init** and **pthread_rwlock_unlock** subroutines.

pthread_rwlock_rdlock or pthread_rwlock_tryrdlock Subroutines

Purpose

Locks a read–write lock object for reading.

Library

Threads Library (**libpthreads.a**)

Syntax

```
#include <pthread.h>

int pthread_rwlock_rdlock ( pthread_rwlock_t *rwlock);
int pthread_rwlock_tryrdlock (pthread_rwlock_t *rwlock);
```

Description

The **pthread_rwlock_rdlock** function applies a read lock to the read–write lock referenced by *rwlock*. The calling thread acquires the read lock if a writer does not hold the lock and there are no writers blocked on the lock. It is unspecified whether the calling thread acquires the lock when a writer does not hold the lock and there are writers waiting for the lock. If a writer holds the lock, the calling thread will not acquire the read lock. If the read lock is not acquired, the calling thread blocks (that is, it does not return from the **pthread_rwlock_rdlock call**) until it can acquire the lock. Results are undefined if the calling thread holds a write lock on *rwlock* at the time the call is made.

Implementations are allowed to favour writers over readers to avoid writer starvation.

A thread may hold multiple concurrent read locks on *rwlock* (that is, successfully call the **pthread_rwlock_rdlock** function *n* times). If so, the thread must perform matching unlocks (that is, it must call the **pthread_rwlock_unlock** function *n* times).

The function **pthread_rwlock_tryrdlock** applies a read lock as in the **pthread_rwlock_rdlock** function with the exception that the function fails if any thread holds a write lock on *rwlock* or there are writers blocked on *rwlock*.

Results are undefined if any of these functions are called with an uninitialised read–write lock.

If a signal is delivered to a thread waiting for a read–write lock for reading, upon return from the signal handler the thread resumes waiting for the read–write lock for reading as if it was not interrupted.

Return Values

If successful, the **pthread_rwlock_rdlock** function returns zero. Otherwise, an error number is returned to indicate the error.

The function **pthread_rwlock_tryrdlock** returns zero if the lock for reading on the read–write lock object referenced by *rwlock* is acquired. Otherwise an error number is returned to indicate the error.

Error Codes

The **pthread_rwlock_tryrdlock** function will fail if:

EBUSY	The read–write lock could not be acquired for reading because a writer holds the lock or was blocked on it.

The **pthread_rwlock_rdlock** and **pthread_rwlock_tryrdlock** functions will fail if:

EINVAL The value specified by *rwlock* does not refer to an initialised read–write lock object.

EDEADLK The current thread already owns the read–write lock for writing.

EAGAIN The read lock could not be acquired because the maximum number of read locks for *rwlock* has been exceeded.

Implementation Specifics

Similar functions are being developed by IEEE PASC. In keeping with its objective of ensuring that CAE Specifications are fully aligned with formal standards, The Open Group intends to add any new interfaces adopted by an official IEEE standard in this area.

Realtime applications may encounter priority inversion when using read–write locks. The problem occurs when a high priority thread 'locks' a read–write lock that is about to be 'unlocked' by a low priority thread, but the low priority thread is preempted by a medium priority thread. This scenario leads to priority inversion; a high priority thread is blocked by lower priority threads for an unlimited period of time. During system design, realtime programmers must take into account the possibility of this kind of priority inversion. They can deal with it in a number of ways, such as by having critical sections that are guarded by read–write locks execute at a high priority, so that a thread cannot be preempted while executing in its critical section.

Related Information

The **pthread.h** file.

The **pthread_rwlock_init**, **pthread_rwlock_wrlock**, **pthread_rwlockattr_init**, and **pthread_rwlock_unlock** subroutines.

pthread_rwlock_unlock Subroutine

Purpose

Unlocks a read–write lock object.

Library

Threads Library (**libthreads.a**)

Syntax

```
#include <pthread.h>

int pthread_rwlock_unlock (pthread_rwlock_t *rwlock);
```

Description

The **pthread_rwlock_unlock** function is called to release a lock held on the read–write lock object referenced by *rwlock*. Results are undefined if the read–write lock *rwlock* is not held by the calling thread.

If this function is called to release a read lock from the read–write lock object and there are other read locks currently held on this read–write lock object, the read–write lock object remains in the read locked state. If this function releases the calling thread's last read lock on this read–write lock object, then the calling thread is no longer one of the owners of the object. If this function releases the last read lock for this read–write lock object, the read–write lock object will be put in the unlocked state with no owners.

If this function is called to release a write lock for this read–write lock object, the read–write lock object will be put in the unlocked state with no owners.

If the call to the **pthread_rwlock_unlock** function results in the read–write lock object becoming unlocked and there are multiple threads waiting to acquire the read–write lock object for writing, the scheduling policy is used to determine which thread acquires the read–write lock object for writing. If there are multiple threads waiting to acquire the read–write lock object for reading, the scheduling policy is used to determine the order in which the waiting threads acquire the read–write lock object for reading. If there are multiple threads blocked on rwlock for both read locks and write locks, it is unspecified whether the readers acquire the lock first or whether a writer acquires the lock first.

Results are undefined if any of these functions are called with an uninitialised read–write lock.

Return Values

If successful, the **pthread_rwlock_unlock** function returns zero. Otherwise, an error number is returned to indicate the error.

Error Codes

The **pthread_rwlock_unlock** function will fail if:

EINVAL	The value specified by rwlock does not refer to an initialised read–write lock object.
EPERM	The current thread does not own the read–write lock.

Implementation Specifics

Similar functions are being developed by IEEE PASC. In keeping with its objective of ensuring that CAE Specifications are fully aligned with formal standards, The Open Group intends to add any new interfaces adopted by an official IEEE standard in this area.

Related Information

The **pthread.h** file.

The **pthread_rwlock_init**, **pthread_rwlock_wrlock**, **pthread_rwlockattr_init**, **pthread_rwlock_rdlock** subroutines.

pthread_rwlock_wrlock or pthread_rwlock_trywrlock Subroutines

Purpose

Locks a read–write lock object for writing.

Library

Threads Library (**libpthreads.a**)

Syntax

```
#include <pthread.h>

int pthread_rwlock_wrlock (pthread_rwlock_t *rwlock );
int pthread_rwlock_trywrlock (pthread_rwlock_t *rwlock );
```

Description

The **pthread_rwlock_wrlock** function applies a write lock to the read–write lock referenced by *rwlock*. The calling thread acquires the write lock if no other thread (reader or writer) holds the read–write lock *rwlock*. Otherwise, the thread blocks (that is, does not return from the **pthread_rwlock_wrlock** call) until it can acquire the lock. Results are undefined if the calling thread holds the read–write lock (whether a read or write lock) at the time the call is made.

Implementations are allowed to favour writers over readers to avoid writer starvation.

The function **pthread_rwlock_trywrlock** applies a write lock like the **pthread_rwlock_wrlock** function, with the exception that the function fails if any thread currently holds rwlock (for reading or writing).

Results are undefined if any of these functions are called with an uninitialised read–write lock.

If a signal is delivered to a thread waiting for a read–write lock for writing, upon return from the signal handler the thread resumes waiting for the read–write lock for writing as if it was not interrupted.

Return Values

If successful, the **pthread_rwlock_wrlock** function returns zero. Otherwise, an error number is returned to indicate the error.

The function **pthread_rwlock_trywrlock** returns zero if the lock for writing on the read–write lock object referenced by rwlock is acquired. Otherwise an error number is returned to indicate the error.

Error Codes

The **pthread_rwlock_trywrlock** function will fail if:

EBUSY	The read–write lock could not be acquired for writing because it was already locked for reading or writing.

The **pthread_rwlock_wrlock** and **pthread_rwlock_trywrlock** functions will fail if:

EINVAL	The value specified by rwlock does not refer to an initialised read–write lock object.
EDEADLK	The current thread already owns the read–write lock for writing or reading.

Implementation Specifics

Similar functions are being developed by IEEE PASC. In keeping with its objective of ensuring that CAE Specifications are fully aligned with formal standards, The Open Group intends to add any new interfaces adopted by an official IEEE standard in this area.

Realtime applications may encounter priority inversion when using read–write locks. The problem occurs when a high priority thread 'locks' a read–write lock that is about to be 'unlocked' by a low priority thread, but the low priority thread is preempted by a medium priority thread. This scenario leads to priority inversion; a high priority thread is blocked by lower priority threads for an unlimited period of time. During system design, realtime programmers must take into account the possibility of this kind of priority inversion. They can deal with it in a number of ways, such as by having critical sections that are guarded by read–write locks execute at a high priority, so that a thread cannot be preempted while executing in its critical section.

Related Information

The **pthread.h** file.

The **pthread_rwlock_init**, **pthread_rwlock_unlock**, **pthread_rwlockattr_init**, **pthread_rwlock_rdlock** subroutines.

pthread_rwlockattr_getpshared or pthread_rwlockattr_setpshared Subroutines

Purpose

Gets and sets process–shared attribute of read–write lock attributes object.

Library

Threads Library (**libpthreads.a**)

Syntax

```
#include <pthread.h>

int pthread_rwlockattr_getpshared (const pthread_rwlockattr_t
*attrint *pshared );
int pthread_rwlockattr_setpshared (pthread_rwlockattr_t *attr,
int pshared);
```

Description

The process–shared attribute is set to PTHREAD_PROCESS_SHARED to permit a read–write lock to be operated upon by any thread that has access to the memory where the read–write lock is allocated, even if the read–write lock is allocated in memory that is shared by multiple processes. If the process–shared attribute is PTHREAD_PROCESS_PRIVATE, the read–write lock will only be operated upon by threads created within the same process as the thread that initialised the read–write lock; if threads of differing processes attempt to operate on such a read–write lock, the behaviour is undefined. The default value of the process–shared attribute is PTHREAD_PROCESS_PRIVATE.

The **pthread_rwlockattr_getpshared** function obtains the value of the process–shared attribute from the initialised attributes object referenced by *attr*. The **pthread_rwlockattr_setpshared** function is used to set the process–shared attribute in an initialised attributes object referenced by *attr*.

Return Values

If successful, the **pthread_rwlockattr_setpshared** function returns zero. Otherwise, an error number is returned to indicate the error.

Upon successful completion, the **pthread_rwlockattr_getpshared** returns zero and stores the value of the process–shared attribute of attr into the object referenced by the pshared parameter. Otherwise an error number is returned to indicate the error.

Error Codes

The **pthread_rwlockattr_getpshared** and **pthread_rwlockattr_setpshared** functions will fail if:

EINVAL	The value specified by *attr* is invalid.

The **pthread_rwlockattr_setpshared** function will fail if:

EINVAL	The new value specified for the attribute is outside the range of legal values for that attribute.

Implementation Specifics

Similar functions are being developed by IEEE PASC. In keeping with its objective of ensuring that CAE Specifications are fully aligned with formal standards, The Open Group intends to add any new interfaces adopted by an official IEEE standard in this area.

Related Information

The **pthread.h** file.

The **pthread_rwlock_init**, **pthread_rwlock_unlock**, **pthread_rwlock_wrlock**, **pthread_rwlock_rdlock**, **pthread_rwlockattr_init** subroutines.

pthread_rwlockattr_init or pthread_rwlockattr_destroy Subroutines

Purpose

Initialises and destroys read–write lock attributes object.

Library

Threads Library (**libpthreads.a**)

Syntax

```
#include <pthread.h>



int pthread_rwlockattr_init (pthread_rwlockattr_t *attr);
int pthread_rwlockattr_destroy (pthread_rwlockattr_t *attr);
```

Description

The function **pthread_rwlockattr_init** initialises a read–write lock attributes object *attr* with the default value for all of the attributes defined by the implementation. Results are undefined if **pthread_rwlockattr_init** is called specifying an already initialised read–write lock attributes object.

After a read–write lock attributes object has been used to initialise one or more read–write locks, any function affecting the attributes object (including destruction) does not affect any previously initialised read–write locks.

The **pthread_rwlockattr_destroy** function destroys a read–write lock attributes object. The effect of subsequent use of the object is undefined until the object is re–initialised by another call to **pthread_rwlockattr_init**. An implementation may cause **pthread_rwlockattr_destroy** to set the object referenced by attr to an invalid value.

Return Value

If successful, the **pthread_rwlockattr_init** and **pthread_rwlockattr_destroy** functions return zero. Otherwise, an error number is returned to indicate the error.

Error Codes

The **pthread_rwlockattr_init** function will fail if:

ENOMEM	Insufficient memory exists to initialise the read–write lock attributes object.

The **pthread_rwlockattr_destroy** function will fail if:

EINVAL	The value specified by attr is invalid.

Implementation Specifics

Similar functions are being developed by IEEE PASC. In keeping with its objective of ensuring that CAE Specifications are fully aligned with formal standards, The Open Group intends to add any new interfaces adopted by an official IEEE standard in this area.

Related Information

The **pthread.h** file.

The **pthread_rwlock_init**, **pthread_rwlock_unlock**, **pthread_rwlock_wrlock**, **pthread_rwlock_rdlock**, and **pthread_rwlockattr_getpshared** subroutines.

pthread_self Subroutine

Purpose

Returns the calling thread's ID.

Library

Threads Library (**libpthreads.a**)

Syntax

```
#include <pthread.h>

pthread_t pthread_self (void);
```

Description

The **pthread_self** subroutine returns the calling thread's ID.

Note: The **pthread.h** header file must be the first included file of each source file using the threads library. Otherwise, the **–D_THREAD_SAFE** compilation flag should be used, or the cc_r compiler used. In this case, the flag is automatically set.

Return Values

The calling thread's ID is returned.

Errors No errors are defined.

The **pthread_self** function will not return an error code of EINTR.

Implementation Specifics

This subroutine is part of the Base Operating System (BOS) Runtime.

Related Information

The **pthread_create** subroutine, **pthread_equal** subroutine, **pthread.h** file.

Creating Threads in *AIX General Programming Concepts : Writing and Debugging Programs*.

Threads Library Quick Reference in *AIX General Programming Concepts : Writing and Debugging Programs*.

pthread_setcancelstate, pthread_setcanceltype or pthread_testcancel Subroutines

Purpose

Sets the calling thread's cancelability state.

Library

Threads Library (**libpthreads.a**)

Syntax

```
#include <pthread.h>



int pthread_setcancelstate (int state, int *oldstate);
int pthread_setcanceltype (int type, int *oldstype);
int pthread_testcancel (void);
```

Description

The **pthread_setcancelstate** function atomically both sets the calling thread's cancelability state to the indicated state and returns the previous cancelability state at the location referenced by *oldstate*. Legal values for state are PTHREAD_CANCEL_ENABLE and PTHREAD_CANCEL_DISABLE.

The **pthread_setcanceltype** function atomically both sets the calling thread's cancelability type to the indicated type and returns the previous cancelability type at the location referenced by *oldtype*. Legal values for type are PTHREAD_CANCEL_DEFERRED and PTHREAD_CANCEL_ASYNCHRONOUS.

The cancelability state and type of any newly created threads, including the thread in which **main** was first invoked, are PTHREAD_CANCEL_ENABLE and PTHREAD_CANCEL_DEFERRED respectively.

The **pthread_testcancel** function creates a cancellation point in the calling thread. The **pthread_testcancel** function has no effect if cancelability is disabled.

Parameters

state — Specifies the new cancelability state to set. It must have one of the following values:

PTHREAD_CANCEL_DISABLE
: Disables cancelability; the thread is not cancelable. Cancellation requests are held pending.

PTHREAD_CANCEL_ENABLE
: Enables cancelability; the thread is cancelable, according to its cancelability type. This is the default value.

oldstate — Points to where the previous cancelability state value will be stored.

Return Values

If successful, the **pthread_setcancelstate** and **pthread_setcanceltype** functions return zero. Otherwise, an error number is returned to indicate the error.

Error Codes

The **pthread_setcancelstate** function will fail if:

EINVAL	The specified state is not PTHREAD_CANCEL_ENABLE or PTHREAD_CANCEL_DISABLE.

The **pthread_setcanceltype** function will fail if:

EINVAL	The specified type is not PTHREAD_CANCEL_DEFERRED or PTHREAD_CANCEL_ASYNCHRONOUS.

These functions will not return an error code of EINTR.

Implementation Specifics

This subroutine is part of the Base Operating System (BOS) Runtime.

Related Information

The **pthread_cancel** subroutine, the **pthread.h** file.

Terminating Threads in *AIX General Programming Concepts : Writing and Debugging Programs*.

Threads Library Quick Reference in *AIX General Programming Concepts : Writing and Debugging Programs*.

pthread_setschedparam Subroutine

Purpose

Returns the current schedpolicy and schedparam attributes of a thread.

Library

Threads Library (**libpthreads.a**)

Syntax

```
#include <pthread.h>
#include <sys/sched.h>
int pthread_setschedparam (thread, schedpolicy, schedparam)
pthread_t thread;
int schedpolicy;
const struct sched_param *schedparam;
```

Description

The **pthread_setschedparam** subroutine dynamically sets the schedpolicy and schedparam attributes of the thread *thread*. The schedpolicy attibute specifies the scheduling policy of the thread. The schedparam attribute specifies the scheduling parameters of a thread created with this attributes object. The `sched_priority` field of the **sched_param** structure contains the priority of the thread. It is an integer value.

If the target thread has system contention scope, the process must have root authority to set the scheduling policy to either **SCHED_FIFO** or **SCHED_RR**.

> **Note:** The **pthread.h** header file must be the first included file of each source file using the threads library. Otherwise, the **–D_THREAD_SAFE** compilation flag should be used, or the cc_r compiler used. In this case, the flag is automatically set.

Parameters

thread	Specifies the target thread.
schedpolicy	Points to the schedpolicy attribute to set. It must have one of the following values:

SCHED_FIFO	Denotes first–in first–out scheduling.
SCHED_RR	Denotes round–robin scheduling.
SCHED_OTHER	Denotes the default AIX scheduling policy. It is the default value.

Note: It is not permitted to change the priority of a thread when setting its scheduling policy to SCHED_OTHER. In this case, the priority is managed directly by the kernel, and the only legal value that can be passed to **pthread_setschedparam** is DEFAULT_PRIO, which is defined in **pthread.h** as 1.

schedparam	Points to where the scheduling parameters to set are stored. The `sched_priority` field must be in the range from 1 to 127, where 1 is the least favored priority, and 127 the most favored.

Return Values

Upon successful completion, 0 is returned. Otherwise, an error code is returned.

Error Codes

The **pthread_setschedparam** subroutine is unsuccessful if the following is true:

EINVAL	The *thread* or *schedparam* parameters are not valid.
ENOSYS	The priority scheduling POSIX option is not implemented.
ENOTSUP	The value of the schedpolicy or schedparam attributes are not supported.
EPERM	The target thread has insufficient permission to perform the operation or is already engaged in a mutex protocol.
ESRCH	The thread *thread* does not exist.

Implementation Specifics

This subroutine is part of the Base Operating System (BOS) Runtime. The implementation of this subroutine is dependent on the priority scheduling POSIX option. The priority scheduling POSIX option is implemented in AIX.

Related Information

The **pthread_getschedparam** subroutine, **pthread_attr_setschedpolicy** subroutine, **pthread_attr_setschedparam** subroutine.

Threads Scheduling in *AIX General Programming Concepts : Writing and Debugging Programs*.

Threads Library Options in *AIX General Programming Concepts : Writing and Debugging Programs*.

Threads Library Quick Reference in *AIX General Programming Concepts : Writing and Debugging Programs*.

pthread_sigmask Subroutine

Purpose

Examines and changes blocked signals.

Library

Threads Library (**libpthreads.a**)

Syntax

```
#include <signal.h>

int pthread_sigmask (int how, const sigset_t *set, sigset_t *oset
fP);
```

Description

Refer to **sigprocmask**.

pthread_signal_to_cancel_np Subroutine

Purpose

Cancels the specified thread.

Library

Threads Library (**libpthreads.a**)

Syntax

```
#include <pthread.h>

int pthread_signal_to_cancel_np (sigset, thread)
sigset_t *sigset;
pthread_t *target;
```

Description

The **pthread_signal_to_cancel_np** subroutine cancels the target thread *thread* by creating a handler thread. The handler thread calls the **sigwait** subroutine with the *sigset* parameter, and cancels the target thread when the **sigwait** subroutine returns. Successive call to this subroutine override the previous one.

Notes:

1. The **pthread.h** header file must be the first included file of each source file using the threads library. Otherwise, the **–D_THREAD_SAFE** compilation flag should be used, or the cc_r compiler used. In this case, the flag is automatically set.
2. The **pthread_signal_to_cancel_np** subroutine is not portable.

Parameters

sigset	Specifies the set of signals to wait on.
thread	Specifies the thread to cancel.

Return Values

Upon successful completion, 0 is returned. Otherwise, an error code is returned.

Error Codes

The **pthread_signal_to_cancel_np** subroutine is unsuccessful if the following is true:

EAGAIN	The handler thread cannot be created.
EINVAL	The *sigset* or *thread* parameters are not valid.

Implementation Specifics

This subroutine is part of the Base Operating System (BOS) Runtime.

This subroutine is not POSIX compliant and is provided only for compatibility with DCE threads. It should not be used when writing new applications.

Related Information

The **pthread_cancel** subroutine, **sigwait** subroutine.

pthread_suspend_np and pthread_continue_np Subroutine

Purpose

Suspends execution of the pthread specified by *thread*.

Library

Threads Library (**libpthreads.a**)

Syntax

```
#include <pthread.h>

int pthread_suspend_np(pthread_t thread);

int pthread_continue_np(pthread_t thread);
```

Description

The **pthread_suspend_np** routine immediately suspends the execution of the pthread specified by *thread*. On successful return from **pthread_suspend_np**, the suspended pthread is no longer executing. If **pthread_suspend_np** is called for a pthread that is already suspended, the pthread is unchanged and **pthread_suspend_np** returns successful.

The **pthread_continue_np** routine resumes the execution of a suspended pthread. If **pthread_continue_np** is called for a pthread that is not suspended, the pthread is unchanged and **pthread_continue_np** returns successful.

A suspended pthread will not be awakened by a signal. The signal stays pending until the execution of pthread is resumed by **pthread_continue_np**.

Parameters

thread	Specifies the target thread.

Return Values

Zero is returned when successful. A non–zero value indicates an error.

Error Codes

If any of the following conditions occur, **pthread_suspend_np** and **pthread_continue_np** fail and return the corresponding value:

ESRCH	The *thread* attribute cannot be found in the current process.

Implementation Specifics

This subroutine is part of the Base Operating System (BOS) Runtime.

pthread_unlock_global_np Subroutine

Purpose

Unlocks the global mutex.

Library

Threads Library (**libpthreads.a**)

Syntax

```
#include <pthread.h>

void pthread_unlock_global_np ()
```

Description

The **pthread_unlock_global_np** subroutine unlocks the global mutex when each call to the **pthread_lock_global_np** subroutine is matched by a call to this routine. For example, if a thread called the **pthread_lock_global_np** three times, the global mutex is unlocked after the third call to the **pthread_unlock_global_np** subroutine.

If no threads are waiting for the global mutex, it becomes unlocked with no current owner. If one or more threads are waiting to lock the global mutex, exactly one thread returns from its call to the **pthread_lock_global_np** subroutine.

Notes:

1. The **pthread.h** header file must be the first included file of each source file using the threads library. Otherwise, the **–D_THREAD_SAFE** compilation flag should be used, or the cc_r compiler used. In this case, the flag is automatically set.
2. The **pthread_unlock_global_np** subroutine is not portable.

Implementation Specifics

This subroutine is part of the Base Operating System (BOS) Runtime.

This subroutine is not POSIX compliant and is provided only for compatibility with DCE threads. It should not be used when writing new applications.

Related Information

The **pthread_lock_global_np** subroutine.

Using Mutexes in *AIX General Programming Concepts : Writing and Debugging Programs*.

pthread_yield Subroutine

Purpose

Forces the calling thread to relinquish use of its processor.

Library

Threads Library (**libpthreads.a**)

Syntax

```
#include <pthread.h>

void pthread_yield ()
```

Description

The **pthread_yield** subroutine forces the calling thread to relinquish use of its processor, and to wait in the run queue before it is scheduled again. If the run queue is empty when the **pthread_yield** subroutine is called, the calling thread is immediately rescheduled.

If the thread has global contention scope (**PTHREAD_SCOPE_SYSTEM**), calling this subroutine acts like calling the **yield** subroutine. Otherwise, another local contention scope thread is scheduled.

The **pthread.h** header file must be the first included file of each source file using the threads library. Otherwise, the **–D_THREAD_SAFE** compilation flag should be used, or the cc_r compiler used. In this case, the flag is automatically set.

Implementation Specifics

This subroutine is part of the Base Operating System (BOS) Runtime.

Related Information

The **yield** subroutine and the **sched_yield** subroutine.

Threads Scheduling in *AIX General Programming Concepts : Writing and Debugging Programs*.

Threads Library Options and Threads Library Quick Reference in *AIX General Programming Concepts : Writing and Debugging Programs*.

ptrace, ptracex Subroutine

Purpose

Traces the execution of another process.

Library

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

Syntax

```
#include <sys/reg.h>
#include <sys/ptrace.h>
#include <sys/ldr.h>

int ptrace (Request, Identifier,Address, Data, Buffer)
int Request;
int Identifier;
int *Address;
int Data;
int *Buffer;
```

```
int ptracex (request, identifier, long long addr, data, buff)
int request;
int identifier, long long addr;
int data;
int *buff;
```

Description

The **ptrace** subroutine allows a 32–bit process to trace the execution of another process. The **ptrace** subroutine is used to implement breakpoint debugging.

A debugged process executes normally until it encounters a signal. Then it enters a stopped state and its debugging process is notified with the **wait** subroutine. While the process is in the stopped state, the debugger examines and modifies its memory image by using the **ptrace** subroutine. For multi–threaded processes, the **getthrds** subroutine is used to identify each kernel thread in the debugged process. Also, the debugging process can cause the debugged process to terminate or continue, with the possibility of ignoring the signal that caused it to stop.

As a security measure, the **ptrace** subroutine inhibits the set–user–ID facility on subsequent **exec** subroutines.

(This paragraph only applies to AIX 4.3.1 and previous releases.) When a process is executing under **ptrace** control, portions of the process's address space are recopied after **load**, **unload**, and **loadbind** calls. For a 32–bit process, the main program text (loaded in segment 1) and shared library modules (loaded in segment 13) is recopied. Any breakpoints or other modifications to these segments must be reinserted after **load**, **unload**, or **loadbind**. Changes to privately loaded modules persist. For a 64–bit process, shared library modules are recopied after **load** and **unload** are called. (For AIX 4.3.0 and 4.3.1, these segments have a virtual address of 0x09000000xxxxxxxx, where x denotes any value.) The segments for the main programs and the segments containing privately loaded modules are not recopied. When a 64–bit process calls **loadbind**, no segments are recopied and the debugger is not notified.

(This paragraph only applies to AIX 4.3.2 and later releases.) When a process executing under **ptrace** control calls **load** or **unload**, the debugger is notified and the **W_SLWTED** flag is set in the status returned by **wait**. (A 32–bit process calling **loadbind** is stopped as well.) If the process being debugged has added modules in the shared library to its address space, the modules are added to the process's private copy of the shared library segments. If shared library modules are removed from a process's address space, the modules are deleted from the process's private copy of the library text segment by freeing the pages that contain the module. No other changes to the segment are made, and existing breakpoints do not have to be reinserted.

When a process being traced forks, the child process is initialized with the unmodified main program and shared library segment, effectively removing breakpoints in these segments in the child process. If multiprocess debugging is enabled, new copies of the main program and shared library segments are made. Modifications to privately loaded modules, however, are not affected by a fork. These breakpoints will remain in the child process, and if these breakpoints are executed, a SIGTRAP signal will be generated and delivered to the process.

If a traced process initiates an **exec** subroutine, the process stops before executing the first instruction of the new image and returns the **SIGTRAP** signal.

Note: **ptrace** and **ptracex** are not supported in 64–bit mode.

For the 64–bit Process

Use **ptracex** where the debuggee is a 64–bit process and the operation requested uses the third (address) parameter to reference the debuggee's address space or is sensitive to register size.

If returning or passing an **int** doesn't work for a 64–bit debuggee (for example, PT_READ_GPR), the buffer parameter takes the address for the result. Thus, with the **ptracex** subroutine, PT_READ_GPR and PT_WRITE_GPR take a pointer to an 8 byte area representing the register value.

In general, **ptracex** supports all the calls that **ptrace** does when they are modified for any that are extended for 64–bit addresses (for example, GPRs, LR, CTR, IAR, and MSR). Anything whose size increases for 64–bit processes must be allowed for in the obvious way (for example, PT_REGSET must be an array of long longs for a 64–bit debuggee).

Parameters

Request

Determines the action to be taken by the **ptrace** subroutine and has one of the following values:

PT_ATTACH This request allows a debugging process to attach a current process and place it into trace mode for debugging. This request cannot be used if the target process is already being traced. The *Identifier* parameter is interpreted as the process ID of the traced process. The *Address*, *Data*, and *Buffer* parameters are ignored.

If this request is unsuccessful, –1 is returned and the **errno** global variable is set to one the following codes:

ESRCH *Process* ID is not valid; the traced process is a kernel process; the process is currently being traced; or, the debugger or traced process already exists.

EPERM Real or effective user ID of the debugger does not match that of the traced process, or the debugger does not have root authority.

EINVAL The debugger and the traced process are the same.

PT_CONTINUE This request allows the process to resume execution. If the *Data* parameter is 0, all pending signals, including the one that caused the process to stop, are concealed before the process resumes execution. If the data parameter is a valid signal number, the process resumes execution as if it had received that signal. If the *Address* parameter equals 1, the execution continues from where it stopped. If the *Address* parameter is not 1, it is assumed to be the address at which the process should resume execution. Upon successful completion, the value of the *Data* parameter is returned to the debugging process. The *Identifier* parameter is interpreted as the process ID of the traced process. The *Buffer* parameter is ignored.

If this request is unsuccessful, –1 is returned and the **errno** global variable is set to the following code:

EIO The signal to be sent to the traced process is not a valid signal number.

Note: For the **PT_CONTINUE** request, use **ptracex** with a 64–bit debuggee because the resume address needs 64 bits.

PTT_CONTINUE

This request asks the scheduler to resume execution of the kernel thread specified by *Identifier*. This kernel thread must be the one that caused the exception. The *Data* parameter specifies how to handle signals:

- If the *Data* parameter is zero, the kernel thread which caused the exception will be resumed as if the signal never occurred.
- If the *Data* parameter is a valid signal number, the kernel thread which caused the exception will be resumed as if it had received that signal.

The *Address* parameter specifies where to resume execution:

- If the *Address* parameter is one, execution resumes from the address where it stopped.
- If the *Address* parameter contains an address value other than one, execution resumes from that address.

The *Buffer* parameter should point to a `PTTHREADS` structure, which contains a list of kernel thread identifiers to be started. This list should be NULL terminated if it is smaller than the maximum allowed.

On successful completion, the value of the *Data* parameter is returned to the debugging process. On unsuccessful completion, the value –1 is returned, and the **errno** global variable is set as follows:

EINVAL The *Identifier* parameter names the wrong kernel thread.

EIO The signal to be sent to the traced kernel thread is not a valid signal number.

ESRCH The *Buffer* parameter names an invalid kernel thread. Each kernel thread in the list must be stopped and belong to the same process as the kernel thread named by the *Identifier* parameter.

Note: For the **PTT_CONTINUE** request, use **ptracex** with a 64–bit debuggee because the resume address needs 64 bits.

PT_DETACH This request allows a debugged process, specified by the *Identifier* parameter, to exit trace mode. The process then continues running, as if it had received the signal whose number is contained in the data parameter. The process is no longer traced and does not process any further **ptrace** calls. The *Address* and *Buffer* parameters are ignored.

If this request is unsuccessful, –1 is returned and the **errno** global variable is set to the following code:

EIO Signal to be sent to the traced process is not a valid signal number.

PT_KILL This request allows the process to terminate the same way it would with an **exit** subroutine.

PT_LDINFO This request retrieves a description of the object modules that were loaded by the debugged process. The *Identifier* parameter is interpreted as the process ID of the traced process. The *Buffer* parameter is ignored. The *Address* parameter specifies the location where the loader information is copied. The *Data* parameter specifies the size of this area. The loader information is retrieved as a linked list of **ld_info** structures. The **ld_info** structures are defined in the **/usr/include/sys/ldr.h** file. The linked list is implemented so that the `ldinfo_nxt` field of each element gives the offset of the next element from this element. The `ldinfo_nxt` field of the last element has the value 0.

If this request is unsuccessful, –1 is returned and the **errno** global variable is set to the following code:

ENOMEM Either the area is not large enough to accommodate the loader information, or there is not enough memory to allocate an equivalent buffer in the kernel.

Note: For the **PT_LDINFO** request, use **ptracex** with a 64–bit debuggee because the source address needs 64 bits.

PT_MULTI This request turns multiprocess debugging mode on and off, to allow debugging to continue across **fork** and **exec** subroutines. A 0 value for the data parameter turns multiprocess debugging mode off, while all other values turn it on. When multiprocess debugging mode is in effect, any **fork** subroutine allows both the traced process and its newly created process to trap on the next instruction. If a traced process initiated an **exec** subroutine, the process stops before executing the first instruction of the new image and returns the **SIGTRAP** signal. The *Identifier* parameter is interpreted as the process ID of the traced process. The *Address* and *Buffer* parameters are ignored.

Also, when multiprocess debugging mode is enabled, the following values are returned from the **wait** subroutine:

W_SEWTED Process stopped during execution of the **exec** subroutine.

W_SFWTED Process stopped during execution of the **fork** subroutine.

PT_READ_BLOCK
This request reads a block of data from the debugged process address space. The *Address* parameter points to the block of data in the process address space, and the *Data* parameter gives its length in bytes. The value of the *Data* parameter must not be greater than 1024. The *Identifier* parameter is interpreted as the process ID of the traced process. The *Buffer* parameter points to the location in the debugging process address space where the data is copied. Upon successful completion, the **ptrace** subroutine returns the value of the data parameter.

If this request is unsuccessful, –1 is returned and the **errno** global variable is set to one of the following codes:

EIO The *Data* parameter is less than 1 or greater than 1024.

EIO The *Address* parameter is not a valid pointer into the debugged process address space.

EFAULT The *Buffer* parameter does not point to a writable location in the debugging process address space.

Note: For the **PT_READ_BLOCK** request, use **ptracex** with a 64–bit debuggee because the source address needs 64 bits.

Note: For the **PT_READ_BLOCK** request, use **ptracex** with a 64–bit debuggee because the source address needs 64 bits.

PT_READ_FPR
This request stores the value of a floating–point register into the location pointed to by the *Address* parameter. The *Data* parameter specifies the floating–point register, defined in the **sys/reg.h** file for the machine type on which the process is executed. The *Identifier* parameter is interpreted as the process ID of the traced process. The *Buffer* parameter is ignored.

If this request is unsuccessful, –1 is returned and the **errno** global variable is set to the following code:

EIO The *Data* parameter is not a valid floating–point register. The *Data* parameter must be in the range 256–287.

PTT_READ_FPRS
This request writes the contents of the 32 floating point registers to the area specified by the *Address* parameter. This area must be at least 256 bytes long. The *Identifier* parameter specifies the traced kernel thread. The *Data* and *Buffer* parameters are ignored.

PT_READ_GPR
This request returns the contents of one of the general–purpose or special–purpose registers of the debugged process. The *Address* parameter specifies the register whose value is returned. The value of the *Address* parameter is defined in the **sys/reg.h** file for the machine type on which the process is executed. The *Identifier* parameter is interpreted as the process ID of the traced process. The *Data* and *Buffer* parameters are ignored. The buffer points to long long target area.

Note: If **ptracex** with a 64–bit debugee is used for this request, the register value is instead returned to the 8–byte area pointed to by the buffer pointer.

If this request is unsuccessful, –1 is returned and the **errno** global variable is set to the following code:

EIO The *Address* is not a valid general–purpose or special–purpose register. The *Address* parameter must be in the range 0–31 or 128–136.

PTT_READ_GPRS
This request writes the contents of the 32 general purpose registers to the area specified by the *Address* parameter. This area must be at least 128 bytes long.

Note: If **ptracex** with a 64–bit debugee is used for the **PTT_READ_GPRS** request, there must be at least a 256 byte target area. The *Identifier* parameter specifies the traced kernel thread. The *Data* and *Buffer* parameters are ignored.

PT_READ_I or **PT_READ_D**
These requests return the word–aligned address in the debugged process address space specified by the *Address* parameter. On all machines currently supported by the Version 4 operating system, the **PT_READ_I** and **PT_READ_D** instruction and data requests can be used with equal results. The *Identifier* parameter is interpreted as the process ID of the traced process. The *Data* parameter is ignored.

If this request is unsuccessful, –1 is returned and the **errno** global variable is set to the following code:

EIO The *Address* is not word–aligned, or the *Address* is not valid. User blocks, kernel segments, and kernel extension segments are not considered as valid addresses.

Note: For the **PT_READ_I** or the **PT_READ_D** request, use **ptracex** with a 64–bit debuggee because the source address needs 64 bits.

PTT_READ_SPRS
This request writes the contents of the special purpose registers to the area specified by the *Address* parameter, which points to a **ptsprs** structure. The *Identifier* parameter specifies the traced kernel thread. The *Data* and *Buffer* parameters are ignored.

Note: For the **PTT_READ_SPRS** request, use **ptracex** with the 64–bit debuggee because the new **ptxsprs** structure must be used.

PT_REATT This request allows a new debugger, with the proper permissions, to trace a process that was already traced by another debugger. The *Identifier* parameter is interpreted as the process ID of the traced process. The *Address*, *Data*, and *Buffer* parameters are ignored.

If this request is unsuccessful, –1 is returned and the **errno** global variable is set to one the following codes:

ESRCH The *Identifier* is not valid; or the traced process is a kernel process.

EPERM Real or effective user ID of the debugger does not match that of the traced process, or the debugger does not have root authority.

EINVAL The debugger and the traced process are the same.

PT_REGSET This request writes the contents of all 32 general purpose registers to the area specified by the *Address* parameter. This area must be at least 128 bytes for the 32–bit debuggee or 256 bytes for the 64–bit debuggee. The *Identifier* parameter is interpreted as the process ID of the traced process. The *Data* and *Buffer* parameters are ignored.

If this request is unsuccessful, –1 is returned and the **errno** global variable is set to the following code:

EIO The *Address* parameter points to a location outside of the allocated address space of the process.

Note: For the **PT_REGSET** request, use **ptracex** with the 64–bit debuggee because 64–bit registers requiring 256 bytes are returned.

PT_TRACE_ME
This request must be issued by the debugged process to be traced. Upon receipt of a signal, this request sets the process trace flag, placing the process in a stopped state, rather than the action specified by the **sigaction** subroutine. The *Identifier*, *Address*, *Data*, and *Buffer* parameters are ignored. Do not issue this request if the parent process does not expect to trace the debugged process.

As a security measure, the **ptrace** subroutine inhibits the set–user–ID facility on subsequent **exec** subroutines, as shown in the following example:

```
if((childpid = fork()) == 0)
{ /* child process */
 ptrace(PT_TRACE_ME,0,0,0,0);
   execlp(          )/* your favorite exec*/
   }
else
{    /* parent                          */
     /* wait for child to stop          */
        rc = wait(status)
```

Note: This is the only request that should be performed by the child. The parent should perform all other requests when the child is in a stopped state.

If this request is unsuccessful, –1 is returned and the **errno** global variable is set to the following code:

ESRCH Process is debugged by a process that is not its parent.

PT_WRITE_BLOCK
This request writes a block of data into the debugged process address space. The *Address* parameter points to the location in the process address space to be written into. The *Data* parameter gives the length of the block in bytes, and must not be greater than 1024. The *Identifier* parameter is interpreted as the process ID of the traced process. The *Buffer* parameter points to the location in the debugging process address space where the data is copied. Upon successful completion, the value of the *Data* parameter is returned to the debugging process.

If this request is unsuccessful, –1 is returned and the **errno** global variable is set to one of the following codes:

EIO The *Data* parameter is less than 1 or greater than 1024.

EIO The *Address* parameter is not a valid pointer into the debugged process address space.

EFAULT The *Buffer* parameter does not point to a readable location in the debugging process address space.

Note: For the **PT_WRITE_BLOCK** request, use **ptracex** with the 64–bit debuggee because 64–bit registers requiring 256 bytes are returned.

PT_WRITE_FPR
This request sets the floating–point register specified by the *Data* parameter to the value specified by the *Address* parameter. The *Identifier* parameter is interpreted as the process ID of the traced process. The *Buffer* parameter is ignored.

If this request is unsuccessful, –1 is returned and the **errno** global variable is set to the following code:

EIO The *Data* parameter is not a valid floating–point register. The *Data* parameter must be in the range 256–287.

PTT_WRITE_FPRS
This request updates the contents of the 32 floating point registers with the values specified in the area designated by the *Address* parameter. This area must be at least 256 bytes long. The *Identifier* parameter specifies the traced kernel thread. The *Data* and *Buffer* parameters are ignored.

PT_WRITE_GPR
This request stores the value of the *Data* parameter in one of the process general–purpose or special–purpose registers. The *Address* parameter specifies the register to be modified. Upon successful completion, the value of the *Data* parameter is returned to the debugging process. The *Identifier* parameter is interpreted as the process ID of the traced process. The *Buffer* parameter is ignored.

Note: If **ptracex** with a 64–bit debugee is used for the **PT_WRITE_GPR** request, the new register value is NOT passed via the data parameter, but is instead passed via the 8–byte area pointed to by the buffer parameter.

If this request is unsuccessful, –1 is returned and the **errno** global variable is set to the following code:

EIO The *Address* parameter is not a valid general–purpose or special–purpose register. The *Address* parameter must be in the range 0–31 or 128–136.

PTT_WRITE_GPRS
This request updates the contents of the 32 general purpose registers with the values specified in the area designated by the *Address* parameter. This area must be at least 128 bytes long. The *Identifier* parameter specifies the traced kernel thread. The *Data* and *Buffer* parameters are ignored.

Note: For the **PTT_WRITE_GPRS** request, use **ptracex** with the 64–bit debuggee because 64–bit registers requiring 256 bytes are returned. The buffer points to long long source area.

PT_WRITE_I or **PT_WRITE_D**
These requests write the value of the data parameter into the address space of the debugged process at the word–aligned address specified by the *Address* parameter. On all machines currently supported by the Version 4 operating system, instruction and data address spaces are not separated. The **PT_WRITE_I** and **PT_WRITE_D** instruction and data requests can be used with equal results. Upon successful completion, the value written into the address space of the debugged process is returned to the debugging process. The *Identifier* parameter is interpreted as the process ID of the traced process. The *Buffer* parameter is ignored.

If this request is unsuccessful, –1 is returned and the **errno** global variable is set to the following code:

EIO The *Address* parameter points to a location in a pure procedure space and a copy cannot be made; the *Address* is not word–aligned; or, the *Address* is not valid. User blocks, kernel segments, and kernel extension segments are not considered valid addresses.

Note: For the or **PT_WRITE_I** or **PT_WRITE_D** request, use **ptracex** with a 64–bit debuggee because the target address needs 64 bits.

PTT_WRITE_SPRS
This request updates the special purpose registers with the values in the area specified by the *Address* parameter, which points to a **ptsprs** structure. The *Identifier* parameter specifies the traced kernel thread. The *Data* and *Buffer* parameters are ignored.

Identifier Determined by the value of the *Request* parameter.

Address Determined by the value of the *Request* parameter.

Data Determined by the value of the *Request* parameter.

Buffer Determined by the value of the *Request* parameter.

Note: For the **PTT_READ_SPRS** request, use **ptracex** with the 64–bit debuggee because the new **ptxsprs** structure must be used.

Error Codes

The **ptrace** subroutine is unsuccessful when one of the following is true:

EFAULT The *Buffer* parameter points to a location outside the debugging process address space.

EINVAL The debugger and the traced process are the same; or the *Identifier* parameter does not identify the thread that caused the exception.

EIO The *Request* parameter is not one of the values listed, or the *Request* parameter is not valid for the machine type on which the process is executed.

ENOMEM Either the area is not large enough to accommodate the loader information, or there is not enough memory to allocate an equivalent buffer in the kernel.

EPERM The *Identifier* parameter corresponds to a kernel thread which is stopped in kernel mode and whose computational state cannot be read or written.

ESRCH The *Identifier* parameter identifies a process or thread that does not exist, that has not executed a **ptrace** call with the **PT_TRACE_ME** request, or that is not stopped.

For **ptrace**: If the debuggee is a 64–bit process, the options that refer to GPRs or SPRs fail with errno = **EIO**, and the options that specify addresses are limited to 32–bits.

For **ptracex**: If the debuggee is a 32–bit process, the options that refer to GPRs or SPRs fail with errno = **EIO**, and the options that specify addresses in the debuggee's address space that are larger than 2**32 – 1 fail with errno set to **EIO**.

Also, the options PT_READ_U and PT_WRITE_U are not supported if the debuggee is a 64–bit program (errno = **ENOTSUP**).

Implementation Specifics

This subroutine is part of Base Operating System (BOS) Runtime.

Related Information

The **exec**, **getprocs**, **getthrds**, **load**, **sigaction**, **unload**, **wait**, **waitpid**, or **wait3** subroutine.

The **dbx** command.

ptsname Subroutine

Purpose

Returns the name of a pseudo–terminal device.

Library

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

Syntax

```
#include <stdlib.h>

char *ptsname (FileDescriptor)
int FileDescriptor
```

Description

The **ptsname** subroutine gets the path name of the slave pseudo–terminal associated with the master pseudo–terminal device defined by the *FileDescriptor* parameter.

Parameters

FileDescriptor	Specifies the file descriptor of the master pseudo–terminal device

Return Values

The **ptsname** subroutine returns a pointer to a string containing the null–terminated path name of the pseudo–terminal device associated with the file descriptor specified by the *FileDescriptor* parameter. A null pointer is returned and the **errno** global variable is set to indicate the error if the file descriptor does not describe a pseudo–terminal device in the /**dev** directory.

Files

/**dev**/*	Terminal device special files.

Related Information

The **ttyname** subroutine.

The Input and Output Handling Programmer's Overview in *AIX General Programming Concepts : Writing and Debugging Programs*.

putc, putchar, fputc, or putw Subroutine

Purpose

Writes a character or a word to a stream.

Library

Standard I/O Package (**libc.a**)

Syntax

```
#include <stdio.h>

int putc (Character, Stream)
int Character;
FILE *Stream;

int putchar (Character)
int Character;

int fputc (Character, Stream)
int Character;
FILE *Stream;

int putw (Word, Stream)
int Word;
FILE *Stream;
```

Description

The **putc** and **putchar** macros write a character or word to a stream. The **fputc** and **putw** subroutines serve similar purposes but are true subroutines.

The **putc** macro writes the character *Character* (converted to an **unsigned char** data type) to the output specified by the *Stream* parameter. The character is written at the position at which the file pointer is currently pointing, if defined.

The **putchar** macro is the same as the **putc** macro except that **putchar** writes to the standard output.

The **fputc** subroutine works the same as the **putc** macro, but **fputc** is a true subroutine rather than a macro. It runs more slowly than **putc,** but takes less space per invocation.

Because **putc** is implemented as a macro, it incorrectly treats a *Stream* parameter with side effects, such as **putc(C, *f++)**. For such cases, use the **fputc** subroutine instead. Also, use **fputc** whenever you need to pass a pointer to this subroutine as a parameter to another subroutine.

The **putc** and **putchar** macros have also been implemented as subroutines for ANSI compatibility. To access the subroutines instead of the macros, insert **#undef putc** or **#undef putchar** at the beginning of the source file.

The **putw** subroutine writes the word (**int** data type) specified by the *Word* parameter to the output specified by the *Stream* parameter. The word is written at the position at which the file pointer, if defined, is pointing. The size of a word is the size of an integer and varies from machine to machine. The **putw** subroutine does not assume or cause special alignment of the data in the file.

After the **fputcw**, **putwc**, **fputc**, **putc**, **fputs**, **puts**, or **putw** subroutine runs successfully, and before the next successful completion of a call either to the **fflush** or **fclose** subroutine on the same stream or to the **exit** or **abort** subroutine, the `st_ctime` and `st_mtime` fields of the file are marked for update.

Because of possible differences in word length and byte ordering, files written using the **putw** subroutine are machine–dependent, and may not be readable using the **getw** subroutine on a different processor.

With the exception of **stderr**, output streams are, by default, buffered if they refer to files, or line–buffered if they refer to terminals. The standard error output stream, **stderr**, is unbuffered by default, but using the **freopen** subroutine causes it to become buffered or line–buffered. Use the **setbuf** subroutine to change the stream buffering strategy.

When an output stream is unbuffered, information is queued for writing on the destination file or terminal as soon as it is written. When an output stream is buffered, many characters are saved and written as a block. When an output stream is line–buffered, each line of output is queued for writing on the destination terminal as soon as the line is completed (that is, as soon as a new–line character is written or terminal input is requested).

Parameters

Stream	Points to the file structure of an open file.
Character	Specifies a character to be written.
Word	Specifies a word to be written (not portable because word length and byte–ordering are machine–dependent).

Return Values

Upon successful completion, these functions each return the value written. If these functions fail, they return the constant **EOF**. They fail if the *Stream* parameter is not open for writing, or if the output file size cannot be increased. Because the **EOF** value is a valid integer, you should use the **ferror** subroutine to detect **putw** errors.

Error Codes

The **fputc** subroutine will fail if either the *Stream* is unbuffered or the *Stream* buffer needs to be flushed, and:

EAGAIN	The **O_NONBLOCK** flag is set for the file descriptor underlying *Stream* and the process would be delayed in the write operation.
EBADF	The file descriptor underlying *Stream* is not a valid file descriptor open for writing.
EFBIG	An attempt was made to write a file that exceeds the file size of the process limit or the maximum file size.
EFBIG	The file is a regular file and an attempt was made to write at or beyond the offset maximum.
EINTR	The write operation was terminated due to the receipt of a signal, and either no data was transferred or the implementation does not report partial transfers for this file. **Note:** Depending upon which library routine the application binds to, this subroutine may return **EINTR**. Refer to the **signal** Subroutine regarding **sa_restart**.
EIO	A physical I/O error has occurred, or the process is a member of a background process group attempting to perform a **write** subroutine to its controlling terminal, the **TOSTOP** flag is set, the process is neither ignoring nor blocking the **SIGTTOU** signal and the process group of the process is orphaned. This error may also be returned under implementation–dependent conditions.

ENOSPC	There was no free space remaining on the device containing the file.
EPIPE	An attempt is made to write to a pipe or first–in–first–out (FIFO) that is not open for reading by any process. A **SIGPIPE** signal will also be sent to the process.

The **fputc** subroutine may fail if:

ENOMEM	Insufficient storage space is available.
ENXIO	A request was made of a nonexistent device, or the request was outside the capabilities of the device.

Implementation Specifics

These subroutines are part of Base Operating System (BOS) Runtime.

Related Information

The **fclose** or **fflush** subroutine, **feof**, **ferror**, **clearerr**, or **fileno** subroutine, **fopen**, **freopen**, or **fdopen** subroutine, **fread** or **fwrite** subroutine, **getc**, **fgetc**, **getchar**, or **getw** subroutine, **getwc**, **fgetwc**, or **getwchar** subroutine, **printf**, **fprintf**, **sprintf**, **NLprintf**, **NLfprintf**, **NLsprintf**, or **wsprintf** subroutine, **putwc**, **fputwc**, or **putwchar** subroutine, **puts** or **fputs** subroutine, **setbuf** subroutine.

putenv Subroutine

Purpose

Sets an environment variable.

Library

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

Syntax

```
int putenv (String)
char *String;
```

Description

Attention: Unpredictable results can occur if a subroutine passes the **putenv** subroutine a pointer to an automatic variable and then returns while the variable is still part of the environment.

The **putenv** subroutine sets the value of an environment variable by altering an existing variable or by creating a new one. The *String* parameter points to a string of the form *Name*=*Value*, where *Name* is the environment variable and *Value* is the new value for it.

The memory space pointed to by the *String* parameter becomes part of the environment, so that altering the string effectively changes part of the environment. The space is no longer used after the value of the environment variable is changed by calling the **putenv** subroutine again. Also, after the **putenv** subroutine is called, environment variables are not necessarily in alphabetical order.

The **putenv** subroutine manipulates the **environ** external variable and can be used in conjunction with the **getenv** subroutine. However, the *EnvironmentPointer* parameter, the third parameter to the main subroutine, is not changed.

The **putenv** subroutine uses the **malloc** subroutine to enlarge the environment.

Parameters

String	A pointer to the *Name*=*Value* string.

Return Values

Upon successful completion, a value of 0 is returned. If the **malloc** subroutine is unable to obtain sufficient space to expand the environment, then the **putenv** subroutine returns a nonzero value.

Implementation Specifics

This subroutine is part of Base Operating System (BOS) Runtime.

Related Information

The **exec: execl**, **execv**, **execle**, **execlp**, **execvp**, or **exect** subroutine, **getenv** subroutine, **malloc** subroutine.

puts or fputs Subroutine

Purpose

Writes a string to a stream.

Library

Standard I/O Library (**libc.a**)

Syntax

```
#include  <stdio.h>

int   puts   (String)
const char   *String;

int   fputs  (String, Stream)
const char   *String;
FILE         *Stream;
```

Description

The **puts** subroutine writes the string pointed to by the *String* parameter to the standard output stream, **stdout**, and appends a new–line character to the output.

The **fputs** subroutine writes the null–terminated string pointed to by the *String* parameter to the output stream specified by the *Stream* parameter. The **fputs** subroutine does not append a new–line character.

Neither subroutine writes the terminating null character.

After the **fputwc**, **putwc**, **fputc**, **fputs**, **puts**, or **putw** subroutine runs successfully, and before the next successful completion of a call either to the **fflush** or **fclose** subroutine on the same stream or a call to the **exit** or **abort** subroutine, the `st_ctime` and `st_mtime` fields of the file are marked for update.

Parameters

String	Points to a string to be written to output.
Stream	Points to the **FILE** structure of an open file.

Return Values

Upon successful completion, the **puts** and **fputs** subroutines return the number of characters written. Otherwise, both subroutines return **EOF**, set an error indicator for the stream and set the **errno** global variable to indicate the error. This happens if the routines try to write to a file that has not been opened for writing.

Error Codes

If the **puts** or **fputs** subroutine is unsuccessful because the output stream specified by the *Stream* parameter is unbuffered or the buffer needs to be flushed, it returns one or more of the following error codes:

EAGAIN	Indicates that the **O_NONBLOCK** flag is set for the file descriptor specified by the *Stream* parameter and the process would be delayed in the write operation.
EBADF	Indicates that the file descriptor specified by the *Stream* parameter is not a valid file descriptor open for writing.

EFBIG Indicates that an attempt was made to write to a file that exceeds the process' file size limit or the systemwide maximum file size.

EINTR Indicates that the write operation was terminated due to receipt of a signal and no data was transferred.

Note: Depending upon which library routine the application binds to, this subroutine may return **EINTR**. Refer to the **signal** subroutine regarding the **SA_RESTART** bit.

EIO Indicates that the process is a member of a background process group attempting to perform a write to its controlling terminal, the **TOSTOP** flag is set, the process is neither ignoring or blocking the **SIGTTOU** signal, and the process group of the process has no parent process.

ENOSPC Indicates that there was no free space remaining on the device containing the file specified by the *Stream* parameter.

EPIPE Indicates that an attempt is made to write to a pipe or first–in–first–out (FIFO) that is not open for reading by any process. A **SIGPIPE** signal will also be sent to the process.

ENOMEM Indicates that insufficient storage space is available.

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

Implementation Specifics

These subroutines are part of Base Operating System (BOS) Runtime.

Related Information

The **fopen**, **freopen**, or **fdopen** subroutine, **fread**, or **fwrite** subroutine, **gets** or **fgets** subroutine, **getws** or **fgetws** subroutine, **printf**, **fprintf**, and **sprintf** subroutine, **putc**, **putchar**, **fputc**, or **putw** subroutine, **putwc**, **putwchar**, or **fputwc** subroutine, **putws** or **fputws** subroutine.

The **feof**, **ferror**, **clearerr**, or **fileno** macros.

Subroutines Overview in *AIX General Programming Concepts : Writing and Debugging Programs*.

putwc, putwchar, or fputwc Subroutine

Purpose

Writes a character or a word to a stream.

Library

Standard I/O Library (**libc.a**)

Syntax

```
#include <stdio.h>

wint_t putwc(Character, Stream)
wint_t Character;
FILE *Stream;

wint_t putwchar(Character)
wint_t Character;

wint_t fputwc(Character, Stream)
wint_t Character;
FILE Stream;
```

Description

The **putwc** subroutine writes the wide character specified by the *Character* parameter to the output stream pointed to by the *Stream* parameter. The wide character is written as a multibyte character at the associated file position indicator for the stream, if defined. The subroutine then advances the indicator. If the file cannot support positioning requests, or if the stream was opened with append mode, the character is appended to the output stream.

The **putwchar** subroutine works like the **putwc** subroutine, except that **putwchar** writes the specified wide character to the standard output.

The **fputwc** subroutine works the same as the **putwc** subroutine.

Output streams, with the exception of **stderr**, are buffered by default if they refer to files, or line–buffered if they refer to terminals. The standard error output stream, **stderr**, is unbuffered by default, but using the **freopen** subroutine causes it to become buffered or line–buffered. Use the **setbuf** subroutine to change the stream's buffering strategy.

After the **fputwc**, **putwc**, **fputc**. **putc**, **fputs**, **puts**, or **putw** subroutine runs successfully, and before the next successful completion of a call either to the **fflush** or **fclose** subroutine on the same stream or to the **exit** or **abort** subroutine, the `st_ctime` and `st_mtime` fields of the file are marked for update.

Parameters

Character	Specifies a wide character of type **wint_t**.
Stream	Specifies a stream of output data.

Return Values

Upon successful completion, the **putwc**, **putwchar**, and **fputwc** subroutines return the wide character that is written. Otherwise **WEOF** is returned, the error indicator for the stream is set, and the **errno** global variable is set to indicate the error.

Error Codes

If the **putwc**, **putwchar**, or **fputwc** subroutine fails because the stream is not buffered or data in the buffer needs to be written, it returns one or more of the following error codes:

EAGAIN	Indicates that the **O_NONBLOCK** flag is set for the file descriptor underlying the *Stream* parameter, delaying the process during the write operation.
EBADF	Indicates that the file descriptor underlying the *Stream* parameter is not valid and cannot be updated during the write operation.
EFBIG	Indicates that the process attempted to write to a file that already equals or exceeds the file–size limit for the process. The file is a regular file and an attempt was made to write at or beyond the offset maximum associated with the corresponding stream.
EILSEQ	Indicates that the wide–character code does not correspond to a valid character.
EINTR	Indicates that the process has received a signal that terminates the read operation.
EIO	Indicates that the process is in a background process group attempting to perform a write operation to its controlling terminal. The **TOSTOP** flag is set, the process is not ignoring or blocking the **SIGTTOU** flag, and the process group of the process is orphaned.
ENOMEM	Insufficient storage space is available.
ENOSPC	Indicates that no free space remains on the device containing the file.
ENXIO	Indicates a request was made of a non–existent device, or the request was outside the capabilities of the device.
EPIPE	Indicates that the process has attempted to write to a pipe or first–in–first–out (FIFO) that is not open for reading. The process will also receive a **SIGPIPE** signal.

Implementation Specifics

These subroutines are part of Base Operating System (BOS) Runtime.

Related Information

Other wide character I/O subroutines: **fgetwc** subroutine, **fgetws** subroutine, **fputws** subroutine, **getwc** subroutine, **getwchar** subroutine, **getws** subroutine, **putws** subroutine, **ungetwc** subroutine.

Related standard I/O subroutines: **fdopen** subroutine, **fgets** subroutine, **fopen** subroutine, **fprintf** subroutine, **fputc** subroutine, **fputs** subroutine, **fread** subroutine, **freopen** subroutine, **fwrite** subroutine, **gets** subroutine, **printf** subroutine, **putc** subroutine, **putchar** subroutine, **puts** subroutine, **putw** subroutine, **sprintf** subroutine.

National Language Support Overview for Programming, Subroutines Overview, Understanding Wide Character Input/Output Subroutines in *AIX General Programming Concepts : Writing and Debugging Programs*.

putws or fputws Subroutine

Purpose

Writes a wide–character string to a stream.

Library

Standard I/O Library (**libc.a**)

Syntax

```
#include <stdio.h>

int putws (String)
const wchar_t *String;

int fputws (String, Stream)
const wchar_t *String;
FILE *Stream;
```

Description

The **putws** subroutine writes the **const wchar_t** string pointed to by the *String* parameter to the standard output stream (**stdout**) as a multibyte character string and appends a new–line character to the output. In all other respects, the **putws** subroutine functions like the **puts** subroutine.

The **fputws** subroutine writes the **const wchar_t** string pointed to by the *String* parameter to the output stream as a multibyte character string. In all other respects, the **fputws** subroutine functions like the **fputs** subroutine.

After the **putws** or **fputws** subroutine runs successfully, and before the next successful completion of a call to the **fflush** or **fclose** subroutine on the same stream or a call to the **exit** or **abort** subroutine, the `st_ctime` and `st_mtime` fields of the file are marked for update.

Parameters

String	Points to a string to be written to output.
Stream	Points to the **FILE** structure of an open file.

Return Values

Upon successful completion, the **putws** and **fputws** subroutines return a nonnegative number. Otherwise, a value of –1 is returned, and the **errno** global variable is set to indicate the error.

Error Codes

The **putws** or **fputws** subroutine is unsuccessful if the stream is not buffered or data in the buffer needs to be written, and one of the following errors occur:

EAGAIN	The **O_NONBLOCK** flag is set for the file descriptor underlying the *Stream* parameter, which delays the process during the write operation.
EBADF	The file descriptor underlying the *Stream* parameter is not valid and cannot be updated during the write operation.
EFBIG	The process attempted to write to a file that already equals or exceeds the file–size limit for the process.
EINTR	The process has received a signal that terminates the read operation.

EIO	The process is in a background process group attempting to perform a write operation to its controlling terminal. The **TOSTOP** flag is set, the process is not ignoring or blocking the **SIGTTOU** flag, and the process group of the process is orphaned.
ENOSPC	No free space remains on the device containing the file.
EPIPE	The process has attempted to write to a pipe or first–in–first–out (FIFO) that is not open for reading. The process also receives a **SIGPIPE** signal.
EILSEQ	The **wc** wide–character code does not correspond to a valid character.

Implementation Specifics

These subroutines are part of Base Operating System (BOS) Runtime.

Related Information

Other wide–character I/O subroutines: **fgetwc** subroutine, **fgetws** subroutine, **fputwc** subroutine, **getwc** subroutine, **getwchar** subroutine, **getws** subroutine, **putwc** subroutine, **putwchar** subroutine, **ungetwc** subroutine.

Related standard I/O subroutines: **fdopen** subroutine, **fgets** subroutine, **fopen** subroutine, **fprintf** subroutine, **fputc** subroutine, **fputs** subroutine, **fread** subroutine, **freopen** subroutine, **fwrite** subroutine, **gets** subroutine, **printf** subroutine, **putc** subroutine, **putchar** subroutine, **puts** subroutine, **putw** subroutine, **sprintf** subroutine.

National Language Support Overview for Programming, Subroutines Overview, Understanding Wide Character Input/Output Subroutines in *AIX General Programming Concepts : Writing and Debugging Programs*.

pwdrestrict_method Subroutine

Purpose

Defines loadable password restriction methods.

Library

Syntax

```
int pwdrestrict_method (UserName, NewPassword, OldPassword,
Message)
char *UserName;
char *NewPassword;
char *OldPassword;
char **Message;
```

Description

The **pwdrestrict_method** subroutine extends the capability of the password restrictions software and lets an administrator enforce password restrictions that are not provided by the system software.

Whenever users change their passwords, the system software scans the **pwdchecks** attribute defined for that user for site specific restrictions. Since this attribute field can contain load module file names, for example, methods, it is possible for the administrator to write and install code that enforces site specific password restrictions.

The system evaluates the **pwdchecks** attribute's value field in a left to right order. For each method that the system encounters, the system loads and invokes that method. The system uses the **load** subroutine to load methods. It invokes the **load** subroutine with a *Flags* value of **1** and a *LibraryPath* value of /**usr**/**lib**. Once the method is loaded, the system invokes the method.

To create a loadable module, use the **–e** flag of the **ld** command. Note that the name **pwdrestrict_method** given in the syntax is a generic name. The actual subroutine name can be anything (within the compiler's name space) except **main**. What is important is, that for whatever name you choose, you must inform the **ld** command of the name so that the **load** subroutine uses that name as the entry point into the module. In the following example, the C compiler compiles the **pwdrestrict.c** file and pass **–e pwdrestrict_method** to the **ld** command to create the method called **pwdrestrict**:

```
cc –e pwdrestrict_method –o pwdrestrict pwdrestrict.c
```

The convention of all password restriction methods is to pass back messages to the invoking subroutine. Do not print messages to stdout or stderr. This feature allows the password restrictions software to work across network connections where stdout and stderr are not valid. Note that messages must be returned in dynamically allocated memory to the invoking program. The invoking program will deallocate the memory once it is done with the memory.

There are many caveats that go along with loadable subroutine modules:

1. The values for *NewPassword* and *OldPassword* are the actual clear text passwords typed in by the user. If you copy these passwords into other parts of memory, clear those memory locations before returning back to the invoking program. This helps to prevent clear text passwords from showing up in core dumps. Also, do not copy these passwords

into a file or anywhere else that another program can access. Clear text passwords should never exist outside of the process space.

2. Do not modify the current settings of the process' signal handlers.
3. Do not call any functions that will terminate the execution of the program (for example, the **exit** subroutine, the **exec** subroutine). Always return to the invoking program.
4. The code must be thread–safe.
5. The actual load module must be kept in a write protected environment. The load module and directory should be writable only by the root user.

One last note, all standard password restrictions are performed before any of the site specific methods are invoked. Thus, methods are the last restrictions to be enforced by the system.

Parameters

UserName	Specifies a "local" user name.
NewPassword	Specifies the new password in clear text (not encrypted).This value may be a NULL pointer. Clear text passwords are always in 7 bit ASCII.
OldPassword	Specifies the current password in clear text (not encrypted).This value may be a NULL pointer. Clear text passwords are always in 7 bit ASCII.
Message	Specifies the address of a pointer to **malloc**'ed memory containing an NLS error message. The method is expected to supply the **malloc**'ed memory and the message.

Return Values

The method is expected to return the following values. The return values are listed in order of precedence.

–1	Internal error. The method could not perform its password evaluation. The method must set the **errno** variable. The method must supply an error message in *Message* unless it can't allocate memory for the message. If it cannot allocate memory, then it must return the NULL pointer in *Message*.
1	Failure. The password change did not meet the requirements of the restriction. The password restriction was properly evaluated and the password change was not accepted. The method must supply an error message in *Message*. The **errno** variable is ignored. Note that composition failures are cumulative, thus, even though a failure condition is returned, trailing composition methods will be invoked.
0	Success. The password change met the requirements of the restriction. If necessary, the method may supply a message in *Message*; otherwise, return the NULL pointer. The **errno** variable is ignored.

Appendix A. Base Operating System Error Codes for Services That Require Path–Name Resolution

The following errors apply to any service that requires path name resolution:

EACCES	Search permission is denied on a component of the path prefix.
EFAULT	The *Path* parameter points outside of the allocated address space of the process.
EIO	An I/O error occurred during the operation.
ELOOP	Too many symbolic links were encountered in translating the *Path* parameter.
ENAMETOOLONG	A component of a path name exceeded 255 characters and the process has the **DisallowTruncation** attribute (see the **ulimit** subroutine) or an entire path name exceeded 1023 characters.
ENOENT	A component of the path prefix does not exist.
ENOENT	A symbolic link was named, but the file to which it refers does not exist.
ENOENT	The path name is null.
ENOTDIR	A component of the path prefix is not a directory.
ESTALE	The root or current directory of the process is located in a virtual file system that is unmounted.

Related Information

List of File and Directory Manipulation Services.

Appendix B. ODM Error Codes

When an ODM subroutine is unsuccessful, a value of –1 is returned and the **odmerrno** variable is set to one of the following values:

ODMI_BAD_CLASSNAME The specified object class name does not match the object class name in the file. Check path name and permissions.

ODMI_BAD_CLXNNAME The specified collection name does not match the collection name in the file.

ODMI_BAD_CRIT The specified search criteria is incorrectly formed. Make sure the criteria contains only valid descriptor names and the search values are correct. For information on qualifying criteria, see ”Understanding ODM Object Searches” in *AIX General Programming Concepts : Writing and Debugging Programs*.

ODMI_BAD_LOCK Cannot set a lock on the file. Check path name and permissions.

ODMI_BAD_TIMEOUT The time–out value was not valid. It must be a positive integer.

ODMI_BAD_TOKEN Cannot create or open the lock file. Check path name and permissions.

ODMI_CLASS_DNE The specified object class does not exist. Check path name and permissions.

ODMI_CLASS_EXISTS The specified object class already exists. An object class must not exist when it is created.

ODMI_CLASS_PERMS The object class cannot be opened because of the file permissions.

ODMI_CLXNMAGICNO_ERR The specified collection is not a valid object class collection.

ODMI_FORK Cannot fork the child process. Make sure the child process is executable and try again.

ODMI_INTERNAL_ERR An internal consistency problem occurred. Make sure the object class is valid or contact the person responsible for the system.

ODMI_INVALID_CLASS The specified file is not an object class.

ODMI_INVALID_CLXN Either the specified collection is not a valid object class collection or the collection does not contain consistent data.

ODMI_INVALID_PATH The specified path does not exist on the file system. Make sure the path is accessible.

ODMI_LINK_NOT_FOUND The object class that is accessed could not be opened. Make sure the linked object class is accessible.

ODMI_LOCK_BLOCKED Cannot grant the lock. Another process already has the lock.

ODMI_LOCK_ENV Cannot retrieve or set the lock environment variable. Remove some environment variables and try again.

ODMI_LOCK_ID	The lock identifier does not refer to a valid lock. The lock identifier must be the same as what was returned from the **odm_lock** subroutine.
ODMI_MAGICNO_ERR	The class symbol does not identify a valid object class.
ODMI_MALLOC_ERR	Cannot allocate sufficient storage. Try again later or contact the person responsible for the system.
ODMI_NO_OBJECT	The specified object identifier did not refer to a valid object.
ODMI_OPEN_ERR	Cannot open the object class. Check path name and permissions.
ODMI_OPEN_PIPE	Cannot open a pipe to a child process. Make sure the child process is executable and try again.
ODMI_PARAMS	The parameters passed to the subroutine were not correct. Make sure there are the correct number of parameters and that they are valid.
ODMI_READ_ONLY	The specified object class is opened as read–only and cannot be modified.
ODMI_READ_PIPE	Cannot read from the pipe of the child process. Make sure the child process is executable and try again.
ODMI_TOOMANYCLASSES	Too many object classes have been accessed. An application can only access less than 1024 object classes.
ODMI_UNLINKCLASS_ERR	Cannot remove the object class from the file system. Check path name and permissions.
ODMI_UNLINKCLXN_ERR	Cannot remove the object class collection from the file system. Check path name and permissions.
ODMI_UNLOCK	Cannot unlock the lock file. Make sure the lock file exists.

Related Information

List of ODM Commands and Subroutines in *AIX General Programming Concepts : Writing and Debugging Programs*.

Index

Symbols

Numbers

A

D

E

G

J

M

N

O

P

Q

R

S

U

V

W

Y

Vos remarques sur ce document / Technical publication remark form

Titre / Title : Bull Technical Reference Base Operating System and Extensions Volume 1/2

Nº Reférence / Reference Nº : 86 A2 81AP 05

Daté / Dated : February 1999

ERREURS DETECTEES / ERRORS IN PUBLICATION

AMELIORATIONS SUGGEREES / SUGGESTIONS FOR IMPROVEMENT TO PUBLICATION

Vos remarques et suggestions seront examinées attentivement.
Si vous désirez une réponse écrite, veuillez indiquer ci-après votre adresse postale complète.

Your comments will be promptly investigated by qualified technical personnel and action will be taken as required.
If you require a written reply, please furnish your complete mailing address below.

NOM / NAME : ______________________________ Date : __________

SOCIETE / COMPANY : ______________________________

ADRESSE / ADDRESS : ______________________________

____

Remettez cet imprimé à un responsable BULL ou envoyez-le directement à :

Please give this technical publication remark form to your BULL representative or mail to:

BULL ELECTRONICS ANGERS
CEDOC
34 Rue du Nid de Pie – BP 428
49004 ANGERS CEDEX 01
FRANCE

Technical Publications Ordering Form

Bon de Commande de Documents Techniques

To order additional publications, please fill up a copy of this form and send it via mail to:
Pour commander des documents techniques, remplissez une copie de ce formulaire et envoyez-la à :

BULL ELECTRONICS ANGERS
CEDOC
ATTN / MME DUMOULIN
34 Rue du Nid de Pie – BP 428
49004 ANGERS CEDEX 01
FRANCE

Managers / Gestionnaires :
Mrs. / Mme : **C. DUMOULIN** +33 (0) 2 41 73 76 65
Mr. / M : **L. CHERUBIN** +33 (0) 2 41 73 63 96

FAX : +33 (0) 2 41 73 60 19
E–Mail / Courrier Electronique : srv.Cedoc@franp.bull.fr

Or visit our web site at: / Ou visitez notre site web à:
`http://www-frec.bull.com` (PUBLICATIONS, Technical Literature, Ordering Form)

CEDOC Reference # Nº Référence CEDOC	**Qty** Qté	**CEDOC Reference #** Nº Référence CEDOC	**Qty** Qté	**CEDOC Reference #** Nº Référence CEDOC	**Qty** Qté
__ __ ____ _ [__]		__ __ ____ _ [__]		__ __ ____ _ [__]	
__ __ ____ _ [__]		__ __ ____ _ [__]		__ __ ____ _ [__]	
__ __ ____ _ [__]		__ __ ____ _ [__]		__ __ ____ _ [__]	
__ __ ____ _ [__]		__ __ ____ _ [__]		__ __ ____ _ [__]	
__ __ ____ _ [__]		__ __ ____ _ [__]		__ __ ____ _ [__]	
__ __ ____ _ [__]		__ __ ____ _ [__]		__ __ ____ _ [__]	
__ __ ____ _ [__]		__ __ ____ _ [__]		__ __ ____ _ [__]	
[_ _] : **no revision number means latest revision** / pas de numéro de révision signifie révision la plus récente					

NOM / NAME : ______________________________ Date : ____________

SOCIETE / COMPANY : ______________________________

ADRESSE / ADDRESS : ______________________________

____

PHONE / TELEPHONE : ____________________ FAX : ____________________

E–MAIL : ______________________________

For Bull Subsidiaries / Pour les Filiales Bull :
Identification: ______________________________

For Bull Affiliated Customers / Pour les Clients Affiliés Bull :
Customer Code / Code Client : ______________________________

For Bull Internal Customers / Pour les Clients Internes Bull :
Budgetary Section / Section Budgétaire : ______________________________

For Others / Pour les Autres :
Please ask your Bull representative. / Merci de demander à votre contact Bull.

PLACE BAR CODE IN LOWER LEFT CORNER

BULL ELECTRONICS ANGERS
CEDOC
34 Rue du Nid de Pie – BP 428
49004 ANGERS CEDEX 01
FRANCE

ORDER REFERENCE
86 A2 81AP 05

Bull

Utiliser les marques de découpe pour obtenir les étiquettes.
Use the cut marks to get the labels.

AIX

Technical
Reference
Base Operating
System and
Extensions

Volume 1/2
86 A2 81AP 05

AIX

Technical
Reference
Base Operating
System and
Extensions

Volume 1/2
86 A2 81AP 05

AIX

Technical
Reference
Base Operating
System and
Extensions

Volume 1/2
86 A2 81AP 05