AIX Version 3 for RISC System/6000™

Calls and Subroutines Reference: Base Operating System

Volume 1

SC23-21

AIX Version 3 for
RISC System/6000™

Calls and Subroutines Reference: Base Operating System

Volume 1

First Edition (March 1990)

This edition of the *AIX Calls and Subroutines Reference for IBM RISC System/6000* applies to IBM AIX Version 3 for RISC System/6000, Version 3 of IBM AIXwindows Environment/6000, IBM AIX System Network Architecture Services/6000, IBM AIX 3270 Host Connection Program/6000, IBM AIX 3278/79 Emulation/6000, IBM AIX Network Management/6000, and IBM AIX Personal Computer Simulator/6000 and to all subsequent releases of these products until otherwise indicated in new releases or technical newsletters.

Requests for copies of this publication and for technical information about IBM products should be made to your IBM Authorized Dealer or your IBM Marketing Representative.

A reader's comment form is provided at the back of this publication. If the form has been removed, address comments to IBM Corporation, Department 997, 11400 Burnet Road, Austin, Texas 78758–3493. IBM may use or distribute whatever information you supply in any way it believes appropriate without incurring any obligation to you.

This software is derived in part from the ISO Development Environment (ISODE). IBM acknowledges source author Marshall Rose and the following institutions for their role in its development: The Northrup Corporation and The Wollongong Group.

Trademarks and Acknowledgements

Note to Users

The term “network information services (NIS)” is now used to refer to the service formerly known as “Yellow Pages.” The functionality remains the same; only the name has changed. The name “Yellow Pages” is a registered trademark in the United Kingdom of British Telecommunications plc, and may not be used without permission.

Legal Notice to Users Issued by Sun Microsystems, Inc.

“Yellow Pages” is a registered trademark in the United Kingdom of British Telecommunications plc, and may also be a trademark of various telephone companies around the world. Sun will be revising future versions of software and documentation to remove references to “Yellow Pages.”

About This Book

This book, *Calls and Subroutines Reference: Base Operating System*, provides information on application programming interfaces to the Advanced Interactive Executive Operating System (referred to in this text as AIX) for use on the IBM RISC System/6000 System. This book is part of *AIX Calls and Subroutines Reference for IBM RISC System/6000*, SC23–2198, which is divided into the following four major sections:

- Volumes 1 and 2, *Calls and Subroutines Reference*: *Base Operating System*, contains reference information about the system calls, subroutines, functions, macros, and statements associated with AIX base operating system runtime services, communications services, and devices services.
- Volumes 3 and 4, *Calls and Subroutines Reference*: *User Interface*, contain reference information about the AIXwindows widget classes, subroutines, and resource sets; the AIXwindows Desktop resource sets; the Enhanced X–Windows subroutines, macros, protocols, extensions, and events; the X–Window toolkit subroutines and macros; and the curses and extended curses subroutine libraries.
- Volume 5, *Calls and Subroutines Reference*: *Kernel Reference*, contains reference information about kernel services, device driver operations, file system operations subroutines, the configuration subsystem, the communications subsystem, the high function terminal (HFT) subsystem, the logical volume subsystem, the printer subsystem, and the SCSI subsystem.
- Volumes 6, *Calls and Subroutines Reference*: *Graphics*, contains reference information and example programs for the Graphics Library (GL) and the AIXwindows Graphics Support Library (XGSL) subroutines.

Who Should Use This Book

This book is intended for experienced C programmers. To use this book effectively, you should be familiar with AIX or UNIX System V commands, system calls, subroutines, file formats, and special files. If you are not already familiar with the AIX operating system or the UNIX System V operating system, see *AIX General Concepts and Procedures.*

How to Use This Book

Overview of Contents

This book contains the following alphabetically arranged sections consisting of system calls, subroutines, functions, macros and statements. In this book all system calls are described as subroutines.

- Base Operating System Runtime (BOS) Services
- Communications Services
  - SNA Services
  - AIX 3270 Host Connection Program (HCON)
  - Remote Procedure Calls (RPC)
  - Sockets
  - Simple Network Management Protocol (SNMP)
  - Network Computing System (NCS)

- Data Link Controls
- X.25 Application

- Devices Services

Highlighting

The following highlighting conventions are used in this book:

Bold	Identifies commands, keywords, files, directories, and other items whose names are predefined by the system.
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.

Related Publications

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

- *AIX General Programming Concepts for IBM RISC System/6000*, Order Number SC23–2205.
- *AIX Communication Programming Concepts for IBM RISC System/6000*, Order Number SC23–2206.
- *AIX Kernel Extensions and Device Support Programming Concepts for IBM RISC System/6000*, Order Number SC23–2207.
- *AIX Files Reference for IBM RISC System/6000*, Order Number SC23–2200.
- *IBM RISC System/6000 Problem Solving Guide*, Order Number SC23–2204.
- *XL C Language Reference for IBM AIX Version 3 for RISC System/6000*, Order Number SC09–1260.
- *XL C User's Guide for IBM AIX Version 3 for RISC System/6000*, Order Number SC09–1259.

Ordering Additional Copies of This Book

To order additional copies of this book, use Order Number SC23–2198.

Contents

Base Operating System (BOS) Runtime Services

a64l or l64a Subroutine

Purpose

Converts between long integers and base-64 ASCII strings.

Library

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

Syntax

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	represent	2-11
A-Z	represent	12-37
a-z	represent	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.

Implementation Specifics

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

abort Subroutine

Purpose

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

Library

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

Syntax

int abort ( )

Description

The **abort** subroutine causes a **SIGIOT** signal to be sent to the current process. This usually terminates the process and produces a memory dump.

It is possible for the **abort** subroutine to return control if the **SIGIOT** signal is caught or ignored. In this case, the **abort** subroutine returns the value returned by the **kill** subroutine.

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. The shell then displays the following message:

```
abort — core dumped
```

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

Implementation Specifics

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

Related Information

The **exit, atexit, _exit** subroutine, **kill, killpg** subroutines, **sigaction, sigvec, signal** subroutines.

The **dbx** command.

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

Purpose

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

Library

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

Syntax

int abs (*i* **)**
int *i*;

long labs (*i* **)**
long *i*;

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

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

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

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

Description

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

Note: A two's-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 zero), the behavior is undefined.

The **labs** subroutine and **ldiv** subroutine are included for compatibility with the ANSI C library, and accept long integers as parameters, rather than as integers. However, on all systems supported by AIX for RISC System/6000, there is no difference between an integer and a long integer.

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.

Parameters

i Specifies, for **abs,** some integer; for **labs** and **imul_dbl,** some long integer; for **umul_dbl,** some unsigned long integer.

Numerator Specifies, for **div,** some integer; for **ldiv,** some long integer.

j Specifies, for **imul_dbl,** some long integer; for **umul_dbl,** some unsigned long integer.

Denominator Specifies, for **div,** some integer; for **ldiv**, some long integer.

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

Return Values

The **abs** and **labs** 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 */
};
```

Implementation Specifics

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

The **imul_dbl** subroutine and **umul_dbl** subroutine are not included in the ANSI C Library.

Related information

The **floor, ceil, nearest, trunc, itrunc, uitrunc, fmod, fabs** subroutines.

access Subroutine

Purpose

Determines the accessibility of a file.

Library

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

Syntax

#include <sys/access.h>
int access (*Path*, *AccessMode***)**
char **Path*;
int *AccessMode*;

Description

The **access** subroutine checks the accessibility of the file, using the path name.

Parameters

Path Points to the full path name. If the *Path* parameter refers to a symbolic link, the **access** subroutine returns information about the file pointed to by the symbolic link.

Access permission to all components of the *Path* parameter is determined using the *real user ID* instead of the *effective user ID*, the group access list (including the *real group ID*) instead of the *effective group ID*, and the *inherited privilege set* instead of the *effective privilege set*.

AccessMode Specifies the type of access. The bit pattern contained in the *AccessMode* parameter is constructed by logically ORing the following values:

R_ACC Checks read permission.

W_ACC Checks write permission.

X_ACC Checks execute (search) permission.

E_ACC Checks to see if the file exists.

Return Values

If the requested access is permitted, the **access** subroutine returns a value of 0. If the requested access is denied, it returns a value of –1 and sets the global variable **errno** to indentify the error.

Error Codes

Access to the file is denied if one or more of the following are true:

ENOENT The named file does not exist.

EACCES	Permission bits of the file mode do not permit the requested access.
EROFS	Write access is requested for a file on a read–only file system.

The **access** subroutine can also fail if additional errors on page A–1 occur.

If Network File System is installed on the system, the **access** subroutine can also fail if the following is true:

ETIMEDOUT	The connection timed out.

Implementation Specifics

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

Related Information

The **chmod**, **fchmod** subroutines, **statx** subroutine.

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 is installed on your system, the accounting file can reside on another node.

Warning: To ensure accurate accounting, each node must have its own accounting file, which can be located on any node in the network.

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

Parameter

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 fails if one or more of the following are true:

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

The **acct** subroutine can also fail if additional errors on page A–1 occur.

If Network File System is installed on the system, the **acct** subroutine can also fail if the following is true:

ETIMEDOUT	The connection timed out.

Implementation Specifics

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

The BSD **acct** subroutine can be used to switch an accounting file; this is not the case with the AIX Version 3 Operating System **acct** subroutine.

Related Information

The **_exit, exit, atexit** subroutines, **raise** subroutine, **sigaction**, **signal**, **sigvec** subroutines.

The **acct** file.

acl_chg or acl_fchg Subroutine

Purpose

Changes the access control information on a file.

Library

Security Library (**libs.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 ACL. This parameter must be one of:

- **ACC_PERMIT** — Allow the types of access included in the *Mode* parameter.
- **ACC_DENY** — Deny 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 must be one of:

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 global variable **errno** 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:

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.

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

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

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

EBADF The file descriptor *FileDescriptor* 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:

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

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

EINVAL The *Mode* parameter contained values other than **R_ACC**, **W_ACC**, or **X_ACC**.

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

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 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 AIX Base Operating System (BOS) Runtime.

Related Information

The **chacl** subroutine, **statacl** subroutine, **chmod** subroutine, **stat** subroutine.

The **acl_get** subroutine, **acl_put** subroutine, **acl_set** subroutine

The **acl_get** command, **acl_put** command, **chmod** command.

acl_get or acl_fget Subroutine

Purpose

Gets the access control information of a file.

Library

Security Library (**libs.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 pathname 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 global variable **errno** is set to indicate the error.

Error Codes

The **acl_get** subroutine fails 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 the process has the disallow truncation attribute (see the **ulimit** subroutine).
ENOENT	The *Path* parameter was null.
EACCESS	Search permission is denied on a component of the *Path* prefix.
EFAULT	The *Path* parameter points to a location outside of the allocated address space of the process.
ESTALE	The process's 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 **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 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.

Auditable Events None.

Implementation Specifics

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

Related Information

The **chacl** subroutine, **statacl** subroutine, **chmod** subroutine, **stat** subroutine.

The **acl_chg, acl_fchg** subroutines, **acl_put, acl_fput** subroutines, **acl_set, acl_fset** subroutines.

The **acl_get** command, **acl_put** command, **chmod** command.

acl_put or acl_fput Subroutine

Purpose

Sets the access control information of a file.

Library

Security Library (**libs.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** subroutines. The structure of the data in this buffer is unspecified.

Parameters

Path Specifies the pathname 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 Means the space is not deallocated.
- 1 Means the 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 global variable **errno** 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:

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.

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

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

ESTALE The process's 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 **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:

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

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.

If 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.

Auditable Events

Event Name	Tail Information
chacl	*Path*
fchacl	*FileDescriptor*

Implementation Specifics

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

Related Information

The **chacl** subroutine, **statacl** subroutine, **chmod** subroutine, **stat** subroutine.

The **acl_chg** subroutine, **acl_get** subroutine, **acl_set** subroutine.

The **acl_get** command, **acl_put** command, **chmod** command.

acl_set or acl_fset Subroutine

Purpose

Sets the access control information of a file.

Library

Security Library (**libs.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 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 bitmask 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 global variable **errno** 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:

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.

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

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

ESTALE The process's 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 **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:

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

EINVAL One of the *Mode* parameters contained values other than **R_ACC**, **W_ACC**, or **X_ACC**.

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 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.

Auditable Events

Event Name	Tail Information
chacl	*Path*
fchacl	*FileDescriptor*

Implementation Specifics

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

Related Information

The **chacl** subroutine, **statacl** subroutine, **chmod** subroutine, **stat** subroutine.

The **acl_get** subroutine, **acl_put** subroutine, **acl_chg** subroutine.

The **acl_get** command, **acl_put** command, **chmod** command.

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 <sys/spc.h>

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

Description

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

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

Parameter

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 **odmerrno** 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_NONAME	No subsystem name specified.
SRC_NOPATH	No subsystem path specified.
SRC_BADNSIG	Invalid stop normal signal.
SRC_BADFSIG	Invalid stop force signal.
SRC_NOCONTACT	Contact not signal, sockets, or message queue
SRC_SUBEXIST	New subsystem name already on file.
SRC_SYNEXIST	New subsystem synonym name already on file.
SRC_SUBSYS2BIG	Subsystem name too long.
SRC_SYN2BIG	Synonym name too long.
SRC_CMDARG2BIG	Command arguments too long.

SRC_PATH2BIG	Subsystem path too long.
SRC_STDIN2BIG	stdin path too long.
SRC_STDOUT2BIG	stdout path too long.
SRC_STDERR2BIG	stderr path too long.
SRC_GRPNAM2BIG	Group name too long.

Implementation Specifics

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

File

/etc/objrepos/SRCsubsys	SRC Subsystem Configuration object class.

Related Information

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

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

The System Resource Controller Overview in *General Programming Concepts.*

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 *Delta* is negative, the clock is slowed down by incrementing it more slowly than normal until the correction is complete. If *Delta* 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. A time correction from an earlier call to **adjtime** may not be finished when **adjtime** is called again. If the *Olddelta* parameter is non–zero, 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 he error.

Error Codes

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

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

Implementation Specifics

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

Related Information

The **gettimeofday, settimeofday, ftime** subroutines, **gettimer** subroutine.

asinh, acosh, or atanh Subroutine

Purpose

Computes inverse hyperbolic functions.

Library

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** subroutine, **acosh** subroutine, and **atanh** subroutine compute the inverse hyperbolic functions.

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

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

```
cc asinh.c -lm
```

Parameters

x	Specifies some double-precision floating-point value.

Error Codes

The **acosh** subroutine returns a NaNQ if $x < 1$.

The **atanh** subroutine returns a NaNQ if $|x| > 1$.

Implementation Specifics

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

Related Information

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

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

For Japanese Language Support, the error message is taken from the standard C library message catalog.

If you compile a program with the preprocessor option –DNDEBUG, or with the preprocessor control statement **#define** NDEBUG before the **#include <assert.h>** statement, assertions will not be compiled into the program.

Parameter

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 AIX Base Operating System (BOS) Runtime.

The **assert** macro uses the **_assert( )** library routine.

Related Information

The **abort** subroutine.

The **cpp** command.

atof, strtod, atoff, or strtof Subroutine

Purpose

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

Library

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

Syntax

#include <stdlib.h>

double atof (*NumberPointer*)
char **NumberPointer;*

double strtod (*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. 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.

These subroutines recognize a character string when the characters appear in one of the two following orders:

- An optional string of white-space characters
- An optional sign
- A non-empty string of digits optionally containing a radix character
- An optional **e** or **E** followed by an optionally signed integer.

Or ....

- An optional string of white-space characters
- An optional sign
- One of the strings: "INF", "infinity", "NaNQ", or "NaNS" (case insensitive).

Parameters

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

Error Codes

If the string is empty or begins with an unrecognized character, +0.0 is returned.

For the **strtod** or **strtof** subroutines, if the value of *EndPointer* is not:

(char**) NULL

then a pointer to the character that terminated the scan is stored in **EndPointer*. If a floating-point value cannot be formed, **EndPointer* is set to *NumberPointer*.

The **atof** (*NumberPointer*) subroutine call is equivalent to:

strtod (*NumberPointer*, (char **) NULL).

The **atoff** (*NumberPointer*) subroutine call is equivalent to:

strtof (*NumberPointer*, (char **) NULL).

If the correct return value overflows, a properly signed HUGE_VAL is returned. On underflow, a properly signed zero is returned.

Note: The **setlocale** function may affect the radix character used in the conversion.

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

If the correct value would cause overflow, +/– HUGE is returned (according to the sign of the value), and **errno** is set to ERANGE.

If the correct value would cause underflow, zero is returned and **errno** is set to ERANGE.

Implementation Specifics

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

The **atoff** and **strtof** subroutines are not part of the ANSI C Library. The accuracy of these routines is 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 nine leading zeroes. Leading zeroes are not counted as significant digits.

Related Information

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

audit Subroutine

Purpose

Enables and disables system auditing.

Library

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

Syntax

#include <sys/audit.h>
nt 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** header 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**, **AUDIT_PANIC**, and **AUDIT_NOPANIC** flags. The *Argument* parameter is ignored.

AUDIT_ON Enables auditing. If auditing is already enabled, only the failure mode behavior will change. The *Argument* parameter is used to specify recovery behavior in the presence of failure and may include one or more of the following values, defined in **sys/audit.h**:

AUDIT_PANIC The operating system will shutdown if an audit record cannot be written to a bin. Note that binmode auditing **must** be enabled prior to invoking this call if **AUDIT_PANIC** is specified.

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 for **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.

Argument Specifies the behavior when a bin write fails.

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_NOPANIC** 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 **errno** 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 includes values other than **AUDIT_PANIC**.

Implementation Specifics

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

Related Information

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

The **audit** command.

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, it may 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, may specify:

AUDIT_EXCL If the file specified by *Current* is not the kernel's current bin file, the **auditbin** subroutine fails immediately with **errno** 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, **auditbin** 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, **auditbin** fails with **errno** set to the return code from the **auditwrite** subroutine.

bin contention Another process had already issued a successful **auditbin** subroutine. If this occurs, **audtbin** fails with **errno** set to **EBUSY**.

system shutdown
The auditing system was shutdown, If this occurs, **auditibin** fails with **errno** set to **EINTR**.

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

Next Specifies the file descriptor which 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 should fail. If this value is –1, no switch will occur.

Threshold Specifies the maximum size of the current bin. If 0, the auditing subsystem will not switch bins. If it is non–zero, the kernel will begin 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 *Threshold* bytes. If no next bin is defined and **AUDIT_PANIC** was specified when the auditing subsystem was enabled, the system will be shutdown. If the size of the *Threshold* parameter was too small to contain a bin header and a bin tail, then the **auditbin** subroutine will fail and an **errno** of **EINVAL** will be set.

Return Values

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

If the **auditbin** subroutine fails, a value of– 1 returns and **errno** 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 are 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.

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.

EBUSY The *Command* parameter specifies **AUDIT_EXCL** and the kernel is not writing audit records to the file specified by *Current*.

EBUSY The *Command* parameter specifies **AUDIT_WAIT** and another process has already registered a bin.

EINTR The auditing subsystem is shutdown.

Implementation Specifics

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

Related Information

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

The **audit** command.

The **audit.h** file.

auditevents Subroutine

Purpose

Gets or sets the status of system event auditing.

Syntax

#include <sys/audit.h>
int auditevents (*Command*, *Classes*, *Nclasses*)
int *Command*;
struct audit_class **Classes*;
int *Nclasses*;

Description

The **auditevents** subroutine reads or writes the audit class definitions which control event auditing in the kernel. Each audit class is a set of one or more audit events.

System auditing need not be enabled to set or query the event lists. The **audit** subroutine can be directed to clear all event lists with the **AUDIT_RESET** command.

Parameters

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

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

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

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

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

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

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

Security

The calling process must have the **AUDIT_CONFIG** kernel privilege 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 **errno** is set to indicate the error.

Error Codes

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

EPERM The calling process does not have the **AUDIT_CONFIG** kernel privilege.

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

EINVAL The *Command* parameter is **AUDIT_SET** and the values 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 as specified by *Nclasses* 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** field of one or more **audit_class** structures passed for an **AUDIT_SET** operation points outside of the process' address space.

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

Implementation Specifications

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

Related Information

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

The **auditread** subroutine, **auditwrite** subroutine.

The **audit** command.

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 will append 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 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 **sys/audit.h** 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 non–zero values of the *Result* parameter will be converted into **AUDIT_FAIL**.

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 character.

Return Values

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

The **auditlog** subroutine does not return any indication of a failure to write the record due to the auditing subsystem configuration.

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.

Implementation Specifics

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

Related Information

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

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 reads or writes 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 per access mode. This call allows an administrator to define new audit events in the system that correspond to accesses to the specified objects. These events are not treated differently than the system–defined events.

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

Parameters

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

AUDIT_SET Sets the list of object audit events.

AUDIT_GET Queries the list of object audit events.

AUDIT_LOCK Queries the list of object audit events. This 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 **auditobj** with the *Command* parameter set to **AUDIT_SET**.

Obj_Events Specifies a buffer that contains **AUDIT_SET**, or will contain **AUDIT_GET** or **AUDIT_LOCK** as the list of object audit events. This buffer is an array of **o_event** structures. The **o_event** structure is defined in **sys/audit.h** 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 indices 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 **errno** is set to indicate the error.

Error Codes

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

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

EINVAL The *Command* parameter is **AUDIT_SET** and either the value of one or more of the **o_type** fields 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.

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** fields 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.

Implementation Specifics

This subroutine is part of AIX 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

auditpack Subroutine

Purpose

Compresses and uncompresses audit bins.

Library

Security Library (**libs.a**)

Syntax

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

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

Description

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

Parameters

Expand — Specifies the operation. Valid values, which are 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 **errno** 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 *Buffer* parameter does not point to a valid buffer.

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

ENOSPC — The function is unable to allocate space for a new buffer.

Implementation Specifics

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

Related Information

The **auditread** subroutine.

The **auditcat** command.

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;*
int *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 administrative events to be audited for this process. Administrative events 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. This is described the **auditlog** subroutine documentation.

Parameters

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

Command Specifies the action to be taken. Defined in the **audit.h** file, valid values for the are as follows:

AUDIT_QEVENTS Returns the list of audit classes defined for the current process. The *Argument* parameter is a pointer to a character buffer. The *Length* parameter is 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.

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, –1 returns and

errno is set to **EINVAL**. The *Length* and *Argument* parameters are ignored. A return value of **AUDIT_SUSPEND** indicates 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 Specifies a character pointer for the audit class buffer for an **AUDIT_EVENT** or an **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 was **AUDIT_QSTATUS** or **AUDIT_STATUS**).
- The value 0 if the call queried or set audit events (the *Command* parameter was **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 **AUDIT_QSTATUS** or **AUDIT_STATUS** value and the **pid** value is nonzero.

EINVAL The *Command* parameter is set to **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 return value is the required buffer size, in bytes.

EFAULT The *Command* parameter is **AUDIT_QEVENTS** or **AUDIT_EVENTS** and the *Argument* parameter points to a location outside of the process's allocated address space.

Implementation Specifics

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

Related Information

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

auditread Subroutine

Purpose

Reads an audit record.

Library

Security Library (**libs.a**)

Syntax

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

char *auditread (*FilePointer*, *AuditRecord***)**
**FILE ***FilePointer*;
**struct aud_rec ***AuditRecord*;

Description

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

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.

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 it is unsuccessful, a NULL pointer is returned and **errno** is set to indicate the error.

Error Codes

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

EINVAL	The **ah_magic** field in the header does not contain a valid number.
EBADF	The *FilePointer* parameter 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 AIX Base Operating System (BOS) Runtime.

Related Information

The **auditpack** subroutine.

auditwrite Subroutine

Purpose

Writes an audit record.

Library

Security Library (**libs.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 will build the tail of an audit record and then write it with the **auditlog** subroutine. The tail is built by gathering the specified buffers. The last buffer pointer must be a NULL.

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 **errno** is set to indicate the error.

Error Codes

The **auditwrite** subroutine fails if one or more of 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 AIX Base Operating System (BOS) Runtime.

Related Information

The **auditlog** subroutine.

bcopy, bcmp, bzero or ffs Subroutine

Purpose

Performs bit and byte string operations.

Library

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

Syntax

void bcopy (*Source, Destination, Length)*
char **Source*, **Destination*;
int *Length*;

int bcmp (*String1, String2, Length*)
char **String1*, **String2*;
int *Length*;

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

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

Description

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 byte string in the *String1* parameter against 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.

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

Implementation Specifics

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

Related Information

The **memcmp**, **memccpy**, **memchr**, **memcpy**, **memmove**, **memset** subroutines, **string** subroutines, **NCstring** subroutines, **NLstring** subroutines, **swab** subroutine.

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

Purpose

Computes Bessel functions.

Library

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.

Error Codes

When using **libm.a** (**–lm**):

Non-positive values cause **y0, y1**, and **yn** to return the value NaNQ.

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

Values too large in magnitude cause the functions **j0, j1**, **y0**, and **y1** to return 0 and to set **errno** to ERANGE. In addition, a message indicating TLOSS error is printed on the standard error output.

Non-positive values cause **y0, y1**, and **yn** to return the value –HUGE and to set **errno** 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 AIX Base Operating System (BOS) Runtime.

Related Information

The **matherr** subroutine.

brk or sbrk Subroutine

Purpose

Changes data segment space allocation.

Syntax

int brk (*EndDataSegment***)**
**char ***EndDataSegment*;

char *sbrk (*Increment***)**
int *Increment*;

Description

The **brk** subroutine and the **sbrk** subroutine 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** system calls.)

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 area in the process private segment. 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 global variable **errno** 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 AIX Base Operating System (BOS) Runtime.

Related Information

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

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

bsearch Subroutine

Purpose

Performs a binary search.

Library

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

Syntax

#include <stdlib.h>

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

void **Key, *Base;*
Size_t *Size, NumberOfElements;*
int (**ComparisonPointer***) (void *, 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

For the *Key* parameter: 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 AIX Base Operating System (BOS) Runtime.

Related Information

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

Donald E. Knuth's *The Art of Computer Programming*, Volume 3, 6.2.1, Algorithm B. This book was published in Reading, Massachusetts by Addison-Wesley, 1981.

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 you may reach the NL_MAXOPEN number of opened catalogs, and you must close some before opening more. Before exiting, programs should close any catalog they have opened.

The **catclose** subroutine will close a message catalog only when the number of calls to **catclose** matches the combined number of calls to **catopen** and **NLcatopen** in an application.

Parameter

CatalogDescriptor	Points to the message catalog that is returned from a call to the **catopen** or **NLcatopen** subroutine.

Return Values

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

Error Codes

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

Implementation Specifics

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

Related Information

The **catopen, NLcatopen** subroutine.

catgetmsg Subroutine

Purpose

Copies a message from a catalog into a user-defined character string buffer.

Library

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

Syntax

#include <nl_types>

char *catgetmsg (*CatalogDescriptor, SetNumber, MessageNumber, Buffer, BufferLength***)**
nl_catd *CatalogDescriptor;*
int *SetNumber, MessageNumber, BufferLength;*
char **Buffer;*

Description

The **catgetmsg** subroutine retrieves a message from a catalog after a successful call to the **catopen** subroutine. As with the **catgets** subroutine, you specify a catalog with the *CatalogDescriptor* parameter returned by the **catopen** subroutine.

If the message is found, the **catgetmsg** subroutine returns the *Buffer* pointer that points to the message.

The **catgetmsg** subroutine copies up to *BufferLength*–1 bytes of the message into the buffer specified by the *Buffer* parameter. The **catgetmsg** subroutine does not split a 2-byte character (an extended character).

Parameters

CatalogDescriptor	Specifies a catalog description that is returned by the **catopen** subroutine.
SetNumber	Specifies the set ID.
MessageNumber	Specifies the message ID. *SetNumber* and *MessageNumber* specify a particular message in the catalog to retrieve.
Buffer	Points to the buffer in which the retrieved message is placed.
BufferLength	Specifies the length of the buffer.

Error Codes

If the **catgetmsg** subroutine fails, the *Buffer* parameter points to an empty string.

Implementation Specifics

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

The **catgetmsg** subroutine has been withdrawn from X/Open.

Related Information

The **catgets** subroutine, **NLcatgets** subroutine, **NLgetamsg** subroutine.

catgets Subroutine

Purpose

Retrieves a message from a catalog.

Library

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

Syntax

#include <nl_types>

char *catgets (*CatalogDescriptor, SetNumber, MessageNumber, String***)**
nl_catd *CatalogDescriptor;*
int *SetNumber, MessageNumber;*
char **String;*

Description

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

The pointer is used to reference the buffer and display the message; use the **printf** or **NLprintf** subroutine with either the %s or %n$s conversion specification. The message in the buffer is overwritten by the next call to the **catgets** subroutine.

The **catgets** and **catgetmsg** subroutines retrieve messages from an open catalog. The AIX operating system includes two functions for getting messages that are not defined by X/Open: the **NLcatgets** and the **NLgetamsg** subroutines.

Parameters

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

Error Codes

If the **catgets** subroutine fails for any reason, it returns the user-supplied default message string, *String*.

Implementation Specifics

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

Related Information

The **catgetmsg** subroutine, **NLcatgets** subroutine, **NLgetamsg** subroutine.

catopen or NLcatopen Subroutine

Purpose

Opens a specified message catalog.

Library

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

Syntax

 # include <limits.h>
 # include <nl_types.h>

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

nl_catd NLcatopen (*CatalogName, Parameter*)
char **CatalogName;*
int *Parameter;*

Description

The **catopen** subroutine opens a specified message catalog and returns a catalog descriptor that you use to retrieve messages from the catalog.

The **NLcatopen** subroutine prepares a catalog to be opened. To avoid unnecessary opening of files, **NLcatopen** does not actually open the catalog until a message is needed.

The special **nl_catd** data type is used for catalog descriptors. Since this data type is defined in the **nl_types.h** header file, include this file in your application program.

If the catalog file name referred to by the *CatalogName* parameter begins with a /, it is assumed to be an absolute path name. If the catalog file name is not an absolute path name, the user environment determines the directory paths to search.

The environment variable NLSPATH defines the directory search path. You can use two special variables, %N and %L, in the environment variable NLSPATH.

The variable %N will be replaced by the catalog name referred to by the call that opens the message catalog. The variable %L will be replaced by the value of the LANG environment variable.

You can use the LANG environment variable to refer to message catalogs that are separated into directories based on natural languages. 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 variable %N in a directory specification within the environment variable NLSPATH, the application assumes that the path defines a directory and searches for the catalog in that directory before searching the next specified path.

Parameters

CatalogName	Specifies the catalog file to open.
Parameter	Included for compatibility with X/Open, but is not used by the AIX operating system. Takes the value of 0.

Error Codes

The **catopen** and **NLcatopen** subroutines return a value of –1 if they cannot find the file or if the number of catalogs already open is equal to the NL_MAXOPEN limit defined in the **mesg.h** header file.

Implementation Specifics

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

Related Information

The **catclose** subroutine.

cfgetospeed, cfsetospeed, cfgetispeed, or cfsetispeed Subroutine

Purpose

Get and set input and output baud rates.

Library

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

Syntax

#include <termios.h>

speed_t cfgetospeed (*TermiosPointer***)**
**struct termios ****TermiosPointer*;

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

speed_t cfgetispeed (*TermiosPointer***)**
**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 values shown below are supported. The name symbols in this table are defined in the **termios.h** file.

The type **speed_t** is defined in the **termios.h** file 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 are discussed in the **tcsetattr** subroutine.

Baud Rate Values

Name	Description	Name	Description
B0	Hang up	B600	600 baud
B50	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

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 zero if successful and –1 to indicate an error.

Example

To set the output baud rate to zero to force modem control lines to no longer be asserted, enter:

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

Implementation Specifics

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

Related Information

The **tcsetattr** subroutine.

The **termios.h** header file.

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 structure pointed to by the *ACL* parameter. This structure is defined in the **sys/acl.h** file and contains the following members:

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

acl_mode The file mode.

u_access The access permissions for the file owner.

g_access The access permissions for the file group.

o_access The access permissions for the default class *others*.

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

The following bits in the **acl_mode** field 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 are ignored.

The fields for the base ACL – owner, group, and others – may 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.

Parameters

Path Specifies the path name of the file.

FileDescriptor
Specifies the file descriptor of an open file.

ACL Specifies the Access Control List to be established on the file. The format of an ACL is defined in the **sys/acl.h** header file.

ACLSize Specifies the size of the buffer containing the ACL.

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 global variable **errno** 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** system call).

ENOENT The *Path* parameter was null.

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

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

ESTALE The process's 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 **fclacl** subroutine fails and the access control information for a file remains unchanged if one or more of the following are true:

EROFS The named file 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 Access Control List.

EINVAL The **ACL_Len** field in the ACL is not valid.

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.

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

EBADF The file descriptor *FileDescriptor* is not valid.

If 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.

Auditable Events

Event Name	***Tail Information***
chacl	*Path*
fchacl	*FileDescriptor*

Implementation Specifics

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

Related Information

The **statacl** subroutine, **chmod** subroutine, **stat** subroutine.

The **acl_get** subroutine, **acl_put** subroutine, **acl_set** subroutine, **acl_chg** subroutine.

The **acl_get** command, **acl_put** command.

chdir Subroutine

Purpose

Changes the current directory.

Library

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

Syntax

int chdir (*Path*)
char **Path*;

Description

The **chdir** subroutine changes the current directory to the directory indicated by the *Path* parameter.

Parameter

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 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 global variable **errno** 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 fail if additional errors on page A–1 occur.

If Network File System is installed on the system, the **chdir** system call can also fail if the following is true:

ETIMEDOUT	The connection timed out.

Implementation Specifics

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

Related Information

The **chroot** subroutine.

The **cd** command.

chmod or fchmod Subroutine

Purpose

Changes file access permissions.

Library

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

Syntax

#include <sys/stat.h>

int chmod (*Path*, *Mode***)**
**char ****Path*;
int *Mode*;

int fchmod (*FileDescriptor*, *Mode***)**
**char ****FileDescriptor*;
int *Mode*;

Description

The **chmod** subroutine sets the access permissions of the file specified by the *Path* parameter. If Network File System 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. The use of these subroutines will implicitly disable extended ACL entries and is therefore discouraged.

Parameters

FileDescriptor
Specifies the file descriptor of an open file.

Mode
Specifies the bit pattern which 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** header 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.
Enables the group inheritance attribute for a directory. Files created in this directory will have a group equal to the group of the directory.

S_ISVTX
Enables the **link/unlink** attribute for a directory. Files may not be linked to in this directory and files may only be unlinked if the requesting process has write permission for the directory and is either the owner of the file or the owner of the directory.

S_ISVTX	Enables the **link/unlink** attribute for a direcsave 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** subroutine return a value of 0. If the **chmod** subroutine or **fchmod** subroutine fails, a value of –1 is returned, and the global variable **errno** is set to identify the error.

Error Codes

The **chmod** subroutine fails and the file permissions remain unchanged if one or more of the following are true:

ENOTDIR	A component of the *Path* prefix is not a directory.
EACCESS	Search permission is denied on a component of the *Path* prefix.
EFAULT	The *Path* parameter points to a location outside of the allocated address space of the process.
ESTALE	The process's 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.

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

EBADF The file descriptor *FileDescriptor* is not valid.

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

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

EIO An I/O error occurred during the operation.

EBUSY The value of the *Mode* parameter would change the enforced lov=cking attribute of an open file.

If 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.

Security

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

Implementation Specifics

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

Related Information

The **chacl** subroutine, **statacl** subroutine, **stat** subroutine.

The **acl_get** subroutine, **acl_put** subroutine, **acl_set** subroutine, **acl_chg** subroutine.

The **acl_get** command, **acl_put** command, **chmod** command.

chown, fchown, chownx, or fchownx Subroutine

Purpose

Changes file ownership.

Syntax

#include <sys.chownx.h>
int chown (*Path*, *Owner*, *Group*)
char **Path*;
uid_t *Owner*;
gid_t *Group*;

int fchown (*FileDescriptor*, *Owner*, *Group*)
int *FileDescriptor*;
uid_t *Owner*;
gid_t *Group*;

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**, and **fchownx** 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.

The new owner or group will inherit the access control permissions in the base Access Control List. All other permissions are unchanged by this function.

Parameters

FileDescriptor — Specifies the file descriptor of an open file.

Flags — Specifies whether each of the file owner ID and group ID is to be changed. This parameter is constructed by logically ORing the following values:

- **T_OWNER_AS_IS** — Ignores the value specified in the *Owner* parameter and leaves the owner ID of the file unaltered.
- **T_GROUP_AS_IS** — Ignores the value specified in 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 will not be changed.

Owner — Specifies the new owner of the file. If this value is –1, the owner will not be changed.

Path — Specifies the full path name of the file. If *Path* resolves to a symbolic link, the ownership of the symbolic link is changed.

Return Values

Upon successful completion, the **chown**, **chownx**, **fchown**, and **fchownx** subroutines return a value of 0. If the **chown**, **chownx**, **fchown**, or **fchownx** subroutines fail, a value of –1 is returned and **errno** is set to indicate the error.

Error Codes

The **chown** or **chownx** subroutines fail and the owner and group of a file remain unchanged if one or the following are true:

ENOTDIR — A component of the path prefix is not a directory.

EACCESS — Search permission is denied on a component of the *Path* parameter.

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

ESTALE — The process's 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.

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

ENOENT — The *Path* parameter was null.

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

The **fchown** or **fchownx** subroutines fail and the file owner and group remain unchanged if the following is true:

EBADF — The named file resides on a read–only 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.

Auditing Events

Event	Information
FILE_SetOwner	object descriptor, owner, group

Implementation Specifics

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

Related Information

The **chmod** subroutine.

chroot Subroutine

Purpose

Changes the effective root directory.

Library

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

Syntax

int chroot (*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 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 .. (dot dot) entry in the effective root directory is interpreted to mean the effective root directory itself. Thus, .. (dot dot) cannot be used to access files outside the subtree rooted at the effective root directory.

Parameter

Path	A 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 global variable **errno** 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 also fail if additional errors on page A–1 occur.

If Network File System 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 AIX Base Operating System (BOS) Runtime.

Related Information

The **chdir** subroutine.

The **chroot** command.

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 <sys/spc.h>

int chssys(*SubsystemName,SRCSubsystem***)**
char ****SubsystemName*;**
struct SRCsubsys ****SRCSubsystem*;**

Description

The **chssys** subroutine modifies the subsystem objects associated with *SubsystemName* with the values in the *SRCsubsystem* parameter. This will modify the objects associated with subsystem in the following object classes: Subsystem object, Subserver object, Notify object. The Subserver and Notify object classes will only be updated if the subsystem name has been changed.

The **SRCsubsys** structure is defined in the **sys/srcobj.h** header file.

The executable 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 **odmerrno** is set to indicate the error or an 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.

Implementation Specifics

This subroutine is part of AIX 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.

Related Information

The **addssys** subroutine, **delssys** subroutine.

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

The System Resource Controller Overview in *General Programming Concepts*.

ckuserID Subroutine

Purpose

Authenticates the user

Library

Security Library (**libs.a**)

Syntax

#include<login.h>
int ckuserID(*User, Mode)*
int *Mode*;
char **User;*

Description

The **ckuserID** function will authenticate the account specified by the *User* parameter. The mode of the authentication is given by the *Mode* parameter.

Parameters

User Specifies the name of the user to 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 need not be done successfully.

Primary and secondary authentication methods are set for each user in **/etc/security/user** by defining the **AUTH1** and **AUTH2** attributes. If no primary methods are defined for a user, **SYSTEM** is assumed. If no secondary methods are defined, there is no default.

Security

file access The calling process must have access to the account information in the user data base and the authentication data. These include:

modes	**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 **errno** 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 not one or more of **S_PRIMARY** or **S_SECONDARY**.

Implementation Specifics

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

Related Information

The **ckuseracct** subroutine, **getpcred** subroutine, **setpcred** subroutine, **getpenv** subroutine, **setpenv** subroutine.

The **login** command and **su** command.

ckuseracct Subroutine

Purpose

Checks the validity of the user account

Library

Security Library (**libs.a**)

Syntax

#include <usersec.h>

int ckuseracct(*Name*, *Mode*, *Tty***)**
char **Name;*
int *Mode*;
char **Tty*;

Description

The **ckuseracct** subroutine will check the validity of the account of the user specified by the *Name* parameter. The mode of the account usage is given by the *Mode* parameter, while the *Tty* parameter defines the terminal being used for the access.

The **ckuseracct** subroutine will check for the following conditions:

- account existence
- account expiration

Other *Mode* specific checks are made as described in the *Mode* parameter.

- **S__LOGIN**
- **S__RLOGIN**
- **S__SU**
- **S_DAEMON**

Parameters

Name Specifies the login name of the user whose account is to be validated.

Mode Specifies the manner of usage. Valid values are defined in the **usersec.h** file and are listed below. The *Mode* parameter must be one of these or zero.

S_LOGIN	Verifies the 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 which can invoke the **su** command to switch to the account.
S_DAEMON	Verifies the account can be used to invoke daemon or batch programs via the **src** or **cron** subsystems.
S_RLOGIN	Verifies the account can be used for remote logins via 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

File Access The calling process must have access to the account information in the user data base. This includes:

modes	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 **errno** 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 user 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 AIX Base Operating System (BOS) Runtime.

Related Information

The **ckuserID** subroutine, **getpcred** subroutine, **setpcred** subroutine, **getpenv** subroutine, **setpenv** subroutine.

The **login** command, **cron** command, **rlogin** command, **telnet** command, **su** command.

class, finite, isnan, or unordered Subroutines

Purpose

Determines classifications of floating-point numbers.

Library

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

Syntax

#include<math.h>
#include<float.h>

int class(x)
double *x*;

int finite(*x***)**
double *x*;

int isnan(*x)*
double *x*;

int unordered(*x, y)*
double *x, y*;

Description

The **class** subroutine, **finite** subroutine, **isnan** subroutine, and **unordered** subroutin 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. 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)

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 zero.

The **unordered** subroutine returns a nonzero value if a floating-point comparison between *x* and *y* would be unordered. Otherwise, it returns zero.

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 AIX Base Operating System (BOS) Runtime.

Related Information

IEEE Standard for Binary Floating-Point Arithmetic (ANSI/IEEE Standards 754-1985 and 854-1987)

clock Subroutine

Purpose

Reports CPU time used.

Library

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

Syntax

#include <time.h>

clock_t clock ( );

Description

The **clock** subroutine reports the amount of CPU time used (in microseconds). 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.

Return Value

The **clock** subroutine returns the amount of CPU time used since the first call to the **clock** subroutine.

Implementation Specifics

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

Related Information

The **getrusage, times** subroutine, **wait, waitpid, wait3** subroutine.

The **system** subroutine, **pclose** subroutine, **vtimes** subroutine.

close Subroutine

Purpose

Closes the file associated with a file descriptor.

Syntax

close (*FileDescriptor*)
int *FileDescriptor*;

Description

The **close** subroutine closes the file associated with the *FileDescriptor* parameter. If Network File System 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.

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 *FileDescriptor* 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* 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 **exec** if the close–on–exec flag has been set for that file descriptor.

Parameter

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 global variable **errno** is set to identify the error.

Error Codes

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

EBADF	The *FileDescriptor* parameter does not specify a valid open file descriptor.

If Network File System is installed on the system, the **close** subroutine can also fail if the following is true:

ETIMEDOUT	The connection timed out.

Implementation Specifics

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

Related Information

The **exec** subroutines, **fcntl** subroutine, **ioctl** subroutine, **lockfx** subroutine, **open**, **openx**, **creat** subroutines, **pipe** subroutine, **socket** subroutine.

compile, step, or advance Subroutine

Purpose

Compiles and matches regular–expression patterns.

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 **InString*, **Expbuffer*, **Endbuffer*;
char *EndOfFile*;

int step (*String*, *Expbuffer***)**
char **String*, **Expbuffer*;

int advance (*String*, *Expbuffer***)**
char **String*, **Expbuffer*;

Description

The **regexp.h** header file defines several general purpose subroutines that perform regular–expression pattern matching. Programs that perform regular–expression pattern matching such as **ed**, **sed**, **grep**, **bs**, and **expr** use this source file. In this way, only this file needs to be changed in order to maintain regular expression compatibility between programs.

The **NLregexp.h** header file handles extended characters and requires access to the locale information for collation and character class determination. **NLregexp.h** accepts character classes as described in **ed**.

The interface to these header files is complex. Programs that include this file define the following six macros before the **#include <regexp.h>** or the **#include <NLregexp.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 **INIT** 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 **GETC**, **PEEKC**, and **UNGETC**. Otherwise, you can use **INIT** to declare external variables that **GETC**, **PEEKC**, and **UNGETC** need.

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 on 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 on 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 \{ \}.
48	Invalid end point in range expression.
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 instance, you may want 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** below 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 **ed** this character is usually / (slash).

The **regexp.h** and **NLregexp.h** header files define other subroutines that perform actual regular–expression pattern matching. One of these is the **step** subroutine.

The *String* parameter of **step** 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, which was 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 **compile** if the regular expression begins with a ^ (circumflex). If this variable is set, then **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, then 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 **advance** until **advance** 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 **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 \{ \}. It is sometimes desirable to 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, **advance** breaks out of the loop that backs up and returns **0**. This is used by **ed** and **sed** for global substitutions on the whole line so that expressions like **s/y*//g** do not loop forever.

Parameters

Instring	String containing the pattern to be compiled. The *Instring* parameter is not used explicitly by the **compile** subroutine, but may be used in macros.
Expbuffer	Pointer to a character array where the compiled regular expression is to be placed.

Endbuffer Pointer to the location that immediately follows the character array where the compiled regular expression is to be placed.

EndOfFile Character that marks the end of the regular expression.

String Pointer to a null–terminated string of characters to be checked for a match.

Example

The following is an example of the regular expression macros and calls from the **grep** command.

```
#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 AIX Base Operating System (BOS) Runtime.

Related Information

The **NCctype** subroutine, and **regcmp, regex** subroutines.

The **ed** command, **sed** command, and, **grep** command.

National Language Support Overview in *General Programming Concepts*

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 **NC***xxxxxx* subroutines translate all characters, including extended characters, as code points. The other subroutines translate traditional ASCII characters only.

The **toupper** and the **tolower** subroutines have as domain the range of the **getc** subroutine: –1 through 255.

If the parameter of the **toupper** subroutine represents a lowercase letter, the result is the corresponding uppercase letter. If the parameter of the **tolower** subroutine represents an uppercase letter, the result is the corresponding lowercase letter. All other values in the domain are returned unchanged.

The **_toupper** and **_tolower** routines are macros that accomplish the same thing as the **toupper** and **tolower** subroutines, but they have restricted domains and are faster. 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 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 **NLtolower** 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 (?).

(The **NCflatchr** subroutine is not supported when running AIX with Japanese Language Support.)

Note: The **setlocale** subroutine may affect the conversion of the decimal point symbol and the thousands separator.

Parameters

Character	The character to be converted.
Xcharacter	An NLchar value to be converted.
CharacterPointer	A pointer to an ASCII character array.
Pointer	A pointer to an escape sequence.

Implementation Specifics

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

Related Information

The **ctype** subroutines, **Japanese conv** subroutines, **getc**, **fgetc**, **getchar**, **getw** subroutines, **getwc**, **fgetwc**, **getwchar** subroutines, **setlocale** subroutine.

copysign, nextafter, scalb, logb, or ilogb Subroutine

Purpose

Computes certain binary floating-point arithmetic functions.

Library

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*, *n*)
double *x;*
int *n;*

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 *y*.

The **nextafter** subroutine returns the next representable neighbor of *x* in the direction of *y*. If *x* = *y*, the result is *x*.

The **scalb** subroutine returns *x* times 2**n.

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 nonzero, then **logb** (*x)* 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 *x*. Special cases are:

ilogb (NaN) = **LONG_MIN**
ilogb (INF) = **LONG_MAX**
ilogb (0) = **LONG_MIN**

Note: **ilogb** (*x*) is equivalent to (**int**) **logb** (*x*). However, **ilogb** may be faster on some platforms of IBM AIX Version 3 for RISC System/6000.

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

```
cc copysign.c –lm
```

Parameters

x Specifies some double-precision floating-point value.

y Specifies some double-precision floating-point value.

n Specifies some integer value.

Return Values

The **nextafter** subroutine sets the overflow bit in the floating-point exception status when *x* is finite but **nextafter** (*x, y*) is infinite. Likewise, when the **nextafter** subroutine is denormalized, the underflow exception status flag is set.

The **logb**(0) subroutine returns –INF and sets the division-by-zero exception status flag.

The **ilogb**(0) subroutine returns LONG_MIN and sets the division-by-zero exception status flag.

Implementation Specifics

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

Related Information

The **class, finite, isnan, unordered** subroutines, **fp_invalid_op, fp_divbyzero, fp_overflow, fp_underflow, fp_inexact, fp_any_xcp, fp_iop_snan, fp_iop_infsinf, fp_iop_infdinf, fp_iop_zrdzr, fp_iop_infmzr, fp_iop_invcmp** subroutines.

The IEEE Standard for Binary Floating-Point Arithmetic (ANSI/IEEE Standards 754–1985 and 854–1987).

crypt, encrypt or setkey Subroutine

Purpose

Performs basic encryption of data.

Library

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

Syntax

char *crypt (*Key*, *Salt*)
char **Key*, **Salt*;

void encrypt (*Block*, *Edflag*)
char **Block*;
int *Edflag*;

void setkey (*Key*)
char **Key;*

Description

The **crypt** and **encrypt** subroutines provide encryption of data. The **crypt** subroutine performs a one way encryption of a fixed data array with the supplied *Key* parameter, using the *Salt* parameter to perturb the encryption algorithm. The **encrypt** subroutine encrypts or decrypts the data supplied in the *Block* parameter by 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, with each character having the value of ASCII "0" or ASCII "1".

Parameters

Block — A 64–character array containing the values (char) 0 and (char) 1. Upon return, this buffer will contain the encrypted or decrypted data.

Edflag — If this parameter is zero, the argument is encrypted; if non–zero, it is decrypted.

Key — Specifies an 8 character string which is used to change the encryption algorithm.

Salt — Specifies a 2 character string chosen from the set **["a–zA–Z0–9./"]**. The *Salt* parameter is used to vary the hashing algorithm in one of 4096 different ways.

Compatibility Interface

These functions are provided for compatibility with UNIX system implementations.

Return Values

The **crypt** subroutine returns a pointer to the encrypted password. The first two characters of it are the same as the *Salt* parameter.

Note: The return value points to static data that is overwritten by subsequent calls.

Implementation Specifics

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

Related Information

The **newpass** subroutine.

The **login** command, **passwd** command, **su** command.

cs Subroutine

Purpose

Compare and swap data.

Syntax

int cs (*Destination*, *Compare*, *Value*)
int **Destination;*
int *Compare;*
int *Value;*

Description

The **cs** command compares *Compare* with the integer pointed to by *Destination*. If they are equal, *Value* is stored in the integer pointed by *Destination* and **cs** returns 0. If the values are different, **cs** returns 1, and the value pointed by *Destination* is not affected. The compare and store are executed as an atomic operation, 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 examples shows how a new element can be inserted in a NULL terminated list stored in shared memory and maintained by several processes, with the following code:

```
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 that will be compared with *Compare*, and if need be, where *Value* will be stored.

Compare Specifies the value that will be compared with the integer pointed by *Destination.*

Value Specifies the value that will be stored in the integer pointed by *Destination* if *Destination* and *Compare* are equal.

Error Codes

If the integer pointed by *Destination* references memory that does not belong to the process address space then the SIGSEGV signal is sent to the process.

Implementation Specifics

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

Related Information

The **sigaction, sigvec, signal** subroutine, **shmget** subroutine, **shmat** subroutine, **shmdt** subroutine, **shmctl** subroutine.

ctermid Subroutine

Purpose

Generates the path name for 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.

The difference between the **ctermid** subroutine and the **ttyname** subroutine is that the **ttyname** subroutine must be handed a file descriptor and returns the actual name of the terminal associated with that file descriptor, while the **ctermid** subroutine returns a string (/**dev/tty**) 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.

Parameter

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** header 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 AIX Base Operating System (BOS) Runtime.

Related Information

The **ttyname, isatty** subroutines.

ctime, localtime, gmtime, mktime, difftime, asctime, tzset, or timezone Subroutine

Purpose

Converts the formats of date and time representations.

Library

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

Syntax

#include <time.h>
char *ctime (*Clock***)**
time_t ****Clock*;**

struct tm *localtime (*Clock***)**
time_t ****Clock*;**

struct tm *gmtime (*Clock***)**
time_t ****Clock*;**

time_t mktime(*Timeptr***)**
struct tm ****Timeptr*;**

double *difftime(*Time1***,** *Time0***)**
time_t *Time0***,** *Time1***;**

char *asctime (*Tm***)**
struct tm ****Tm*;**

void tzset ( )

extern long timezone;
extern int daylight;
extern char *tzname[2];

char *timezone(*Zone***,** *Destination***)**
int *Zone***,** *Destination***;**

Description

The **ctime** subroutine converts a time value pointed to by the *Clock* parameter, which represents the time in seconds since 00:00:00 Greenwich Mean Time (GMT), January 1, 1970, into a 26–character string in the following form:

```
Sun Sep 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 timezone and daylight savings 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 GMT, January 1, 1970, into a **tm** structure. The **localtime** subroutine adjusts for the time zone and for daylight–saving time, if it is in effect.

The **gmtime** subroutine converts the long integer pointed to by the *Clock* parameter into a **tm** structure containing the Greenwich Mean Time, which is the time that AIX uses.

The **tm** structure is defined in the **time.h** header 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 **gmtime** subroutine.

The **difftime** subroutine computes the difference between two calendar times: the *Time1 –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**. **tzset** 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 GMT and local standard time. For example, **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. See environ. for information about specifying alternate daylight–saving time conventions.

The **tzname** external variable contains the name of the standard time zone (**tzname[0]**) and of the time zone when daylight–saving time is in effect (**tzname[1]**). For example:

```
char *tzname[2] = {"EST", "EDT"};
```

The **timezone** subroutine returns the name of the timezone associated with the first argument, which is measured in minutes westward of Greenwich. If the second argument is 0, the standard name is used, otherwise the Daylight Saving version. If the required name does not appear in an internal table, the difference from GMT is produced; e.g. in Afganistan timezone [–(60 * 4 + 30), 0] is appropriate because it is 4:30 ahead of GMT and the string GMT+4:30 is produced.

The **time.h** header file contains declarations of all these subroutines, externals, and the **tm** structure.

Warning: The return values point to static data that is overwritten by each call.

Parameters

Clock	Pointer to the time value in seconds.
Timeptr	Pointer to a **tm** structure.
Time1	Pointer to a **time_t** structure.
Time0	Pointer to a **time_t** structure.
Tm	Pointer to a **tm** structure.
Zone	The minutes westward of Greenwich Mean Time.
Destination	Standard Time, if 0, otherwise Daylight Savings Time

Implementation Specifics

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

The **timezone** subroutine was added for BSD compatibility, and is not part of the ANSI C Library.

Related Information

The **getenv**, **NLgetenv** subroutines, **NLstrtime, strftime** subroutines, and **NLtmtime** subroutine, **gettimer** subroutine.

The **gettimer** subroutine.

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**.

The following list shows the set of values for which each subroutine returns a nonzero (**TRUE**) value:

isalnum	*Character* is a letter or a digit.
isalpha	*Character* is a letter.
isupper	*Character* is an uppercase letter.
islower	*Character* is a lowercase letter.
isdigit	*Character* is a digit in the range [0–9].
isxdigit	*Character* is a hexadecimal digit in the range [0–9], [A–F], or [a–f].
isspace	*Character* is a space, tab, carriage return, new–line, vertical tab, or form feed character.
ispunct	*Character* is a punctuation character (neither a control character nor an alphanumeric character).
isprint	*Character* is a printing character: alphanumeric, punctuation, or space.
isgraph	*Character* is a printing character, like **isprint**, but, unlike **isprint**, **isgraph** returns **FALSE** (0) for the space character.
iscntrl	*Character* is an ASCII delete character (0177 or 0x7F), or an ordinary control character (less than 040 or 0x20).
isascii	*Character* is an ASCII character whose value is in the range 0–0177 (0–0x7F), inclusive.

Parameter

Character	Character to be tested (integer value).

Implementation Specifics

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

Related Information

The **NCctype** subroutines, **Japanese ctype** subroutines, **setlocale** subroutine.

National Language Support Overview in *General Programming Concepts*.

cuserid Subroutine

Purpose

Gets the alphanumeric user name associated with the current process.

Library

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

Syntax

#include <stdio.h>

char *cuserid (*String***)**
**char ****String*;

Description

The **cuserid** subroutine generates a character string representing the user name of the owner of the current process.

Parameter

String — If the *String* parameter is a **NULL** pointer, the character string is stored into an internal static area, the address of which is returned.

If the *String* parameter is not a **NULL** pointer, the character string is stored into the array pointed to by the *String* parameter. This array must contain at least **L_cuserid** characters. **L_cuserid** is a constant defined in the **stdio.h** header file.

If the user name cannot be found, the **cuserid** subroutine returns a **NULL** pointer; if the *String* parameter is not a **NULL** pointer, a null character ('\0') is stored into *String*[0].

Implementation Specifics

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

Related Information

The **getlogin** subroutine, **getpwent, getpwuid, getpwnam, setpwent, endpwent, setpufile** subroutines.

defssys Subroutine

Purpose

Initializes the **SRCsubsys** structure with default values.

Library

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

Syntax

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

void defssys(*SRCSubsystem***)**
struct SRCsubsys **SRCSubsystem*;

Description

The **defssys** subroutine initializes the **SRCsubsys** structure with the following default values:

Field	Value
display	**SRCYES**
multiple	**SRCNO**
contact	**SRCSOCKET**
waittime	**TIMELIMIT**
priority	20
restart	**ONCE**
stderr	**/dev/console**
stdin	**/dev/console**
stdout	**/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.

Parameter

SRCSubsystem Points to the **SRCsubsys** structure, which is defined in the **sys/srcobj.h** header file.

Implementation Specifics

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

Related Information

The **addssys** subroutine.

The System Resource Controller Overview in *General Programming Concepts*.

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 <srcerrno.h>

int delssys(*SubsystemName***)**
**char ****SubsystemName*;

Description

The **delssys** subroutine removes the subsystem objects associated with the *SubsystemName* parameter. This removes all objects associated with the subsystem from the following object classes: Subsystem object, Subserver object, Notify object.

The executable 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 **odmerrno** is set to indicate the error.

Implementation Specifics

This subroutine is part of AIX 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.

Related Information

The **addssys** subroutine, **chssys** subroutine.

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

The System Resource Controller Overview in *General Programming Concepts*.

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 that has content that is no longer needed. This allows the system to stop 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 is to be set to a value of 0.

Return Values

Upon successful completion, 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 global variable **errno** to indicate the error. The **disclaim** subroutine if 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.

Implementation Specifics

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

Related Information

The **shmat** subroutine, **shmctl** subroutine, **shmdt** subroutine, **shmget** subroutine.

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

double drand48 ( )

double erand48 (*xsubi*)
unsigned short *xsubi*[3];

long jrand48 (*xsubi*)
unsigned short *xsubi*[3];

void lcong48 (*Parameter*)
unsigned short *Parameter*[7];

long lrand48 ( )

long mrand48 ( )

long nrand48 (*xsubi*)
unsigned short *xsubi*[3];

unsigned short *seed48 (*Seed16v*)
unsigned short *Seed16v*[3];

void srand48 (*SeedValue*)
long *SeedValue*;

Description

This family of subroutines generates pseudo-random numbers using the linear congruential algorithm and 48-bit integer arithmetic.

The **drand48** subroutine and **erand48** subroutine return non-negative double-precision floating-point values uniformly distributed over the range of *y* values such that $0 < y < 1$.

The **lrand48** subroutine and **nrand48** subroutine return non-negative long integers uniformly distributed over the range of *y* values such that $0 < y < 2^{**}31$.

The **mrand48** subroutine and **jrand48** subroutine return signed long integers uniformly distributed over the range of *y* values such that $-2^{**}31 < y < 2^{**}31$.

The **srand48** subroutine, **seed48** subroutine, and **lcong48** subroutine initialize the random-number generator. Programs should call one of them before calling the **drand48**, **lrand48** or **mrand48** subroutines. (Although it is not recommended practice, constant default initializer values are supplied automatically 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 to be called first.

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; that 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. That is why these routines do not have to be initialized; the calling program merely has to place 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 several 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* as specified previously.

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. Later you can call **seed48** with a pointer to this array to resume where the original sequence left off.

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 AIX Base Operating System (BOS) Runtime.

Related Information

The **rand, srand** subroutine, **random, srandom, initstate, setstate** subroutine.

drem Subroutine

Purpose

Computes the IEEE Remainder as defined in the IEEE Floating-Point Standard.

Library

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

Syntax

#include <math.h>

double drem (*x, y)*
double *x, y*;

Description

The **drem** subroutine calculates the remainder $r = x - n \times y$, where n is the integer nearest the exact value of `x/y`; moreover if `|n -x/y| = 1/2`, then n is an even value. Therefore, the remainder is computed exactly, and `|r|` is less than or equal to `|y|/2`.

The IEEE Remainder differs from FMOD in that the IEEE Remainder always returns an *r* 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. To compile the drem.c file, for example:

```
cc drem.c -lm
```

Parameters

x Some double-precision floating-point value.

y Some double-precision floating-point value.

Return Values

The **drem** subroutine returns a NaNQ for `(x, 0)` and `(+INF, y)`.

Implementation Specifics

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

Related Information

The **floor, ceil, nearest, trunc, rint, itrunc, uitrunc, fmod, fabs** subroutines, **copysign, nextafter, scalb, logb, ilogb** subroutines.

IEEE Standard for Binary Floating-Point Arithmetic (ANSI/IEEE Standards 754-1985 and 854-1987) describes the IEEE Remainder Function.

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** subroutine, **fcvt** subroutine, and **gcvt** subroutine 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 it to 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 *NumberOfDigits*.

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 *Buffer*. The **gcvt** subroutine attempts to produce a string of *NumberOfDigits* 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 **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	NaNQ
Signalling NaN	NaNS
Infinity	INF

The sign associated with each of these values is stored into the *Sign* parameter. Note also that 0 can be positive or negative. In the IEEE floating-point, zeros also have signs and set the *Sign* parameter appropriately.

Warning: 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	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 AIX Base Operating System (BOS) Runtime.

Related Information

The **atof, atoff, strtod, strtof** subroutines, **scanf** subroutine, **printf** subroutine, **fp_read_rnd, fp_swap_rnd** subroutines.

IEEE Standard for Binary Floating-Point Arithmetic (ANSI/IEEE Standards 754-1985 and 854-1987).

_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 input/output subroutines
- The **–p** flag on the **cc** command.

Therefore, use **brk** or **sbrk(0)**, not **end**, to determine the break value of the program.

Implementation Specifics

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

Related Information

The **malloc** subroutine, **brk, sbrk** subroutine.

erf or erfc Subroutine

Purpose

Computes the error and complementary error functions.

Library

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

Syntax

#include <math.h>

double erf (*x*)
double *x*;

double erfc (*x*)
double *x*;

Description

The **erf** subroutine returns the error function of the *x* parameter, defined 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** subroutine is 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
```

Parameter

x Specifies some double-precision floating-point value.

Implementation Specifics

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

Related Information

The **exp, expm1, log, log10, log1p, pow** subroutines, **sqrt, cbrt** subroutines.

errlog Subroutine

Purpose

Logs application errors.

Library

Run-time Services Library.

Syntax

#include <sys/errids.h>
int errlog (*Buffer*, *Cnt***)**
**char ****Buffer*;
unsigned int *Cnt*;

Description

The **errlog** subroutine writes an error to the error log device driver. This subroutine is used by application programs.

Parameters

Buffer	Points to a buffer that contains an error record.
Cnt	Specifies the size in bytes of the error record in the buffer.

Return Values

0	Successful.
–1	Error message if unsuccessful.

Implementation Specifics

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

Files

/dev/error	Provides standard device driver interfaces required by the error log component.
librts.a	Run-time Services Library.

Related Information

The **errdemon** daemon.

The **errclear** command, **errdead** command, **errinstall** command, **errlogger** command, **errmsg** command, **errpt** command, **errstop** command, **errupdate** command.

The **error** file.

The **errsave** kernel service.

exec: execl, execv, execle, execve, execlp, execvp, or exect Subroutine

Purpose

Executes a file.

Library

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

Syntax

int execl (*Path*, *Argument0* **[**, *Argument1*, **. . .], 0)**
char **Path*, **Argument0*, **Argument1*, **. . . ;**

int execle (*Path*, *Argument0* **[**, *Argument1*, **. . .], 0,**
EnvironmentPointer)
char **Path*, **Argument0*, **Argument1*,
. . . , **EnvironmentPointer*[**];**

int execlp (*File*, *Argument0* **[**, *Argument1*, **. . .], 0)**
char **File*, **Argument0*, **Argument1*, **. . . ;**

int execv (*Path*, *ArgumentV*)
char **Path*, **ArgumentV*[**];**

int execve (*Path*, *ArgumentV*,
EnvironmentPointer)
char **Path*, **ArgumentV*[**],** **EnvironmentPointer*[**];**

int execvp (*File*, *ArgumentV*)
char **File*, **ArgumentV*[**];**

extern char **environ;

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 extended COFF format. (See the **a.out** file.)
- 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 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* parameters concatenated together. The rest of the arguments passed are the same as those passed to the **exec** subroutine.

exect is included for compatibility with older programs being traced with the **ptrace** command. The program being executed is forced into hardware single–step mode.

Parameters

Path Specifies a pointer to the path name of the new process image file. If Network File System 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 run–time 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** header file.)

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.)

The **exec** subroutines cause the following changes in the privilege sets of the process:

- Upon **exec**, the *inherited* privilege set is assigned the value of the old *effective* privilege set.
- The *effective* and *maximum* privilege set are assigned the value of the logical union of the old *effective* privilege set and the privilege set assigned to the file named in the *Path* parameter.

The **exec** subroutines do not alter the value of the *TrustedState* parameter of the process.

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.)

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.

Profiling is disabled for the new process. (For information about profiling, see the **profil** subroutine.)

The new process inherits the following attributes from the calling process:

- The nice value (See the **getpriority** subroutine, **setpriority** subroutine, **nice** subroutine)
- The process ID
- The parent process ID
- The process group ID
- The semadj values (See the **semop** subroutine)

- The **tty** group ID (See the **exit**, **atexit**, **_exit** subroutines, **sigaction** subroutine)
- The **trace** flag (See request 0 of the **ptrace** subroutine)
- The time left until an alarm clock signal (See the **incinterval** subroutine, **setitimer** subroutine, and **alarm** subroutine)
- The current directory
- The root directory
- The file mode creation mask (See the **umask** subroutine)
- The file size limit (See the **ulimit** subroutine)
- The resource limits (See the **getrlimit** subroutine, **setrlimit** subroutine, and **vlimit** subroutine)
- The privileges (See the above discussion)
- The **utime**, **stime**, **cutime**, and **cstime** subroutines(See the **times** subroutine)
- The login user ID
- The suspend/resume process audit flag (See the **auditproc** subroutine)
- The general/special user audit flag.

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 that 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("/bin/sh", "sh", "-c", "li -l *.c", 0);
   ```

   This runs the **sh** (shell) 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 /**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:

   ```
   #! /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
```

causes the following command to be run:

```
/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 # (comment character 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 global variable **errno** is set to identify the error.

Error Codes

The **exec** subroutine fails and returns to the calling process if one or more of the following are true:

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 not an **execlp** subroutine or an **execvp** subroutine, and 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 **MAXMEM**.
E2BIG	The number of bytes in the new process argument list is greater than the system–imposed limit. This limit is defined as **NCARGS** in the **sys/param.h** header 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, and the translation tables at the server or client do not allow translation of this user or group ID.

The **exec** subroutines can also fail if one or more of the following conditions that apply to any service that requires path name resolution are true:

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 *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 will be written to standard error before the signal **SIGKILL** is sent to the process. If a shared library cannot be mapped, one or more of the following is true:

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 Network File System 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 AIX Base Operating System (BOS) Runtime.

Related Information

The **chmod, fchmod** subroutines, **exit** subroutine, **fcntl** subroutine, **fork** subroutine, **getrusage, times** subroutines, **incinterval, alarm** subroutines, **nice** subroutine, **profil** subroutine, **ptrace** subroutine, **semop** subroutine, **settimer** subroutine, **sigaction**, **signal**, **sigvec** subroutines, **shmat** subroutine, **system** subroutine, **ulimit** subroutine, **umask** subroutine.

The **varargs** macros.

The **a.out** file.

The **sh** command, **ksh** command.

The **environment** miscellaneous facility.

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*;

int atexit (*Function*)
void (**Function*) **(void))**;

Description

The **atexit** subroutine registers functions to be called at normal process termination for cleanup processing.

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. Finally, it 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:

- All of the file descriptors open in the calling process are closed. If Network File System is installed on your system, some of these files can be remote. Since 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 will be returned to it at that time.
- The parent process is sent a **SIGCHLD** signal when a child terminates; however, since the default action for this signal is to ignore it, the signal usually is not seen.
- Terminating a process by exiting does not terminate its child processes.
- Each attached shared memory segment is detached and the value of shm_nattch 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** 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**, **flock**, and **lockf** subroutines are removed.

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 up to 32 functions that are called at normal process termination for cleanup processing. A push–down stack of functions is kept, such 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 AIX Base Operating System (BOS) Runtime.

If the parent process of the calling process is not ignoring **SIGCHLD**, the calling process is transformed into a zombie process, and its parent process is sent a **SIGCHLD** signal to notify it of the death of a child process.

A zombie process is a process that occupies a slot in the process table, but has no other space allocated to it either in user of 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** header file.)

A process remains a zombie until its parent issues one of the **wait** subroutines. At this time, the zombie is *laid to rest*, and its process table entry is released.

Termination of 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 thus 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 will be 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, then a **SIGHUP** signal followed by a **SIGCONT** signal will be sent to each process in the newly–orphaned process group.

Related Information

The **acct** subroutine, **sigaction, signal, sigvec** subroutines, **times** subroutine, **wait, waitpid, wait3** subroutines.

exp, expm1, log, log10, log1p, or pow Subroutine

Purpose

Computes exponential, logarithm, and power functions.

Library

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

Syntax

#include <math.h>

double exp (*x*)
double *x*;

double expm1 (*x*)
double *x*;

double log (*x*)
double *x*;

double log10 (*x*)
double *x*;

double log1p (*x*)
double *x*;

double pow (*x*, *y*)
double *x*, *y*;

Description

These subroutines are used to compute exponential, logarithm, and power functions.

The **exp** subroutine returns `exp(x)`.

The **expm1** subroutine returns `exp(x)-1`.

The **log** subroutine returns the natural logarithm of *x*. The value of *x* must be positive.

The **log10** subroutine returns the logarithm base 10 of *x*. The value of *x* must be positive.

The **log1p** subroutine returns `log(1 + x)`.

The **pow** subroutine returns `x**y`. If *x* is negative or 0, then *y* must be an integer. If *y* is 0, then **pow** returns 1.0 for all *x*.

The **expm1** subroutine and **log1p** subroutine are useful to guarantee that financial calculations of `((1+x**n)-1)/x`, namely:

```
expm1(n * log1p(x))/x
```

are accurate when x is tiny (for example, when calculating small daily interest rates). These subroutines also simplify writing accurate inverse hyperbolic functions.

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

```
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 **libm.a** (**–lm**):

exp If the correct value would overflow, **exp** returns HUGE_VAL and **errno** is set to ERANGE.

log If *x* is less than zero, **log** returns the value NaNQ and sets **errno** to EDOM. If *x* equals zero, log returns the value –HUGE_VAL but does not modify **errno**.

log10 If *x* is less than zero, **log10** returns the value NaNQ and sets **errno** to EDOM. If *x* equals zero, log returns the value –HUGE_VAL but does not modify **errno**.

pow If the correct value overflows, **pow** returns HUGE_VAL and sets **errno** to ERANGE. If *x* is negative and *y* is not an integer, **pow** returns NaNQ and sets **errno** to EDOM. If *x* equals zero and *y* is negative, **pow** returns HUGE_VAL but does not modify **errno**.

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

exp If the correct value would overflow, **exp** returns HUGE_VAL. If the correct value would underflow, **exp** returns 0. In both cases **errno** is set to ERANGE.

log If *x* is non-positive, **log** returns the value –HUGE_VAL, and sets **errno** to EDOM. A message indicating DOMAIN error (or SING error when $x = 0$) is output to standard error.

log10 If *x* is non-positive, **log10** returns the value –HUGE_VAL 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 *y* is non-positive, or if *x* is negative and *y* is not an integer, **pow** 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 **pow** would overflow or underflow, **pow** returns + or – HUGE_VAL or 0 respectively and sets **errno** to ERANGE.

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

When using either **libm.a (–lm)** or **libmsaa.a (–lmsaa)**:

expm1 If the correct value overflows, **expm1** returns HUGE_VAL but does not modify **errno.**

log1p If $x < -1$, **log1p** returns the value NaNQ. If $x = -1$, **log1p** returns the value –HUGE_VAL. In neither case is **errno** modified.

Implementation Specifics

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

The **expm1** and **log1p** subroutines are not part of the ANSI C Library.

Related Information

The **hypot, cabs** subroutines, **sinh, cosh, tanh** subroutines, **matherr** subroutine.

fclear Subroutine

Purpose

Makes a hole in a file.

Library

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

Syntax

long fclear (*FileDescriptor*, *NumberOfBytes*)
int *FileDescriptor*;
unsigned long *NumberOfBytes*;

Description

The **fclear** subroutine zeros the number of bytes specified by the *NumberOfBytes* parameter starting at the current position of the file open on the file descriptor *FileDescriptor*. If Network File System is installed on your system, this file can reside on another node.

The **fclear** subroutine cannot be applied to a file that a process has opened with the **O_DEFER** mode. Successful completion of the **fclear** subroutine clears the *SetUserID* and *SetGroupID* attributes of the file if the calling process does not have root user authority.

Parameters

FileDescriptor	The file specified by the *FileDescriptor* parameter must be open for writing. 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	The number of bytes that the seek pointer is advanced. If you use the **fclear** subroutine 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 global variable **errno** is set to indicate the error.

Error Codes

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

EIO	I/O error.
EBADF	The *FileDescriptor* parameter 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.

If Network File System is installed on the system the **fclear** subroutine can also fail if the following is true:

ETIMEDOUT	The connection timed out.

Implementation Specifics

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

Related Information

The **truncate, ftruncate** subroutines, **open** subroutine.

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. **fclose** 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.

Parameter

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.

Implementation Specifics

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

Related Information

The **close** subroutine, **exit**, **atexit**, **_exit** subroutines, **fopen**, **freopen**, **fdopen** subroutines, **setvbuf**, **setbuf**, **setbuffer**, **setlinebuf** subroutines.

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*;

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 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.

General Record Locking Information:

Any lock is either an *enforced lock* or an *advisory lock*, and any lock is either a *read lock* or a *write lock*.

Warning: 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** header file provides 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** subroutines. In addition, the **open** and **ftruncate** subroutines are prevented from truncating the locked section of the file, and the **fclear** subroutine can not modify the locked section of the file. If another process attempts to read or modify the locked section of the file, it 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. This means that processes must voluntarily call **fcntl** 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.

Some general rules about file locking include:

- 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.
- When 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 running a **fork** subroutine.

Note: Deadlocks due to file locks in a distributed system are not always detected. When such deadlocks are possible, 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.

Parameters

FileDescriptor Specifies an open file descriptor obtained from a successful **open**, **fcntl**, or **pipe** subroutine.

Argument Specifies a variable that depends on the value of the *Command* parameter.

Command Specifies the operation to be performed. The following *Command* parameter values get a file descriptor or associated flags or set those flags:

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 set to remain open across **exec** subroutines.

F_GETFD Gets the **close–on–exec** flag (FD_CLOEXEC bit) associated with the file descriptor *FileDescriptor*. The *Argument* parameter is ignored.

F_SETFD Sets the **close–on–exec** flag (FD_CLOEXEC bit) associated with the *FileDescriptor* parameter to the value of the *Argument* parameter.

F_GETFL Gets the file status flags for the file referred to by the *FileDescriptor* parameter. The *Argument* parameter is ignored.

F_SETFL The *Argument* parameter specifies the desired flags. The following flags may be given:

- **O_APPEND** or **FAPPEND**
- **O_NDELAY** or **FNDELAY**
- **O_NONBLOCK** or **FNONBLOCK**
- **O_SYNC** or **FSYNC**
- **FASYNC**

O_NDELAY and **O_NONBLOCK** affect only operations against file descriptors derived from the same **open** subroutine. In BSD, these apply to all file descriptors that refer to the object.

F_GETLK Sets or clears a file lock.

F_SETLK Gets the first lock that blocks the lock described in the **flock** structure.

F_SETLKW Performs the same function as **F_SETLK** except that if a read or write lock is blocked by existing locks, the

	process sleeps until the section of the file is free to be locked.
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 the *Argument* parameter as negative; otherwise the *Argument* parameter is interpreted as a process ID.
F_CLOSEM	Closes all file descriptors from *Argument* up to **OPEN_MAX**.

Old Specifies an open file descriptor.

New Specifies an open file descriptor that is returned by the **dup2** subroutine.

Compatibility Interfaces

fnctl (*FileDescriptor, Command, Argument*)
is equivalent to:
lockfx (*FileDescriptor, Command, Argument*)

when the *Command* parameter is **F_SETLK**, **F_SETLKW**, or **F_GETLK**.

dup (*FileDescriptor*)
is equivalent to:
fnctl (*FileDescriptor,* **F_DUPFD**, *0*).

dup2 (*Old, New*)
is equivalent to:
fcntl(*Old* **F_DUPFD**, *New*)

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 global variable **errno** is set to indicate the error.

Error Codes

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

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

EMFILE The *Command* parameter is **F_DUPFD** and **OPEN_MAX** file descriptors are currently open.

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.

ESRCH The value of the *Command* parameter is **F_SETOWN** and the process ID given as *Argument* is not in use.

The **dup** and **dup2** subroutines fail if one or both of the following are true:

EBADF The *Old* parameter is not a valid open file descriptor or the *New* parameter file descriptor is out of range.

EMFILE The number of file descriptors exceeds **OPEN_MAX** or there is no file descriptor above the value of the *New* parameter.

If Network File System is installed on the system the **fcntl** subroutine can also fail if the following is true:

ETIMEDOUT The connection timed out.

Implementation Specifics

These subroutines are part of AIX 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 via use of the **ioctl** subroutine with the **FIOASYNC**, **FIOSETOWN**, and **FIOGETOWN** commands. However, a BSD compatible mechanism is also available if the application is linked with **libbsd**.

When using **libbsd**, asynchronous I/O is enabled by using **F_SETFL** with the **FASYNC** flag set in the *Argument* parameter. The **F_GETOWN** and **F_SETOWN** commands are used to get the current asynchronous I/O owner and to set the asynchronous I/O owner.

Related Information

The **close** subroutine, **execl**, **excecv**, **execle**, **execve**, **execlp**, **execvp**, **exect** subroutines, **lockf** subroutine, **openx**, **open**, **creat** subroutines.

The **fcntl.h** header file.

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. 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/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 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 input pointed to by the *Stream* parameter.

Note: Since this routine is implemented as a macro, it cannot be declared or redeclared.

Parameter

Stream	Specifies the input or output stream.

Implementation Specifics

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

Related Information

The **open** subroutine, **fopen**, **freopen**, **fdopen** subroutines.

floor, ceil, nearest, trunc, rint, itrunc, uitrunc, fmod, or fabs Subroutine

Purpose

The **floor** subroutine, **ceil** 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 **fabs** subroutine compute the Modulo Remainder and floating-point absolute value functions, respectively.

Library

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*;

double ceil (*x*)
double *x*;

double fmod *(x,y)*
double *x, y;*

double fabs *(x)*
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 returns the largest floating-point integer value not greater than the *x* parameter.

The **ceil** subroutine returns 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, the floating-point integer that is even is returned.

The **trunc** subroutine returns the nearest floating-point integer value to the *x* parameter in the direction of zero. 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 zero. This is equivalent to truncating the fraction bits from of 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 zero. This is equivalent to truncating off the fraction bits of the *x* parameter and then converting *x* to an unsigned integer.

The **fmod** subroutine computes the modulo floating-point remainder of *x/y*. The **fmod** subroutine returns the value *x–iy* for some *i* such that if *y* is non-zero, the result has the same sign as *x* and magnitude less than the magnitude of *y*.

The **fabs** subroutine returns the absolute value of *x*, |*x*|.

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

```
cc floor.c -lm
```

Parameters

x Specifies some double-precision floating-point value.

y Specifies some double-precision floating-point value.

Error Codes

The **itrunc** and **uitrunc** subroutines return LONG_MAX if *x* is greater than or equal to LONG_MAX and LONG_MIN if *x* is equal to or less than LONG_MIN. The **itrunc** subroutine returns LONG_MIN if *x* is a NaNQ or NaNS. The **uitrunc** subroutine returns zero if *x* is a NaNQ or NaNS. (LONG_MAX and LONG_MIN are defined in the **limits.h** header file.)

The **fmod** subroutine for (x/0) returns a NaNQ and sets the global variable **errno** to EDOM.

Implementation Specifics

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

The **itrunc**, **uitrunc**, **trunc**, **nearest**, and **rint** subroutines are not part of the ANSI C Library.

Related Information

The **fp_read_rnd, fp_swap_rnd** subroutines.

The ANSI C FLT_ROUNDS macro, which is in the **float.h** header file.

IEEE Standard for Binary Floating-Point Arithmetic (ANSI/IEEE Standards 754–1985 and 854–1987).

fopen, freopen, or fdopen Subroutine

Purpose

Opens a stream.

Library

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

Syntax

#include <stdio.h>

FILE *fopen (*Path*, *Type***)**
**char ***Path*, ***Type*;

FILE *freopen (*Path*, *Type*, *Stream***)**
**char ***Path*, ***Type*;
**FILE ***Stream*;

FILE *fdopen (*FileDescriptor*, *Type***)**
int *FileDescriptor*;
**char ***Type*;

Description

The **fopen** subroutine opens the file named by the *Path* parameter and associates a stream with it. The **fopen** subroutine returns 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**, **fsetpos**, 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 append (that is, when the *Type* parameter is **a** or **a+**), it is impossible to overwrite information already in the file. You can use the **fseek** subroutine to reposition the file pointer to any position in the file, but when output is written to the file, the current file pointer is ignored. All output is written at the end of the file and causes the file pointer to be repositioned to the end of the output.

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 that if the data is buffered, it is not actually written until it is flushed.

The **freopen** subroutine substitutes the named file in place of the open stream. The original stream is closed regardless of whether the **openx** subroutine succeeds. The **freopen** subroutine returns a pointer to the **FILE** structure associated with *Stream*. The **freopen** subroutine is typically used to attach the pre–opened streams associated with stdin, stdout, and stderr 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. Note that the *Type* of stream specified must agree with the mode of the open file.

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	Open text file for reading.
w	Create a new text file for writing, or open and truncate to zero length.
a	Append (open text file for writing at the end of the file, or create for writing).
rb	Open binary file for reading.
wb	Create a binary file for writing, or open and truncate to zero length.
ab	Append (open binary file for update, writing at the end of the file, or create for writing.)
r+	Open for update (reading and writing).
w+	Truncate or create for update.
a+	Append (open text file for update, writing at end of file,or create for writing).
r+b or **rb+**	Open binary file for update (reading and writing).
w+b or **wb+**	Create binary file for update, or open and truncate to zero length.
a+b or **ab+**	Append (Open a binary file for update, writing at the end of the file, or create for writing).

Note: The system does not distinguish between text and binary files. In the AIX Version 3 Operating System, 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 **fopen**, **fdopen**, or **freopen** subroutine fails, a **NULL** pointer is returned and the global variable **errno** is set to indicate the error.

Error Codes

The **fopen** subroutine fails if one or both of the following are true:

EACCES Search permission is denied on a component of the path prefix, or the file exists and the permissions specified by mode are denied, or the file does not exist and write permission is denied for the parent directory of the file to be created.

EINVAL The type of stream given to **fdopen** does not agree with the type of the already open file.

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

EINVAL The *Type* argument is not a valid type.

Implementation Specifics

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

POSIX: **w** and **w+** types do not truncate, and **a** and **a+** types do not create.

SAA: At least eight streams, including three standard text streams, can open simultaneously. Both binary and text modes are supported.

Related Information

The **fclose**, **fflush** subroutines, **fseek**, **rewind**, **ftell**, **fgetpos**, **fsetpos** subroutines, **setbuf**, **setvbuf**, **setbuffer**, **setlinebuf** subroutines.

The **open**, **openx**, **creat** subroutines.

fork or vfork Subroutine

Purpose

Creates a new process.

Libraries

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

vfork: Berkeley Compatibility Library (**libbsd.a**)

Syntax

#include <sys/types.h>

pid_t fork ( )

int vfork ( )

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** subroutines)
- Signal handling settings (that is, **SIG_DFL, SIG_IGN**, *Function Address*)
- Set user ID mode bit
- Set group ID mode bit
- **Inherited**, **effective**, and **maximum** privilege vectors
- Trusted state
- Profiling on/off status
- Nice value
- All attached shared libraries
- Process group ID
- **tty** group ID (described in the **exit, atexit**, and **_exit** subroutines, **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)
- List of auditable events

- Audit status flag
- 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 a unique process ID.
- The child process has a different parent process ID.
- The child process has its own copy of the parent process's file descriptors. However, each of the child's file descriptors shares a common file pointer with the corresponding file descriptor of the parent process.
- All **semadj** values are cleared. (Information about **semadj** values can be found in the **semop** subroutine.)
- Process locks, text locks, and data locks are not inherited by the child process. (Information about locks can be found in the **plock** subroutine.)
- If multi–process debugging is turned on, the trace flags are inherited from the parent; otherwise the trace flags are reset. (A discussion of request 0 can be found in the **ptrace** subroutine.)
- The child process's **utime**, **stime**, **cutime**, and **cstime** subroutines are set to 0. (More information can be found in the **getrusage**, **times**, and **vtimes** subroutines.)
- Any pending alarms are cleared in the child process. (More information can be found in the **incinterval** subroutine, **setitimer** subroutine, and **alarm** subroutine).
- The set of signals pending for the child process is initialized to the empty set.

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 global variable **errno** is set to indicate the error.

Error Codes

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

EAGAIN	The system–imposed limit on the total number of processes executing would be exceeded.
EAGAIN	The system–imposed limit on the total number of processes executing for a single user would be exceeded.
ENOMEM	There is not enough space left for this process.

Implementation Specifics

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

The **vfork** subroutine is supported as a compatibility interface for older BSD system programs, and can be used by compiling with Berkeley Compatibility Library (**libbsd.a**).

In the AIX Version 3 Operating System, the parent process is not forced to wait until the child either exits or execs, as it is 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.

Related Information

The **exec** subroutines, **_exit, exit, atexit** subroutines, **getrusage, times** subroutines, **getpriority, setpriority** subroutines, **nice** subroutine, **plock** subroutine, **ptrace** subroutine, **raise** subroutine, **semop** subroutine, **shmat** subroutine, **sigaction**, **signal**, **sigvec** subroutines, **ulimit** subroutine, **umask** subroutine, **wait, waitpid, wait3** subroutines.

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

The RISC System/6000 currently does not generate an interrupt for floating-point traps. Therefore, the common method of catching the signal SIGFPE and calling an appropriate trap handler to identify a floating-point trap is not supported.

These subroutines aid in manipulating floating-point traps and identifying the trap state and type.

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 trap(s) specified by *Mask* are enabled. Otherwise, 0 is returned.

The **fp_enable_all** subroutine enables all floating-point traps.

The **fp_enable** subroutine enables all floating-point trap(s) specified by *Mask.*

The **fp_disable_all** subroutine disables all floating-point traps.

The **fp_disable** subroutine disables all floating-point trap(s) specified by *Mask.*

Implementation Specifics

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

Related Information

The **fp_clr_flag, fp_set_flag, fp_read_flag, fp_swap_flag** subroutines, **fp_invalid_op, fp_divbyzero, fp_overflow, fp_underflow, fp_inexact, fp_any_xcp, fp_iop_snan, fp_iop_infsinf, fp_iop_infdinf, fp_iop_zrdzr, fp_iop_infmzr, fp_iop_invcmp** subroutines, **fp_read_rnd, fp_swap_rnd** subroutines.

The *IEEE Standard for Binary Floating-Point Arithmetic* (ANSI/IEEE Standards 754-1985 and 854-1987).

fp_clr_flag, fp_set_flag, fp_read_flag, or fp_swap_flag Subroutine

Purpose

These subroutines allow 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

The RISC System/6000 currently does not generate an interrupt for floating-point exceptions. Therefore, the common method of catching the signal SIGFPE and calling an appropriate trap handler to identify a floating-point exception is not supported.

These subroutines aid in determining 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*, there are five types of floating-point operations that must be signaled when detected in a floating-point operation. They are: Invalid Operation, Division by Zero, Overflow, Underflow, and 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...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 can not clear them. You can clear a flag in the floating-point exception status using an explicit software action such as **fp_swap_flag (0).**

The header file **fpxcp.h** defines the following names for the individual flags in the floating-point exception status:

FP_INVALID	Invalid operation summary
FP_OVERFLOW	Overflow
FP_UNDERFLOW	Underflow
FP_DIV_BY_ZERO	Divide by zero
FP_INEXACT	Inexact result

In addition to the above flags, the AIX for RISC System/6000 supports additional information about the cause of an Invalid Operation exception. The following flags are included in the floating-point exception status and defined in the **fpxcp.h** header file. The flag number for each exception type varies, but the mnemonics are the same for all ports. The Invalid Operation detail flags are not required for conformance to the AIX for RISC System/6000.

FP_INV_NANS	Signalling 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_REM_Y0	Remainder (x,y) with y=0
FP_INV_REM_X1	Remainder (x,y) with x=INF
FP_INV_SQRT	Square root of a negative number
FP_INV_CVI	Conversion to integer error

Parameters

Mask A 32-bit pattern that identifies floating-point exception flags.

Return Values

The **fp_clr_flag** (*Mask*) subroutine resets the exception status flag(s) defined by *Mask* to 0 (false). The remaining flags in the exception status are unchanged. The return value is that of the exception status before the reset.

The **fp_set_flag** (*Mask*) subroutine sets the exception status flag(s) defined by *Mask* to 1 (true). The remaining flags in the exception status are unchanged. The return value is that of the exception status before the set.

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** (*Mask*) subroutine writes *Mask* into the floating-point status and returns the floating-point exception status from before the write.

You can 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 AIX Base Operating System (BOS) Runtime.

Related Information

The **fp_invalid_op, fp_divbyzero, fp_overflow, fp_underflow, fp_inexact, fp_any_xcp, fp_iop_snan, fp_iop_infsinf, fp_iop_infdinf, fp_iop_zrdzr, fp_iop_infmzr, fp_iop_invcmp** subroutines.

The **fp_read_rnd, fp_swap_rnd** subroutines.

The **fp_any_enable, fp_is_enabled, fp_enable_all, fp_enable, fp_disable_all, fp_disable** subroutines.

IEEE Standard for Binary Floating-Point Arithmetic (ANSI/IEEE Standards 754-1985 and 854-1987) describes the IEEE floating-point exceptions.

fp_invalid_op, fp_divbyzero, fp_overflow, fp_underflow, fp_inexact, fp_any_xcp, fp_iop_snan, fp_iop_infsinf, fp_iop_infdinf, fp_iop_zrdzr, fp_iop_infmzr, or fp_iop_invcmp 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()

int fp_iop_snan()
int fp_iop_infsinf()

int fp_iop_infdinf()
int fp_iop_zrdzr()

int fp_iop_infmzr()
int fp_iop_invcmp()

Description

The RISC System/6000 currently does not generate an interrupt for floating-point exceptions. Therefore, the common method of catching the signal SIGFPE and calling an appropriate trap handler to identify the floating-point exception is not supported.

These subroutines aid in determining 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.

Return Values

The **fp_invalid_op** subroutine returns 1 if a floating-point invalid operation exception status flag is set. Otherwise, 0 is returned.

The **fp_divbyzero** subroutine returns 1 if a floating-point divide by zero exception status flag is set. Otherwise, 0 is returned.

The **fp_overflow** subroutine returns 1 if a floating-point overflow exception status flag is set. Otherwise, 0 is returned.

The **fp_underflow** subroutine returns 1 if a floating-point underflow exception status flag is set. Otherwise, 0 is returned.

The **fp_inexact** subroutine returns 1 if a floating-point inexact exception status flag is set. Otherwise, 0 is returned.

The **fp_any_xcp** subroutine returns 1 if a floating-point invalid operation, divide by zero, overflow, underflow, or inexact exception status flag is set. Otherwise, 0 is returned.

The following routines are available for the AIX for RISC System/6000 platform only:

The **fp_iop_snan** subroutine returns 1 if a floating-point invalid operation exception status flag is set due to a signalling NaN (NaNS). Otherwise, 0 is returned.

The **fp_iop_infsinf** subroutine returns 1 if a floating-point invalid operation exception status flag is set due to a INF–INF. Otherwise, 0 is returned.

The **fp_iop_infdinf** subroutine returns 1 if a floating-point invalid operation exception status flag is set due to a INF/INF. Otherwise, 0 is returned.

The **fp_iop_zrdzr** subroutine returns 1 if a floating-point invalid operation exception status flag is set due to a 0.0/0.0. Otherwise, 0 is returned.

The **fp_iop_infxzr** subroutine returns 1 if a floating-point invalid operation exception status flag is set due to a INF*0.0. Otherwise, 0 is returned.

The **fp_iop_invcmp** subroutine returns 1 if a floating-point invalid operation exception status flag is set due to a compare involving a NaN. Otherwise, 0 is returned.

Implementation Specifics

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

Related Information

The **fp_read_rnd, fp_swap_rnd** subroutines, **fp_clr_flag, fp_set_flag, fp_read_flag, fp_swap_flag** subroutines, **fp_any_enable, fp_is_enabled, fp_enable_all, fp_enable, fp_disable_all, fp_disable** subroutines.

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 4 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 header file **float.h** 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

Note: For IBM AIX Version 3 for RISC System/6000, the *ANSI C Standard* macro FLT_ROUNDS is defined in **float.h** as an invocation of **fp_read_rnd**. The *ANSI C Standard* does not specify a mechanism for changing the rounding mode.

The subroutine **fp_swap_rnd** 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 FP_RND_RZ, FP_RND_RN, FP_RND_RP, or FP_RND_RM.

Implementation Specifics

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

Related Information

The **floor, ceil, nearest, trunc, rint, itrunc, uitrunc, fmod, fabs** subroutines, **fp_clr_flag, fp_set_flag, fp_read_flag, fp_swap_flag** subroutines, **fp_any_enable, fp_is_enabled, fp_enable_all, fp_enable, fp_disable_all, fp_disable** subroutines.

fread or fwrite Subroutine

Purpose

Performs binary input/output.

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 ((void *) *Pointer*, *Size*, *NumberOfItems*, *Stream*)
size_t *Size*, *NumberOfItems*;
FILE **Stream*;

Description

The **fread** subroutine copies *NumberOfItems* items of data from the input stream into an array beginning at the location pointed to by the *Pointer* parameter. Each data item has the type **Pointer*.

The **fread** subroutine stops copying bytes if an end–of–file 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. It leaves the file pointer of the *Stream* parameter, if defined, pointing to the byte following the last byte read, if there is one. The **fread** subroutine does not change the contents of the *Stream* parameter.

The **fwrite** subroutine appends *NumberOfItems* items of data of the type **Pointer* from the array pointed to by the *Pointer* parameter to the output stream.

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.

Parameters

Pointer	Points to an array.
Size	Specifies the size of the variable type of the array pointed to by the *Pointer* parameter.
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 is negative or 0, no characters are transferred, and a value of 0 is returned.

Implementation Specifics

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

Related Information

The **fopen**, **freopen**, **fdopen** subroutines, **getc**, **fgetc**, **getchar**, **getw**, **getwc**, **fgetwc**, **getwchar** subroutines, **gets**, **fgets**, **getws**, **fgetws** subroutines, **printf**, **fprintf**, **sprintf**, **NLprintf**, **NLfprintf**, **NLsprintf** subroutines, **putc**, **putchar**, **fputc**, **putw**, **putwc**, **putwchar**, **fputwc** subroutines, **puts**, **fputs**, **putws**, **fputws** subroutines, **read** subroutine, **scanf**, **fscanf**, **sscanf**, **NLscanf**, **NLfscanf**, **NLsscanf** subroutines, **write** subroutine.

frevoke Subroutine

Purpose

Revokes access to a file by other processes.

Library

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

Syntax

int frevoke(*Fildescriptor***)**
int *Fildescriptor*;

Description

The **frevoke** subroutine revokes access to a file by other processes.

All accesses to the file are revoked, except through the file descriptor provided as the *Fildescriptor* parameter to the **frevoke** subroutine. Subsequent attempts to access the file using another file descriptor established before the **frevoke** subroutine fail and cause the process to be killed.

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 assure 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.

Parameter

Fildescriptor	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 global variable **errno** is set to indicate the error.

Error Codes

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

EBADF	The *Fildescriptor* parameter 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	Access rights revocation is not implemented for this file.

Implementation Specifics

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

Related Information

The **revoke** subroutine.

frexp, ldexp, or modf Subroutine

Purpose

Manipulates floating-point numbers.

Library

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

Syntax

#include <math.h>

double frexp (*Value*, *Exponent***)**
double *Value*;
int **Exponent*;

double ldexp (*Mantissa*, *Exponent***)**
double *Mantissa*;
int *Exponent*;

double modf (*Value*, *IntegerPointer***)**
double *Value*, **IntegerPointer*;

Description

Every non-zero number can be written uniquely as x * 2**n, where the mantissa (fraction) *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 **ldexp** subroutine multiplies a floating-point number by an integral power of 2.

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 as a *double* in the location pointed to by the *IntegerPointer* parameter.

Parameters

Value	Specifies some double-precision floating-point value.
Exponent	For **frexp**, specifies an integer pointer to store the exponent; for **ldexp**, some integer value.
Mantissa	Specifies some double-precision floating-point value.
IntegerPointer	Specifies a double pointer in which to store the signed integral part.

Return Values

The **frexp** subroutine returns a value *x* such that *x* is in (0.5, 1.0) or is 0, and the *Value* parameter equals x * 2**(**Exponent*). If the *Value* parameter is zero, **Exponent* and *x* are zero. If the *Value* parameter is a NaN, *x* is a NaNQ and **Exponent* is set to LONG_MIN. If the *Value* parameter is +/–INF, *x* is +/– 0.0, and **Exponent* is set to +/– LONG_MAX.

The **ldexp** subroutine returns the value x * 2**(*Exponent*).

The **modf** subroutine returns the signed fractional part of *Value* and stores the signed integral part in the object pointed to by *IntegerPointer.* If *Value* is a NaN, then a NaNQ is returned and a NaNQ is stored in the object pointed to by *IntegerPointer.* If *Value* is +/–INF, then +/– 0.0 is returned, and +/–INF is stored in the object pointed to by *IntegerPointer.*

Error Codes

If the result of the **ldexp** subroutine overflows, then +/– HUGE_VAL is returned, and the global variable **errno** is set to ERANGE.

If the result of the **ldexp** subroutine underflows, 0 is returned, and the global variable **errno** is set to ERANGE.

Implementation Specifics

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

Related Information

The **sgetl**, **sputl** subroutines, **scanf**, **fscanf**, **sscanf**, **NLscanf**, **NLFscanf**, **NLsscanf** subroutines.

fscntl Subroutine

Purpose

Controls file system control operations.

Library

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

Syntax

#include <sys/types.h>
#include <sys/fscntl.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 journalled file system, supports any commands via the **fscntl** subroutine. The only supported command is **FS_EXTENDFS**. This is used to increase the size of a mounted file system.

Note: Application programs should not call this function, as it 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** header 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 global variable **errno** 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 AIX Base Operating System (BOS) Runtime.

fseek, rewind, ftell, fgetpos, or fsetpos 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 *Offset*;
int *Whence*;

void rewind (*Stream*)
FILE **Stream*;

long ftell (*Stream*)
FILE **Stream*;

int fsetpos (*Stream*, *Position*)
FILE **Stream*;
fpos_t *Position*;

int fgetpos (*Stream*, *Position*)
FILE **Stream*;
fpos_t *Position*;

Description

The **fseek** subroutine sets the position of the next input or output operation on the I/O stream specified by the *Stream* parameter. The position of the next operation is determined by the *Offset* parameter, which can be either positive or negative.

The **fseek** subroutine sets the file pointer associated with the specified *Stream* as follows:

- If the *Whence* parameter is 0, the pointer is set to the value of the *Offset* parameter.
- If the *Whence* parameter is 1, the pointer is set to its current location plus the value of the *Offset* parameter.
- If the *Whence* parameter is 2, the pointer is set to the size of the file plus the value of the *Offset* parameter.

The **fseek** subroutine fails 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 **rewind** subroutine is equivalent to **seekdir** (*Stream*, **(long) 0, 0**), except that it does not return a value.

The **fseek** and **rewind** subroutines undo any effects of the **ungetc** subroutine.

A successful call to the **fsetpos** subroutine clears the **EOF** indicator and undoes any effects of the **ungetc** subroutine.

After an **fseek** or a **rewind**, the next operation on a file opened for update can be either input or output.

The **fgetpos** subroutine is similar to the **ftell** subroutine and the **fsetpos** subroutine is similar to the **fseek** subroutine. The **fgetpos** subroutine stores 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** subroutine sets the file position indicator according to the value of the *Position* parameter, returned by a prior call to the **fgetpos** subroutine.

Parameters

Stream Specifies the 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** subroutine returns a value of 0. Otherwise, a nonzero value is returned.

The **ftell** subroutine returns the offset of the current byte relative to the beginning of the file associated with the named stream.

Upon successful completion, the **fgetpos** and **fsetpos** subroutines return 0. Otherwise, a value of –1 is returned and the global variable **errno** is set to **EINVAL**.

Implementation Specifics

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

Related Information

The **lseek** subroutine, **fopen**, **freopen**, **fdopen** subroutines.

fsync Subroutine

Purpose

Writes changes in a file to permanent storage.

Library

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

Syntax

int fsync (*FileDescriptor***)**
int *FileDescriptor*;

Description

The **fsync** subroutine causes all modified data in the file open on 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 some process has opened for deferred update (with **O_DEFER**) will not be written to permanent storage until some process issues an **fsync** subroutine against this file, or until some process runs a synchronous **write** system call (with **O_SYNC**) to this file. See the **fcntl.h** header file and the **open** subroutine for descriptions of **O_DEFER** and **O_SYNC**.

Note: The file identified by the *FileDescriptor* parameter must be open for writing when the **fsync** subroutine is issued or the call fails. This restriction was not enforced in BSD systems.

Parameter

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 global variable **errno** is set to indicate the error.

Error Codes

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

EIO	I/O error.
EBADF	*FileDescriptor* is not a valid file descriptor open for writing.
EINVAL	The file is not a regular file.

Implementation Specifics

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

Related Information

The **open** subroutine, **sync** subroutine, **write** subroutine.

The **fcntl.h** header file.

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*;
char *ID*;

Description

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 of creating keys, but many other methods are possible. Another way to do this, for example, is to use the project ID as the most significant byte of the key, and to use the remaining portion as a sequence number.

Warning: It is important for each installation to define standards for forming keys. If some standard is not adhered to, it is possible for unrelated processes to interfere with each other's operation.

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

Upon successful completion, 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 is a value of 0 ('\0').

Warning: If the *Path* parameter of the **ftok** subroutine names a file that has been removed while keys still refer to it, then the **ftok** subroutine returns an error. If that file is then recreated, the **ftok** subroutine will probably return a different key than the original one.

Implementation Specifics

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

Related Information

The **msgget** subroutine, **semget** subroutine, **shmget** subroutine.

ftw 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***) ( );**
int *Depth*;

Description

The **ftw** subroutine recursively searches the directory hierarchy that descends from the directory specified by the *Path* parameter.

For each file in the hierarchy, the **ftw** subroutine calls the function specified by the *Function* parameter, 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. (See the **stat** system call for more information about this structure.)

The integer passed to the *Function* parameter identifies the file type, and it has one of the following values:

FTW_F Regular file

FTW_D Directory

FTW_DNR Directory that cannot be read

FTW_NS A file for which the **stat** structure could not be executed successfully.

If the integer is FTW_DNR, then the files and subdirectories contained in that directory are not processed.

If the integer is FTW_NS, then 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** subroutine finishes processing a directory before processing any of its files or subdirectories.

The **ftw** subroutine continues 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** subroutine, such as an I/O error.

The **ftw** subroutine uses 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** subroutine runs faster if the value of the *Depth* parameter is at least as large as the number of levels in the tree. However, 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 negative, the effect is the same as if it were 1.

Because the **ftw** subroutine is 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** subroutine uses the **malloc** subroutine to allocate dynamic storage during its operation. If the **ftw** subroutine is terminated prior to its completion, such as by the **longjmp** subroutine being executed by the function specified by the *Function* parameter or by an interrupt routine, the **ftw** 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 Specifies the file type.

Depth Specifies the maximum number of file descriptors to be used.

Return Values

If the directory hierarchy is completed, the **ftw** subroutine returns a value of 0. If the function specified by the *Function* parameter returns a nonzero value, the **ftw** subroutine stops its search and returns the value that was returned by the function.

Error Codes

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

Implementation Specifics

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

Related Information

The **malloc, free, realloc, calloc, mallopt, mallinfo, alloca** subroutines, **setjmp, longjmp** subroutines, **signal** subroutine, **stat** subroutine.

getc, fgetc, getchar, 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 ( )

int getw (*Stream***)**
FILE ****Stream*;**

Description

The **getc** macro returns the next byte from the input specified by the *Stream* parameter and moves the file pointer, if defined, ahead one byte in *Stream*. 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, **getc** does not work correctly with a *Stream* parameter that has side effects. In particular, the following does not work:

```
getc(*f++)
```

In cases like this, use the **fgetc** subroutine instead.

The **fgetc** subroutine performs the same function as the **getc** macro, but **fgetc** is a genuine subroutine, not a macro. The **fgetc** subroutine runs more slowly than **getc**, but takes less space.

The **getchar** macro returns the next byte from **stdin.** the standard input stream. Note that **getchar** is also a macro.

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 possible 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 **getw** on a different type of processor.

Parameter

Stream A pointer to the file structure of an open file.

Return Values

These subroutines and macros return the integer constant **EOF** at the end of the file or upon an error.

Implementation Specifics

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

Related Information

The **fopen, freopen, fdopen** subroutines, **fread, fwrite** subroutines, **getwc**, **fgetwc**, **getwchar** subroutines, **gets, fgets** subroutines, **putc, putchar, fputc, putw** subroutines, **scanf, fscanf, sscanf, NLscanf, NLfscanf, NLsscanf, wsscanf** subroutines.

getcwd Subroutine

Purpose

Gets the path name of the current directory.

Library

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

Syntax

char *getcwd (*Buffer*, *Size***)**
**char ***Buffer*;
int *Size*;

Description

The **getcwd** subroutine returns a pointer to a string containing the path name of the current directory.

The **getcwd** subroutine calls the **getwd** subroutine to obtain the path name.

Parameters

Buffer	Pointer to a string space to hold the path name. If the *Buffer* parameter 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.
Size	The length of the string space. The value of the *Size* parameter must be at least 2 greater than the length of the path name to be returned.

Return Values

If the **getcwd** subroutine fails, a **NULL** value is returned and the global variable **errno** is set to indicate the error. The **getcwd** subroutine fails if the *Size* parameter is not large enough or if an error occurs in a lower–level function.

Error Codes

The **getcwd** subroutine fails if one or both of the following are true:

EINVAL	The *Size* argument is 0 or negative.
ERANGE	The *Size* argument is greater than 0 but is smaller than the length of the path name plus 1.

Implementation Specifics

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

Related Information

The **malloc** subroutine, **getwd** subroutine.

getdtablesize Subroutine

Purpose

Gets the descriptor table size.

Library

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

Syntax

int getdtablesize ( )

Description

Each process has a fixed–size descriptor table, which is guaranteed to have at least 2000 slots. The entries in the descriptor table are numbered with small integers starting at 0.

Return Value

The **getdtablesize** subroutine returns the size of the descriptor table.

Implementation Specifics

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

Related Information

The **close** subroutine, **open** subroutine, **select** subroutine.

getenv or NLgetenv Subroutine

Purpose

Returns the value of an environment variable.

Library

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

Syntax

char *getenv (*Name***)**
**char ***Name*

char *NLgetenv (*Name***)**
**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 since they are frequently set with shell commands.

The **NLgetenv** subroutine gets an NLS parameter from the locale information set up by a call to the **setlocale** subroutine. This parameter should belong in one of the following categories:

LC_MONETARY
LC_NUMERIC
LC_TIME
LC_MESSAGES

If the information solicited is not found in the tables set up by the **setlocale** subroutine, an American English default table is searched and the value in that default table is returned. If no data can be found, a **NULL** pointer is returned.

Parameters

Name	The name of an environment variable; can be null.

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.

Implementation Specifics

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

The **NLgetenv** subroutine is not part of the ANSI C Library.

Related Information

The **setlocale** and **putenv** subroutine.
National Language Support Overview in *Programming Concepts and Procedures.*

getfsent, getfsspec, getsfile, 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 file, opening the file if necessary.

The **setfsent** subroutine opens the file and positions to the first record.

The **endfsent** subroutine closes the 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, so it must be copied if it is to be saved.

Parameters

Special	Specifies the file system file name.
File	Specifies the file name.
Type	Specifies the file system type.

Return Value

The **getfsent, getfsspec, getfstype,** and **getsfile** subroutines return a pointer to a structure that contains information about a file system. The header file **fstab.h** describes the structure. A pointer to **NULL** is returned on **EOF** or error.

Implementation Specifics

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

getgid or getegid Subroutine

Purpose

Gets the process group IDs.

Library

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

Syntax

#include <sys/types.h>

gid_t getgid ()

gid_t getegid ()

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.

Implementation Specifics

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

Related Information

The **getgidx** subroutine, **getgroups** subroutine, **setgidx** subroutine, **setgroups** subroutine, **setgid** subroutine, **initgroups** subroutine.

The **setgroups** command, **groups** command.

getgidx Subroutine

Purpose

Gets the process group IDs.

Library

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

Syntax

#include <sys/id.h>

uid_t getgidx (*Which***)**
int *Which***;**

Description

The **getgidx** subroutine returns the specified group ID of the current process.

Parameter

Which — Specifies which group ID to return. The valid values for this parameter are defined in **sys/id.h** and include:

- **ID_EFFECTIVE** — Returns the effective group ID of the process.
- **ID_REAL** — Returns the real group ID of the process.
- **ID_SAVED** — Returns the saved group ID of the process.

Return Values

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

Error Code

The **getgidx** subroutine fails if:

EINVAL — The *Which* parameter is not one of **ID_EFFECTIVE**, **ID_REAL**, or **ID_SAVED**.

Implementation Specifics

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

Related Information

The **getgroups** subroutine, **setgidx** subroutine, **setgroups** subroutine, **getgid** subroutine, **setgid** subroutine, **initgroups** subroutine.

getgrent, getgrgid, getgrnam, putgrent, setgrent or endgrent Subroutine

Purpose

Accesses the basic group information in the user database.

Library

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

Syntax

#include <grp.h>

struct group *getgrent ( )

struct group *getgrgid (*Gid***)**
gid_t *Gid*;

struct group *getgrnam (*Name***)**
char **Name*;

int putgrent (*Group, File***)**
struct group *Group*;
FILE **File*;

void setgrent ( )

void endgrent ( )

Description

These subroutines may be used to access the basic group attributes. These attributes can also be accessed with the **getgroupattr** subroutine, which can access all group attributes and offer better granularity of access.

The **setgrent** subroutine opens the user database (if not already open) and rewinds 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, **getgrnam** returns the first group in the data base whose name matches the *Name* parameter and **getgrgid** returns the first group in the data base whose group ID matches the *Gid* parameter. The **endgrent** subroutine will close the user data base.

The **putgrent** subroutine writes a group entry to a file in the colon separated format of the **/etc/group** file. Note that an exclamation mark '!' will be written into the **gr_passwd** field and this field is ignored and is only there for compatibility with older versions of UNIX.

The **group** structure, which is returned by the **getgrent, getgrnam**, and **getgrgid** subroutines, is defined in the **grp.h** header file, and it contains the following members:

gr_name The name of the group.

gr_passwd The password of the group. Note that this field is no longer used by the system and so its value is meaningless.

gr_gid The ID of the group.

gr_mem The members of the group.

If NIS is enabled on the system, these routines will attempt to retrieve the group information from the NIS authentication server.

Warning: The information that is returned by the **gretgrent, getgrnam** and **getgrgid** subroutines is stored in a static area and will be overwritten on subsequent calls to these routines. If it is to be saved, it should be copied.

Warning: These subroutines should **not** be used in conjunction with the **getgroupattr** subroutine. The results are unpredictable.

Parameters

Gid Specifies the group ID of the group for which the basic attributes are to be read.

Name Specifies the name of the group for which the basic attributes are to be read.

Return Values

The **getgrent**, **getgrnam,** and **getgrgid** subroutines return a pointer to a valid group structure if successful. Otherwise, a NULL pointer is returned.

Implementation Specifics

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

Related Information

The **getpwent** subroutine, **getgroupattr** subroutine, **getuserattr** subroutine, **setuserdb** subroutine.

getgroupattr, IDtogroup, nextgroup, or putgroupattr Subroutine

Purpose

Accesses the group information in the user database.

Library

Security Library (**libs.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

These subroutines may be used to access group information. Because of their greater granularity and extensibility, these should be used instead of the **getgrent, putgrent, getgrnam, getgrgid, setgrent**, and **endgrent** subroutines.

The **getgroupattr** subroutine reads a specified attribute from the group data base. If the data base is not already open, the **getgroupattr** subroutine will do an implicit open for reading.

The **putgroupattr** subroutine writes a specified attribute into the group data base. If the data base is not already open, the **putgroupattr** subroutine will do an implicit open for reading and writing. The data changed by **putgroupattr** must be explicitly committed by calling the **putgroupattr** subroutine with a *Type* parameter which includes the **SEC_COMMIT** value. Until the data is committed, only the **get** subroutine calls within the process will return the written data.

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 data base. The consistency of consecutive searches depends upon the underlying storage access mechanism and is not guaranteed by this function.

Values which are returned by these subroutines are in dynamically allocated buffers and need not be moved prior to the next call.

Note: These functions and the **setpwent** and **setgrent** functions should not be used simultaneously. The result can be unpredictable.

The **setuserdb** and **enduserdb** subroutines should be used to open and close the user data base.

Parameters

Argument The *Argument* parameter is presently unused and must be specified as NULL.

Attribute Specifies the name of the attribute which is to be read. This can be one of the following, which are defined in the **usersec.h** file:

S_ID The group ID. Type: **SEC_INT.**

S_USERS The members of the group. Type: **SEC_LIST**.

S_ADMS The administrators of the group. Type: **SEC_LIST**.

S_ADMIN Defines the administrative status of a group. Type: **SEC_BOOL**

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. This parameter can be used to delimit the search to one or more user credential data bases. Specifying a non_NULL *Mode* also implicitly rewinds the search. A NULL mode should be used to continue the search sequentially through the data base. This attribute may include one or more of the following values specified as a bit mask; these are defined in the **usersec.h** file:

S_LOCAL The local data base of groups will be 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 **getuserattr** subroutine and the buffer supplied by the **putuserattr** subroutine is 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 list of NULL terminated character strings. The list itself is NULL terminated.

SEC_BOOL The format of the attribute is an integer where zero indicates false and non–zero indicates true.

SEC_COMMIT For the **putgroupattr** 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 will be committed.

SEC_DELETE The corresponding attribute will be deleted from the data base.

SEC_NEW Updates all the group data base files with the new group name when using the **putgroupattr** subroutine.

Value Specifies the address of a buffer in which the attribute is to be stored (**getgroupattr**) or is stored (**putgroupattr**).

Security

file access The calling process must have access to the group information in the user data base. This includes:

modes	file	
rw	/etc/group	(write access for **putgroupattr**)
rw	/etc/security/group	(write access for **putgroupattr**)

Return Values

The **getroupattr** and **putgroupattr** subroutines, when successfully completed, return a value of 0. Otherwise, a value of –1 is returned and **errno** 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 **errno** is set to indicate the error.

Error Codes

These subroutines fail if the following is true:

EACCES Access permission is denied for the data request.

All of these functions will return errors from other functions.

The **getgroupattr** and **putgroupattr** subroutines fail if one or more of the following is true:

ENOATTR The specified group attribute does not exist for this group.

ENOENT The specified *Group* 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.

EINVAL The *Value* parameter does not point to a valid buffer or to valid data for this type of attribute.

EINVAL The *Type* parameter does not contain only one of **SEC_INT**, **SEC_BOOL**, **SEC_CHAR**, or **SEC_LIST** or **SEC_COMMIT**.

EINVAL The *Type* parameter specifies that an individual attribute is to be committed and the *Group* parameter is NULL.

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 is true:

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 AIX Base Operating System (BOS) Runtime.

Related Information

The **getuserattr** subroutine, **getuserpw** subroutine, **setuserdb** subroutine, **setpwdb** subroutine.

getgroups Subroutine

Purpose

Gets the concurrent group set of the current process.

Library

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

Syntax

#include <grp.h>

int getgroups (*Ngroups*, *Gidset***)**
int *Ngroups*;
gid_t **Gidset*;

Description

The **getgroups** subroutine gets the concurrent group set 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 **NGROUPS_MAX** entries. (**NGROUPS_MAX** is a constant defined in the **limits.h** header file.) If *Ngroups* is zero, the **getgroups** subroutine returns the number of groups in the concurrent group set.

Parameters

Gidset Pointer to the array in which the process's concurrent group set of the user 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 **getgroups** fails, then a value of −1 is returned and **errno** is set to indicate the error.

Error Codes

The **getgroups** subroutine fails if the following 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 argument *Ngroups* is smaller than the number of groups in the concurrent group set.

Implementation Specifics

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

Related Information

The **setgroups** subroutine, **getgidx** subroutine, **setgidx** subroutine.

The **getgid** subroutine, **setgid** subroutine, **initgroups** subroutine.

The **setgroups** command, **groups** command.

getinterval, incinterval, absinterval, resinc, resabs, alarm, ualarm, getitimer or setitimer Subroutine

Purpose

Manipulate the expiration time of interval timers.

Library

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

Syntax

#include <sys/times.h>

int getinterval (*Timerid*, *Value*)
timer_t *Timerid*;
itimerstruc_t * *Value*;

int incinterval (*Timerid*, *Value*, *Ovalue)*
timer_t *Timerid*;
itimerstruc_t * *Value*, * *Ovalue*;

int absinterval (*Timerid, Value*, *Ovalue*)
timer_t *Timerid*;
itimerstruc_t * *Value*, * *Ovalue*;

int resabs (*Timerid*, *Resolution*, *Maximum*)
timer_t *Timerid;*
timestruc_t * *Resolution*, * *Maximum*;

int resinc (*Timerid*, *Resolution*, *Maximum*)
timer_t *Timerid*;
timestruc_t * *Resolution*, * *Maximum*;

unsigned int alarm (*Seconds*)
unsigned int *Seconds*;

unsigned int ualarm (*Value*, *Intvl*)
unsigned int *Value*, *Intvl*;

int setitimer (*Which*, *Value*, *Ovalue*)
int *Which*;
struct itimerval * *Value*;
struct itimerval * *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 **itimerstruc_t** structure, which includes the following members:

```
timestruc_t it_interval;   /* timer interval period  */
timestruc_t it_value;      /* timer interval expiration  */
```

If the **it_value** member is non–zero, 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 non–zero, it specifies a value to be used in reloading **it_value** when the timer expires. If **it_interval** is 0, the timer is to be disabled after its next expiration (assuming **it_value** is non–zero).

The **getinterval** subroutine returns an **itimerstruc_t** value to the *Value* parameter. The **it_value** member of this structure represents the amount of time in the current interval before the timer expires, should one exist (or 0 if not) for the per–process timer specified in the *Timerid* parameter. The **it_interval** member has the value last set by the **incinterval** or **absinterval** subroutines. The members 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, absinterval will succeed and the expiration notification will be made. Both functions 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 non–privileged user attempts to submit a fine granularity timer (i.e., a timer request less than 10 milliseconds), the timer request is raised to 10 milliseconds.

Parameters

Timerid — The id of the interval timer.

Value — Pointer to a **itimerstruc_t** structure.

Ovalue — Represents the previous amount of time before the timer would have expired.

Resolution — Resolution of the timer.

Maximum — Maximum value of the interval timer.

Compatibility Interface

The **alarm, ualarm, getitimer.** and **setitimer** subroutines are provided for compatibility with older AIX, AT&T System V, and BSD systems.

The **alarm**, **ualarm**, and **setitimer** subroutines are implemented to call the **incinterval** subroutine with the appropriate flag set.

The **getitimer** subroutine is implemented as a call to the **getinterval** subroutine.

Return Values

If these subroutines are successful, a 0 is returned. A return value of –1 indicates that an error occurred and **errno** 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**, or zero if there was no previous **alarm** request.

Error Codes

If the **getinterval**, **incinterval**, **absinterval**, **resinc** or **resabs** subroutine fails, a –1 is returned and **errno** is set to one of the following error codes:

EINVAL The *Timerid* parameter does not correspond to an id returned by the **gettimerid** subroutine.

A value structure specified a nanosecond value less than zero or greater than or equal to one 1000 million.

EIO An error occurred while accessing the timer device.

Implementation Specifics

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

Related Information

The **gettimer** subroutine, **gettimerid** subroutine.

getlogin Subroutine

Purpose

Gets the user's login name.

Library

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

Syntax

char *getlogin ( )

Description

The **getlogin** subroutine returns a pointer to the login name as found in the **/etc/utmp** file. Use the **getlogin** subroutine in conjunction 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 is called within a process that is not attached to a terminal, it returns a **NULL** pointer.

If the login name is not found, the **getlogin** subroutine returns a **NULL** pointer.

Warning: The **getlogin** subroutine returns a pointer to a static area that is overwritten by successive calls.

File

/etc/utmp

Implementation Specifics

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

Related Information

The **getgrent, getgrgid, getgrnam, setgrent, endgrent** subroutines, **getpwent, getpwuid, setpwent, endpwent** subroutines.

getopt Subroutine

Purpose

Gets flag letters from the argument vector.

Library

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

Syntax

int getopt (*ArgumentC*, *ArgumentV*, *OptionString*)
int *ArgumentC*;
char ***ArgumentV*;
char **OptionString*;

extern int optind;
extern char optopt;
extern int opterr;
extern char *optarg;

Description

The **getopt** subroutine returns the next flag letter in the *ArgumentV* parameter list that matches a letter in the *OptionString* parameter. The **getopt** subroutine is an aid to help programs interpret shell command-line flags that are passed to them.

The **optarg** external variable is set to point to the start of the flag's parameter on return from the **getopt** subroutine.

The **getopt** subroutine places the *ArgumentV* index of the next argument to be processed in **optind**. **optind** is externally initialized to 1 so that *ArgumentV*[0] is not processed.

Parameters

ArgumentC	The number of parameters passed to the routine.
ArgumentV	The list of parameters passed to the routine.
OptionString	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.

Return Values

When all flags have been processed (that is, up to the first non-flag argument), the **getopt** subroutine returns EOF. The special flag — (dash dash) can be used to delimit the end of the flags; EOF is returned, and — is skipped.

Error Codes

The **getopt** subroutine prints an error message on **stderr** and returns (int) '?' (question mark) when it encounters a flag letter that is not included in the *OptionString* parameter.

Note: The external **int optopt** variable is set to the real option found in the *ArgumentV* parameter. This is true whether the flag is in the *OptionString* parameter or not.

You can set the **int** variable **opterr** to zero to suppress the generation of error messages.

Implementation Specifics

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

Related Information

The **getopt** command.

getpagesize Subroutine

Library

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

Purpose

Gets the system page size.

Syntax

int getpagesize( )

Description

The **getpagesize** subroutine returns the number of bytes in a page. Page granularity is the granularity of many of the memory management calls.

The page size is a *System* page size and may not be the same as the underlying hardware page size.

Implementation Specifics

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

Related Information

The **brk, sbrk** subroutines.

The **pagesize** command.

getpass Subroutine

Purpose

Reads a password.

Library

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

Syntax

#include <stdio.h>

char *getpass (*Prompt*)
char **Prompt*;

Description

The **getpass** subroutine will open the controlling terminal of the current process, write the specified *Prompt* parameter to that device and read up to the value of **PASS_MAX** characters until a new line or **EOF** condition is detected. Echoing of charters is disabled during the read.

Note: The characters are returned in a static data area which will be overwritten upon subsequent calls to this routine.

Parameter

Prompt Specifies a prompt to be displayed on the terminal. If this parameter is **NULL**, the prompt **passwd:** is used. Note that an empty string is treated the same as a **NULL** string.

Return Values

If the information is successfully read, a pointer to the string is returned. If an error occurs, a **NULL** pointer is returned and **errno** is set to indicate the error.

Error Codes

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

EINTR An interrupt occurred while reading the terminal device.

ENXIO The process does not have a controlling terminal.

Other errors may be set by any subroutines invoked by this function.

Implementation Specifics

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

Related Information

The **newpass** subroutine, **getuserpw** subroutine.

getpcred Subroutine

Purpose

Reads the current process credentials

Library

Ssecurity Library (**libs.a**)

Syntax

#include <usersec.h>

char **getpcred*(Which)*
int **Which;*

Description

The **getpcred** subroutine will read the specified process security credentials and return them in a character buffer.

Parameters

Which — Specifies which credentials are to be read. This parameter is a bit mask and may contain one or more of the following values, which are defined in the **usersec.h** file:

CRED_RUID	The real user name.
CRED_LUID	The login user name.
CRED_RGID	The real group name.
CRED_GROUPS	The concurrent group set.
CRED_AUDIT	The audit class.
CRED_RLIMITS	The BSD resource limits. **Note:** Support of all the process resource limits is needed, not just the file size. Use the **getrlimit** call.
CRED_UMASK	The umask.

If the *Which* parameter is equal to **NULL**, all credentials are returned.

Return Values

Upon successful return, the **getpcred** subroutine returns a pointer to a string containing the requested values. If **getpcred** fails, a value of –1 is returned and **errno** is set to indicate the error.

Error Codes

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

EINVAL The *Which* parameter contains invalid credentials requests.

Other errors may be set by any subroutines invoked by the **getpcred** subroutine.

Implementation Specifics

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

Related Information

The **setpenv** subroutine, **getpenv** surbroutine, **setpcred** subroutine, **ckuseracct** subroutine, **ckuserID** subroutine.

getpenv Subroutine

Purpose

Reads the current process credentials

Library

Security Library (**libs.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.

Parameter

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, which are 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 is protected from unauthorized access.

All variables will be 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 user state environment is requested, the current working directory is always returned, in a variable named **PWD**. If this variable is not present in the existing environment, the **getpenv** subroutine will add it to the returned string.

Return Values

Upon successful return, the **getpenv** subroutine returns the environment values. If **getpenv** fails, a value of NULL is returned and **errno** is set to indicate the error. Note that this function can partially succeed, returning only the values that the process will permit 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 may be set by any subroutines invoked by the **getpenv** subroutine.

Implementation Specifics

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

Related Information

The **ckuseracct** subroutine, **ckuserID** subroutine, **getpcred** subroutine, **setpenv** subroutine.

getpid, getpgrp, or getppid Subroutine

Purpose

Gets the process ID, process group ID, and parent process ID.

Syntax

pid_t getpid( )

pid_t getpgrp( )

pid_t getppid( )

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 group ID of the calling process parent process.

Implementation Specifics

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

In the AIX Version 3 Operating System, the POSIX version of the **getpgrp** subroutine is implemented. The process group ID of the calling process is returned. (The BSD version allows a process ID as input and returns the process group ID of that process.)

Related Information

The **exec** subroutines, **fork** subroutine, **setpgid** subroutine, **sigaction**, **sigvec**, **signal** subroutines, **setpgrp** subroutine.

getpri Subroutine

Purpose

Returns the scheduling priority of a process.

Library

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

Syntax

int getpri (*ProcessID***)**
pid_t pid;

Description

The **getpri** subroutine returns the scheduling priority of a process.

Parameter

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 the process. Otherwise, a value of –1 is returned and the global variable **errno** is set to indicate the error.

Error Codes

The **getpri** subroutine fails if one or both of the following are 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 AIX Base Operating System (BOS) Runtime.

Related Information

The **setpri** subroutine.

getpriority, setpriority, or nice Subroutine

Purpose

Gets or sets *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*;

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 (lowest numerical value) pertaining to any of the specified processes. The **setpriority** subroutine sets the priorities 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 *Increment*.

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 global variable **errno** 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 global variable **errno** is set to indicate the error.

Note: –1 can also be returned as a valid return value; in that case the calling process should also check **errno**.

Error Codes

The **getpriority** and **setpriority** subroutines fail if one or more of the following are 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 can fail if one or both of the following are true:

EPERM A process was located, but neither the effective nor the real user ID of the caller, and neither the effective nor the real user ID 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 fails if the following is true:

EPERM The *Increment* parameter is negative or greater than 2 x {NZERO} and the calling process does not have appropriate privileges.

Implementation Specifics

These subroutines are part of AIX 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 **nice** are supported by the AIX Version 3 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 to 19, 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**) and the *Increment* parameter is treated as the modifier of a value in the range –20 to 20.

Related Information

The **exec** subroutines.

getpwent, getpwuid, getpwnam, putpwent, setpwent, or endpwent Subroutine

Purpose

Accesses the basic user information in the user data base.

Library

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

Syntax

#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

These subroutines may be used to access the basic user attributes.

The **setpwent** subroutine opens the user database (if not already open) and rewinds the cursor to point to the first user entry in the database.

The **getpwent**, **getpwnam**, and **getpwuid** subroutines return information about the requested user. The **getpwent** subroutine returns the next user entry in the sequential search, **getpwnam** returns the first user entry in the data base whose name matches the *Name* parameter and **getpwuid** returns the first user entry in the data base 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. Note that the **pw_passwd** field will be written into the corresponding field in the file. If this user's password is stored in the shadow password file, this field must be an exclamation mark '**!**'. The password in the shadow file cannot be updated with this function, the **putuserpw** subroutine should be used to update this file.

The **endpwent** subroutine will close the user data base.

The user structure, which is returned by the **getpwent**, **getpwnam** and **getpwuid** subroutines and which is written by the **putpwent** subroutine, is defined in the **pwd.h** file and has the following members:

pw_name The name of the user.

pw_passwd The encrypted password of the user. Note that if the password is not stored in the **/etc/passwd** file and the invoker does not have access to the shadow file which contains them, this field will contain an undecryptable string (usually an asterisk '*').

pw_uid The ID of the user.

pw_gid The group ID of the principle group of the user.

pw_gecos The personal information about the user.

pw_dir The home directory of the user.

pw_shell The initial program for the user.

Warning: All information generated by the **getpwent**, **getpwnam**, and **getpwuid** subroutines is stored in a static area and will be overwritten on subsequent calls to these routines. If it is to be saved, it should be copied.

Warning: These subroutines should not be used in conjunction with the **getuserattr** subroutine. The results are unpredictable.

Parameters

File Specifies an open file whose format is like that of **/etc/passwd**.

Name Specifies the name of the user for which the basic attributes are to be read.

Password Specifies the password structure which contains the user attributes which are to be written.

UserID Specifies the ID of the user for which the basic attributes are to be read.

Security

File Access The calling process must have access to the basic information in the user data base. This includes the following files:

modes	**file**	
rw	**/etc/passwd**	(write access for **putpwent** 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.

Implementation Specifics

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

Related Information

The **getgrent** subroutine, **getgroupattr** subroutine, **getuserattr** subroutine, **setuserdb** subroutine, **getuserpw, putuserpw** subroutines.

getrlimit, setrlimit, 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 getrlimit (*Resource1*, *RLP***)**
int *Resource1*;
struct rlimit **RLP*;

#include <sys/vlimit.h>

vlimit (*Resource2*, *Value***)**
int *Resource2*, *Value*;

Description

Limits on the consumption of system resources by the current process and each process it creates are obtained with the **getrlimit** system call, and set with the **setrlimit** subroutine.

A resource limit is specified as a soft limit and a hard limit. When a soft limit is exceeded a process can receive a signal (for example, if the CPU time is exceeded), but it is allowed to continue until it reaches the hard limit or modifies its resource limit. The **rlimit** structure is used to specify the hard and soft limits on a resource, as defined in the **sys/resource.h** header file.

The calling process must have root user authority in order to raise the maximum limits. Other processes can alter *rlim_cur* within the range from 0 to *rlim_max* or (irreversibly) lower *rlim_max*.

An infinite value for a limit is defined as **RLIM_INFINITY**.

Because this information is stored in the per–process information, this subroutine must be executed directly by the shell if it is to affect all future processes created by the shell; *limit* is thus a built–in command to the shells.

The system refuses to extend the data or stack space when the limits would be exceeded in the normal way: a **break** system call fails if the data space limit is reached. 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, this signal kills the process. When the soft CPU time limit is exceeded, a signal **SIGXCPU** is sent to the offending process.

The **vlimit** subroutine is also supported, but this facility is superceded by the **getrlimit** subroutine.

Parameters

Resource1 Can be one of the following values:

RLIMIT_CPU The maximum amount of CPU time (in seconds) to be used by each process.

RLIMIT_FSIZE The largest size, in bytes, of any single file that can be created.

RLIMIT_DATA The maximum size, in bytes, of the data segment for a process; this defines how far a program can extend its break with the **sbrk** subroutine.

RLIMIT_STACK The maximum size, in bytes, of the stack segment for a process; this defines how far a program stack segment can be extended. Stack extension is performed automatically by the system.

RLIMIT_CORE The largest size, in bytes, of a **core** file that can be created.

RLIMIT_RSS The maximum size, in bytes, to which a process's resident set size can grow. This imposes a limit on the amount of physical memory to be given to a process; if memory is tight, the system prefers to take memory from processes that are exceeding their declared resident set size.

RLP Points to the **rlimit** structure, which contains the current (soft) and hard limits. For the **getrlimit** subroutine, the requested limits are returned in this structure, and for the **setrlimit** subroutine, the desired new limits are specified here.

Resource2 The flags for this parameter are defined in the **sys/vlimit.h** header file, and are mapped to corresponding flags for the **setrlimit** subroutine.

Value An integer that is used as a hard limit parameter to the **setrlimit** 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 global variable **errno** is set to indicate the error.

Error Codes

The **getrlimit**, **setrlimit** or **vlimit** subroutine fails if one or more of the following are true:

EFAULT The address specified for the *RLP* parameter is invalid.

EINVAL The *Resource1* parameter is not a valid resource.

EPERM The limit specified to the **setrlimit** system call would have raised the maximum limit value, and the caller does not have root user authority.

Implementation Specifics

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

Related Information

The **sigaction**, **sigvec,signal** subroutines, **sigstack** subroutine, **ulimit** subroutine.

getrusage, times, or vtimes Subroutine

Purpose

Gets information about resource utilization.

Libraries

getrusage, **times**: Standard C Library (**libc.a**)

vtimes: Berkeley Compatibility Library (**libbsd.a**)

Syntax

#include <sys/time.h>
#include <sys/time.h>

int getrusage (*Who*, *RUsage*)
int *Who*;
struct rusage **RUsage*;

#include <sys/types.h>
#include <sys/times.h>

time_t times (*Buffer*)
struct tms **Buffer*;

#include <sys/times.h>

vtimes (*ParentVm*, *ChildVm*)
struct vtimes **ParentVm*, *ChildVm*;

Description

The **getrusage** subroutine returns information describing the resources utilized by the current process, or all its terminated child processes.

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 in tenths of a second, unless execution profiling is enabled. When profiling is enabled, the **times** subroutine reports values in 1/60 of a second.

The **tms** structure is defined in the **sys/times.h** header file, and it contains the following members:

```
time_t    tms_utime;
time_t    tms_stime;
time_t    tms_cutime;
time_t    tms_cstime;
```

This information comes from the calling process and each of its terminated child processes for which it has executed a **wait** subroutine.

tms_utime
: The CPU time used while 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_utimes* and the *tms_cutimes* of the child processes.

tms_cstime
: The sum of the *tms_stimes* and the *tms_cstimes* of 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 — **RUSAGE_SELF** or **RUSAGE_CHILDREN**.

RUsage — A pointer to a buffer that will be filled in as described in the **sys/resource.h** header file. The fields are interpreted as follows:

ru_utime	The total amount of time spent executing in user mode.
ru_stime	The total amount of time spent in the system executing on behalf of the process(es).
ru_maxrss	The maximum resident set size utilized (in kilobytes).
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 summing 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 residing 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: here 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 had to perform input.
ru_outblock	The number of times the file system had to perform output.
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 due to a process voluntarily giving up the processor before its time slice was completed (usually to await availability of a resource).

ru_nivcsw	The number of times a context switch resulted due to a higher priority process becoming runnable or because the current process exceeded its time slice.

Note: The numbers the ru_inblock and ru_outblock fields account only for real I/O; data supplied by the caching mechanism is charged only to the first process to read or write the data.

Buffer Points to a structure.

ParentVm Points to a vtimes structure that will contain the accounting information for the current process.

ChildVm Points to a vtimes structure that will contain the accounting information for the terminated child processes of the current process.

Return Values

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

Upon successful completion, the **times** subroutine returns the elapsed real time, in 1/60 of a second, since an arbitrary reference time in the past (for example, system start-up time). This reference time does not change from one call of the **times** subroutine to another.

Error Codes

The **getrusage** subroutine fails 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.

Implementation Specifics

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

The **vtimes** subroutine is supported to provide compatibility with older programs.

The **vtimes** subroutine returns accounting information for the current process and for the terminated child processes of the current process. Either *ParVm* or *ChVm* or both may be 0, in which case only the information for the pointers which are nonzero are returned.

After the call, each buffer contains information as defined by the contents of the **sys/vtimes.h** include file.

Related Information

The **gettimer**, **time** subroutines, **wait**, **waitpid**, **wait3** subroutines.

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 characters from the standard input stream, **stdin**, into the array pointed to by the *String* parameter. Data is read until a new–line character is read or an end–of–file condition is encountered. If reading is stopped due to a new–line character, the new–line character is discarded and the string is terminated with a null character.

The **fgets** subroutine reads characters from the data pointed to by the *Stream* parameter intc the array pointed to by the *String* parameter. Data is read until the value of the *Number* parameter –1 characters have been read, until a new–line character is read and transferred to *String*, or until an end–of–file condition is encountered. The string is then terminated with a null character.

Parameters

String	A pointer to a string to receive characters.
Stream	A pointer to the **FILE** structure of an open file.
Number	An upper bound on the number of characters to read.

Return Value

If the end of the file is encountered and no characters have been read, no characters are transferred to *String* and a **NULL** pointer is returned. If a read error occurs, a **NULL** pointer is returned. Otherwise, *String* is returned.

Implementation Specifics

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

Related Information

The **ferror**, **feof**, **clearerr**, **fileno** macros, **fopen**, **freopen**, **fdopen** subroutines, **fopen, freopen**, **fdopen**, subroutines, **fread** subroutine, **getc**, **fgetc**, **getchar**, **getw** subroutines, **getwc**, **fgetwc**, **getwchar** subroutines, **getws**, **fgetws** subroutines, **puts**, **fputs** subroutines, **putws**, **fputws** subroutines, **scanf**, **fscanf**, **sscanf**, **NLscanf**, **NLsscanf** subroutines.

getssys Subroutine

Purpose

Reads a subsystem record.

Library

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

Syntax

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

int getssys(*SubsystemName, SRCSubsystem***)**
char **SubsystemName*;
struct SRCsubsys **SRCSubsystem*;

Description

The **getssys** subroutine reads a subsystem record associated with the *SubsystemName* parameter and returns the ODM record in the *SRCSubsystem* parameter.

The **SRCsubsys** structure is defined in the **sys/srcobj.h** header file.

Parameters

SRCSubsystem	Points to a **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 either returns a value of –1 and **odmerrno** is set to indicate the error, or it returns **SRC_NOREC**.

Error Code

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

SRC_NOREC	Subsystem name does not exist.

Implementation Specifics

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

File

/etc/objrepos/SRCsubsys	SRC Subsystem Configuration object class.

Related Information

The **addssys** subroutine, **delssys** subroutine, **getsubsvr** subroutine.

The System Resource Controller Overview in *General Programming Concepts.*

getsubsvr Subroutine

Purpose

Reads a subsystem record.

Library

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

Syntax

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

int getsubsvr(*SubserverName*, *SRCSubserver***)**
char **SubserverName*;
struct SRCSubsvr **SRCSubserver*;

Description

The **getsubsvr** subroutine reads a subsystem record associated with the *SubserverName* parameter and returns the ODM record in the *SRCSubserver* parameter.

The **SRCsubsvr** structure is defined in the **sys/srcobj.h** header 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 either returns a value of –1 and **odmerrno** is set to indicate the error, or **SRC_NOREC** is returned.

Error Code

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

SRC_NOREC	The specified **SRCsubsvr** record does not exist.

Implementation Specifics

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

File

/etc/objrepos/SRCsubsvr	SRC Subserver Configuration object class.

Related Information

The **getssys** subroutine.

The System Resource Controller Overview in *General Programming Concepts*.

gettimeofday, settimeofday, or ftime Subroutine

Purpose

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*;
**struct timezone ****Tzp*;

int settimeofday (*Tp*, *Tzp*)
**struct timeval ****Tp*;
**struct timezone ****Tzp*;

int ftime (*Tp*)
**struct timeb ****Tp*;

Description

The system's notion of the current Greenwich time and the current time zone is obtained 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 continuously or in "ticks." If *Tzp* is zero, the time zone information will not be returned or set.

Only users with **SEC_SYS_ATTR** system privilege may change the date and time.

The *Tp* parameter returns a pointer to a **timeval** structure which contains the time since the epoch began in seconds and microseconds.

The **timezone** structure indicates 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.

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

The possible errors are:

EFAULT A parameter points to an invalid address.

EPERM The process's effective user ID does not have root user authority.

Implementation Specifics

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

The **gettimeofday** and **settimeofday** subroutines are supported for compatibility with BSD programs.

The **ftime** subroutine is included for compatibility with older BSD programs. It's function has been obsoleted by the **gettimeofday** subroutine.

Related Information

The **ctime**, **localtime**, **gmtime**, **mktime**, **difftime**, **asctime**, **tzset**, **timezone** subroutines.

The **gettimer** subroutine, **adjtime** subroutine.

The **date** command.

gettimer, settimer, restimer, stime, or time Subroutine

Purpose

Gets or sets the current value for the specified system–wide timer.

Library

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

Syntax

#include <sys/time.h>

int gettimer(*Timer_type*, *TimePointer***)**
int *Timer_type*;
timestruc_t **Time**Pointer*;

int settimer(*Timer_type*, *TimePointer***)**
int *Timer_type*;
timestruc_t **Tp*;

int restimer(*Timer_type*, *Resolution*, *MaximumValue***)**
int *Timer_type*;
timestruc_t **Resolution*, **MaximumValue*;

int stime(*Tp***)**
long *Tp*;

<include time.h>
time_t time(*Tp***)**
time_t **Tp*;

Description

The **settimer** subroutine is used to set the current value of the *Tp* parameter for the system–wide timer, specified by the *Timer_type* parameter. The **gettimer** subroutine is used to get the current value of the *Tp* parameter for the system–wide timer, specified by the *Timer_type* parameter. The *Tp* parameter points to a structure of type **timestruc_t**, which includes the following members:

```
unsigned long tv_sec;     /* seconds */
  long          tv_nsec;  /* nano–seconds */
```

The **tv_nsec** member is only valid if greater than or equal to zero, and less than the number of nanoseconds in a second (1000 million).

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 non–privileged user attempts to submit a fine granularity timer (i.e., a timer request less than 10 milliseconds), the timer request is raised to 10 milliseconds.

Parameters

Timer_type Specifies the system–wide 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 **timestruc_t**.

Resolution The resolution of a specified timer.

MaximumValue The maximum possible timer value.

Tp Time in seconds.

Compatibility Interface

The **stime** and **time** subroutines are implemented to provide compatibility with older AIX, AT&T System V, and BSD systems. They are implemented to simply call the **settimer** and **gettimer** subroutines using the **TIMEOFDAY** timer.

Return Values

The **gettimer**, **settimer**, **restimer**, and **stime** subroutines return a 0 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, (i.e., 00:00:00 GMT, January 1, 1970).

Error Codes

If an error occurs a return value of –1 is received and **errno** is set to one of the following error codes:

EINVAL The *Timer_type* parameter does not specify a known system–wide timer. The *Tp* parameter of the **settimer** subroutine is outside the range for the specified system–wide timer.

EIO An error occurred while accessing the timer device.

EPERM The requesting process does not have the appropriate privilege to set the specified timer.

Implementation Specifics

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

Related Information

The **getinterval** subroutine, **ctime** subroutine.

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(*Timer_type*, *Notify_type***)**
int *Timer_type*;
int *Notify_type*;

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 *Timer_type* parameter, is defined in the **sys/time.h** file, and can identify either a system–wide 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 *Notify_type* parameter, which is defined in the **sys/events.h** file.

The *Timer_type* parameter represents one of the following timer types supported under AIX Version 3:

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 **SIG_ALRM** signal at a timer specified in the call to the **settimer** subroutine.

TIMER_REAL (Real time timer) The real time timer decrements in real time. A **SIG_ALRM** signal is delivered when this timer expires.

TIMER_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.

TIMER_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.

The system shall cause a **SIGALRM** signal to be sent to the process whenever the interval timer expires.

Interval timers are not inherited by a child process across a **fork** subroutine, or across an **exec** subroutine, if the notification mechanism is **DELIVERY_EVENTS**. Interval timers with a notification value of **DELIVER_SIGNALS** are inherited across an **exec** subroutine.

Parameters

Notify_type Notifies the process of the expiration of the timer event.

Timer_type Identifies either a system–wide timer or a per–process timer.

Return Values

If the **gettimerid** subroutine succeeds, it returns a **timer_t** structure which 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

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

Related Information

The **exec** subroutine, **fork** subroutine, **gettimer, settimer, restimer** subroutines, **getinterval, incinterval, absinterval, resabs, resinc** subroutines, **reltimerid** subroutine.

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

The **getttyent** and **getttynam** subroutines each return a pointer to an object with the **ttyent** structure, containing the broken-out fields of a line from the tty description file. The **ttyent** structure is in the **ttyent.h** header file and contains the following fields:

tty_name — The name of the character-special file in the directory "/**dev**". For various reasons, it must reside in the directory "/**dev**".

ty_getty — The command (usually **getty**) which is called by **init** to initialize tty line characteristics. In fact, 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 of the default terminal type connected to this tty line. This is typically a name from the termcap data base. The environment variable **TERM** is initialized with this name by **getty** or **login**.

ty_status — A mask of bit fields which indicate various actions to be allowed on this tty line. The following is a description of each flag.

- **TTY_ON** — Enables logins (i.e., **init** will start the specified **getty** command on this entry).
- **TTY_SECURE** — Allows root user to login on this terminal. Note that **TTY_ON** must be included for this to be useful.

ty_window — The command to execute for a window system associated with the line. The window system will be started before the command specified in the ty_getty entry is executed. If none is specified, this will be null.

ty_comment — The trailing comment field, if any; a leading delimiter and white space will be removed.

Note: The **getttyent** and **getttynam** subroutines require links to /**lib/libodm.a** and /**usr/lib/libcfg.a**.

The **getttyent** subroutine reads the next line from the tty file, opening the file if necessary; the **settyent** subroutine rewinds the file; the **endttyent** subroutine closes it.

The **getttyent** subroutine searches from the beginning of the file until a matching *Name* is found (or until the EOF is encountered).

Parameter

Name Specifies the name of a tty description file.

Return Value

Null pointer (0) returned on EOF or error.

Implementation Specifics

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

Related Information

The **ttyslot** subroutine.

The **init** command, **getty** command, **login** command.

getuid or geteuid Subroutine

Purpose

Gets the process's real or effective user ID.

Library

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

Syntax

#iclude <sys.types.h>

uid_t getuid( )

uid_t geteuid( )

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.

Implementation Specifics

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

Related Information

The **getuidx** subroutine, **setuid** subroutine, **setuidx** subroutine.

getuidx Subroutine

Purpose

Gets the process user IDs.

Library

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

Syntax

#include <sys/id.h>

uid_t getuidx (*Which***)**
int *Which***;**

Description

The **getuidx** subroutine returns the specified user ID of the current process.

Parameter

Which Specifies which user ID to return. The valid values for this parameter are defined in **sys/id.h** and include:

ID_EFFECTIVE Returns the effective user ID of the process.

ID_REAL Returns the real user ID of the process.

ID_SAVED Returns the saved user ID of the process.

ID_LOGIN Returns the login user ID of the process.

Return Values

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

Error Code

The **getuidx** subroutine fails if:

EINVAL The *Which* parameter is not one of **ID_EFFECTIVE**, **ID_REAL**, **ID_SAVED**, or **ID_LOGIN**.

Implementation Specifics

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

Related Information

The **setuid** subroutine, **setuidx** subroutine, **getuid** subroutine.

getuinfo Subroutine

Purpose

Finds the value associated with a user information name.

Library

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

Syntax

char *getuinfo (*Name***)**
**char ***Name*;

Description

The **getuinfo** subroutine searches a user information buffer for a string of the form *Name=value* and returns a pointer to the *value* substring if *Name* is found. **NULL** is returned if *Name* is not found.

The user information buffer searched is pointed to by the global variable:

```
extern char *INuibp;
```

This variable is initialized to **NULL**.

If the **INuibp** global variable is **NULL** when the **getuinfo** subroutine is called, the **usrinfo** subroutine is run to read user information from the kernel into a local buffer. The address of the buffer is then put into the **INuibp** external variable. The **usrinfo** subroutine is automatically called the first time the **getuinfo** subroutine is called if the **INuibp** external variable is not set.

Parameter

Name A user information name.

Implementation Specifics

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

Related Information

The **usrinfo** subroutine.

getuserattr, IDtouser, nextuser, or putuserattr Subroutine

Purpose

Accesses the user information in the user data base.

Library

Security Library (**libs.a**)

Syntax

#include <usersec.h>

int getuserattr (*User, Attribute, Value, Type)*
char **User;*
char **Attribute;*
void **Value;*
int *Type;*

int putuserattr (*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*;

Description

These subroutines may be used to access user information. Because of their greater granularity and extensibility these routines should be used instead of the **getpwent** routines.

The **getuserattr** subroutine reads a specified attribute from the user data base. If the data base is not already open, the **getuserattr** subroutine will do an implicit open for reading.

The **putuserattr** subroutine writes a specified attribute into the user data base. If the data base is not already open, the **putuserattr** subroutine will do an implicit open for reading and writing. The data changed by **putuserattr** must be explicitly committed by calling **putuserattr** with a *Type* parameter equal to **SEC_COMMIT**. Until all the data is committed, only these subroutines within the process will return the written data.

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 data base. The consistency of consecutive searches depends upon the underlying storage access mechanism and is not guaranteed by this function.

Values which are returned by these functions are in dynamically allocated buffers and need not be moved prior to the next call.

The **setuserdb** and **enduserdb** subroutines should be used to open and close the user data base.

Note: These subroutines and the **setpwent** and **setgrent** subroutines should not be used simultaneously. The results can be unpredictable.

Parameters

Argument The *Argument* parameter is presently unused and must be specified as NULL.

Attribute Specifies the name of the attribute which is to be read. This can be one of the following, which are defined in the **usersec.h** file:

S_ID The user ID. Type: **SEC_INT**.

S_PGRP The principle group name. Type: **SEC_CHAR**.

S_GROUPS The groups to which the user belongs, other than the principle group. Type: **SEC_LIST**.

S_ADMGROUPS The groups for which the user is an administrator. Type: **SEC_LIST**.

S_ADMIN Defines the administrative status of a user. Type: **SEC_BOOL**.

S_AUDITCLASSES Defines the audit classes to which the user belongs. Type: **SEC_LIST**.

S_HOME Defines the home directory. Type: **SEC_CHAR**.

S_SHELL Defines the initial program run by a user. Type: **SEC_CHAR**.

S_GECOS Defines the personal information for a user. Type: **SEC_CHAR**.

S_USRENV Defines the user–state environment variables. Type: **SEC_LIST**.

S_SYSENV Defines the protected–state environment variables. Type: **SEC_LIST**.

S_LOGINCHK Defines if the user account can be used for local logins. Type: **SEC_BOOL**.

S_SUCHK Defines if the user account can be accessed with the **su** command. Type **SEC_BOOL**.

S_RLOGINCHK Defines if the user account can be used for remote logins via **telnet** or **rlogin**. Type: **SEC_BOOL**.

S_DAEMONCHK Defines if the user account can be used for daemon execution of programs and subsystems via **cron** or **src**.

S_TPATH Defines how the account may be used on the Trusted Path. Type: **SEC_CHAR**. This attribute must be one of the following:

- **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 Defines a list of ttys which may or may not be used to access this account. Type: **SEC_LIST**.

S_SUGROUPS Defines the groups which may or may not be permitted to access this account. Type: **SEC_LIST**.

S_EXPIRATION Defines the expiration date for this account, in seconds since the epoch. Type: **SEC_CHAR**.

S_AUTH1 Defines the primary authentication methods for this account. Type: **SEC_LIST**.

S_AUTH2 Defines the secondary authentication methods for this account. Type: **SEC_LIST**.

S_UFSIZE Defines the process file size limit. Type: **SEC_INT**.

S_UCPU Defines the process CPU time limit. Type: **SEC_INT**.

S_UDATA Defines the process data segment size limit. Type: **SEC_INT**.

S_USTACK Defines the process stack segment size limit. Type: **SEC_INT**.

S_URSS Defines the process real memory size limit. Type: **SEC_INT**.

S_UCORE Defines the process core file size limit. Type: **SEC_INT**.

S_PWD Defines the passwd field in the **/etc/passwd** file.

S_UMASK Defines the file creation mask for a user. Type: **SEC_INT**.

Note: These values are string constants which should be used by applications both for convenience and to permit optimization in latter implementations.

Mode Specifies the search mode. This parameter can be used to delimit the search to one or more user credentials data bases. Specifying a non_NULL *Mode* also implicitly rewinds the search. A NULL mode should be used to continue the search sequentially through the data base. This attribute may include one or more of the following values specified as a bin mask; these are defined in the **usersec.h** file:

S_LOCAL Locally defined users will be 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. The buffer returned by the **getuserattr** subroutine and the buffer supplied by the **putuserattr** 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 list of NULL terminated character strings. The list itself is NULL terminated.

SEC_BOOL The format of the attribute is a boolean.

SEC_COMMIT For the **putuserattr** subroutine, this value specified by itself indicates that changes to the named group are to be commited to permanent storage. The *Attribute* and *Value* parameters are ignored. If no group is specified, the changes to all modified groups will be committed.

SEC_DELETE The corresponding attribute will be deleted from the data base.

SEC_NEW Updates all the group data base 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 the address of a buffer in which the attribute is to be stored (**getuserattr**) or is stored (**putuserattr**).

Security

File Access The calling process must have access to the account information in the user data base and the authentication data. This includes:

modes	**file**
rw	/etc/passwd
rw	/etc/group

rw	/etc/security/user
rw	/etc/security/limits
rw	/etc/security/audit/audit.config
rw	/etc/security/group
rw	/etc/security/environ

Return Values

The **getuserattr** and **putuserattr** subroutines return 0 if completed successfully. Otherwise, a value of −1 is returned and **errno** is set to indicate the error.

The **IDtouser** and **nextuser** subroutines return a character pointer to a buffer containing the requested group name if successful. Otherwise a **NULL** pointer is returned and **errno** is set to indicate the error.

Error Codes

These subroutines fail if the following is true:

EACCES Access permission is denied for the data request.

The **getuserattr** and **putuserattr** subroutines fail if one or more of the following is true:

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.

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

ENOATTR The specified user attribute does not exist for this user.

ENOENT The *Uid* parameter could not be translated into a valid user name on the system.

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

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 AIX Base Operating System (BOS) Runtime.

Related Information

The **getgroupattr** subroutine, **getuserpw** subroutine, **setuserdb** subroutine, **setpwdb** subroutine.

getuserpw or putuserpw Subroutine

Purpose

Accesses the user authentication data.

Library

Security Library (**libs.a**)

Syntax

#include <userpw.h>

struct userpw *getuserpw(*User***)**
**char ***User*;

int putuserpw(*Password***)**
**struct userpw ***Password*;

Description

These subroutines may be used to access user authentication information. Because of their greater granularity and extensibility these should be used instead of the **getpwent** routines.

The **getuserpw** subroutine reads the user's locally–defined password information.

The **putuserpw** subroutine updates or creates a locally defined password information stanza in the /**etc/security/passwd** file.

Parameters

Password — Specifies the password structure which is to be used to update the password information for this user. This structure is defined in **userpw.h** and contains the following members:

upw_name — Specifies the user's name.

upw_passwd — Specifies the user's password.

upw_lastupdate — Specifies the time (in seconds since the Epoch) when the password was last updated.

upw_flags — Specifies attributes of the password. This member is a bitmask 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 will need to 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 user or by the **root** user.

User Specifies the name of the user for which password information is to be read.

Security

File Access The calling process must have access to the user authentication data in the user data base. This includes:

modes	**file**
rw	/etc/security/passwd

Return Values

The **getuserpw** subroutine returns a valid pointer to a **pw** structure if successfully completed. Otherwise, a **NULL** pointer is returned and **errno** is set to indicate the error.

Error Codes

The **getuserpw** and **putuserpw** subroutines fail if the following are true:

ENOENT The user does not have an entry in the **/etc/security/passwd** file.

ENAMETOOLONG
The user name is greater than the **PW_NAMELEN** value in characters

Other errors may be set by any subroutines invoked by **getuserpw** or **putuserpw**.

Implementation Specifics

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

Related Information

The **getuserattr** subroutine, **getgroupattr** subroutine, **setuserdb** subroutine, **setpwdb, endpwdb** subroutines.

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/inittabid  */
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;               /* The 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 already 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. If you have not searched for the proper place in the file using one of the **getut** routines, the **pututline** subroutine calls **getutid** to search forward for the proper place. It is expected that the user of **pututline** searched for the proper entry using one of the **getut** subroutines. 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. You should do this before each search for a new entry if you want to examine the entire file.

The **endutent** subroutine closes the currently open file.

The **utmpname** subroutine changes the name of the file to be examined from /**etc**/**utmp** to any other file. The name specified is usually /**usr/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 it is currently open, and saves the new file name.

The most current entry is saved in a static structure. If you want 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, but 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 type **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 one of the types **INIT_PROCESS**, **LOGIN_PROCESS**, **USER_PROCESS** or **DEAD_PROCESS** in the *Id* parameter, then 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 the type **LOGIN_PROCESS** or **USER_PROCESS** that also has a *ut_line* string matching the *Line–>ut_line* parameter string. If the end the 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 Value

These subroutines fail and return a **NULL** pointer if a read or write fails due to the end of the file or a permission conflict.

Files

/etc/utmp The path to the **utmp** file, which contains a record of users logged into the system.

/usr/adm/wtmp The path to the **wtmp** file, which contains accounting information about users logged in.

Implementation Specifics

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

Related Information

The **ttyslot** subroutine.

The **utmp, wtmp, .ilog** files.

getvfsent, getvfsbytype, getvfsbyname, getvfsbyflag, setvfsent, or endvfsent Subroutine

Purpose

Gets a **vfs** file entry.

Library

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

Syntax

#include <vfs.h>
#include <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

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 callls can be used to search the entire file.

The **vfs_ent** structure is defined in the **vfs.h** header file, and it contains the following members:

```
char vfsent_name;
int   vfsent_type;
int   vfsent_flags;
char *vfsent_mnt_hlpr;
char *vfsent_fs_hlpr;
char *vfsent_vinfop;
```

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 characters of the name searched for are the ASCII equivalent character.

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 to those 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.

Warning: All information is contained in a static area, so it must be copied if it is to be saved.

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 condition or an error is encountered on reading, a **NULL** pointer is returned.

Implementation Specifics

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

Related Information

The **getfsent**, **getfsspec**, **getfstype**, **getsfile**, **setfsent**, **endfsent** subroutines.

The National Language Support Overview in *General Programming Concepts*.

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>

int getwc (*Stream***)**
**FILE ***Stream*;

int fgetwc (*Stream***)**
**FILE ***Stream*;

int getwchar ( )

Description

The **getwc** subroutine gets the next 1–byte or 2–byte character from the input stream specified by the *Stream* parameter, and returns an **wchar_t** data type as an integer. The **fgetwc** subroutine performs the same function as **getwc**.

The **getwchar** subroutine gets the next 1–byte or 2–byte character from the standard input stream and returns an **wchar_t** as an integer.

Parameter

Stream Input data.

Return Values

These subroutines and macros return the integer constant **EOF** at the end of the file or upon an error.

Implementation Specifics

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

Related Information

The **fopen, freopen, fdopen** subroutines, **fread, fwrite** subroutines, **getc**, **fgetc**, **getchar**, **getw** subroutines, **gets, fgets** subroutines, **putwc, putwchar, fputwc** subroutines, **scanf, fscanf, sscanf, NLscanf, NLfscanf, wsscanf** subroutines.

National Language Support Overview in *General Programming Concepts.*

getwd Subroutine

Purpose

Gets current directory path name.

Library

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

Syntax

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** definition, as specified in the **limits.h** file.

Parameter

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 value of 0 and places a message in the *PathName* parameter.

Implementation Specifics

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

Related Information

The **getcwd** subroutine.

getws or fgetws Subroutine

Purpose

Gets a string from a stream.

Library

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

Japanese Language Support Syntax

When running AIX with Japanese Language Support on your system, the following subroutines stored in **libc.a**, are provided:

#include <stdio.h>
#include <NLchar.h>
NLchar *getws (*String***)**
NLchar **String*;

NLchar *fgetws (*String*, *Number*, *Stream***)**
NLchar **String*;
int *Number*;
FILE **Stream*;

Description

Japanese Language Support Information

The **getws** subroutine expands 1–byte and 2–byte character input values to uniform **NLchar** (2–byte) width. With this exception, **getws** functions exactly like the **gets** subroutine.

The **fgetws** subroutine also expands 1–byte and 2–byte character input values to uniform **NLchar** (2–byte) width. Again, with this exception, **fgetws** works just like **fgets**.

Parameters

String	A pointer to a string to receive characters.
Stream	A pointer to the **FILE** structure of an open file.
Number	An upper bound on the number of characters to read.

Return Value

If the end of the file is encountered and no characters have been read, no characters are transferred to the *String* parameter and a **NULL** pointer is returned. If a read error occurs, a **NULL** pointer is returned. Otherwise, *String* is returned.

Implementation Specifics

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

Related Information

The **ferror**, **feof**, **clearerr**, **fileno** macros, **fopen**, **freopen**, **fdopen** subroutines, **fread** subroutine, **getc**, **fgetc** , **getchar, getw** subroutines, **getwc, fgetwc, getwchar** subroutines, **gets, fgets** subroutines, **puts, fputs** subroutines, **putws, fputws** subroutines, **scanf**, **fscanf**, **sscanf**, **NLscanf, NLsscanf** subroutines.

National Language Support Overview in *General Programming Concepts*

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***)**
unsigned int *NumberOfElements*;
void hdestroy ( )

Description

The **hsearch** subroutine is a hash table search routine. It returns a pointer into a hash table that indicates the location of a given entry. 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 **hdestroy** subroutine deletes the hash table. This allows you to start a new hash table since only one table can be active at a time.

Parameters

Item Identifies a structure of the type ENTRY as defined in the **search.h** header file. It contains two pointers:

Item.key Points to the comparison key.

Item.data Points to any other data associated with that key.

Pointers to types other than **char** should be cast to pointer-to-character.

Action Specifies a value of the *Action* enumeration type that indicates what is to be done with an entry if it cannot be found in the table:

ENTER Enters the Item into the table at the appropriate point. If the table is full, a NULL pointer is returned.

FIND Does not enter the Item into the table, but returns a NULL pointer if the Item cannot be found.

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

Upon successful completion, the **hcreate** subroutine returns a value of 1.

Error Code

The **hcreate** subroutine returns a value of 0 if it cannot allocate sufficient space for the table.

Implementation Specifics

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

Related Information

The **bsearch** subroutine, **lsearch** subroutine, **tsearch** subroutine.

hypot or cabs Subroutine

Purpose

Computes the Euclidean distance function and complex absolute value.

Library

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

Syntax

#include <math.h>

double hypot (*x*, *y*)
double *x*, *y*;

double cabs (*z*)
struct {**double** *x*, *y*;} *z*;

Description

The **hypot** subroutine and **cabs** subroutine compute **sqrt** (x**2 + y**2) in such a way that underflow will not occur, and overflow occurs only if the final result warrants it.

The **cabs** subroutine is commonly referred to as computing the complex absolute value.

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 ($z = xi + yj$).

Error Codes

When using **libm.a** (**–lm**):

If the correct value overflows, the **hypot** subroutine returns HUGE_VAL.

Note: **hypot** (INF, value) = **hypot** (value, INF) = +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 **libmsaa.a** (**–lmsaa**).

Implementation Specifics

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

Related Information

The **sqrt** subroutine, **matherr** subroutine.

IMAIXMapping Subroutine

Purpose

Translates a pair of *KeySymbol* and *State* to a string and returns a pointer to this string.

Library

Input Method Library (**libIM.a**)

Syntax

caddr_t IMAIXMapping (*IMMap, KeySymbol, State, NBytes***)**
IMMap *Immap*;
KeySym *KeySymbol*;
uint *State*;
int **NBytes*;

Description

The **IMAIXMapping** subroutine translates a pair of *KeySymbol* and *State* to a string and returns a pointer to this string.

This function handles the diacritic character sequence and ALT NumPad sequence

Parameters

IMMap	Identifies the keymap
KeySymbol	Key symbol to which the string is mapped.
State	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 postive value, the **IMAIXMapping** subroutine returns a pointer to the returning string. Note that the returning string is not null terminated.

Implementation Specifics

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

Related Information

The **IMInitializeKeymap** subroutine, **IMFreeKeymap** subroutine, **IMSimpleMapping** subroutine.

AIX Input Method Overview in *General Programming Concepts*.

IMAuxCreate Subroutine

Purpose

Callback function that 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 if it wants to create an Auxiliary area. The Auxiliary area may contain several different forms of data and is not restricted by the interface.

Most Input Methods will only have a single Auxiliary area displayed 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	An application datum specified in the parameter of the **IMCreate** subroutine.

Return Values

On successful return of the **IMAuxCreate** subroutine, an id of the created Auxiliary area is set to the *AuxiliaryID* parameter and **IMNoError** is returned. Otherwise, **IMNoError** is returned.

Implementation Specifics

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

Related Information

The **IMCreate** subroutine.

AIX Input Method Overview in *General Programming Concepts.*

IMAuxDestroy Subroutine

Purpose

Callback function that notifies the Callback API to destroy any knowledge of all the Auxiliary areas.

Syntax

int IMAuxDestroy(*IM*, *AuxiliaryID*, *UData***)**
IMObject *IM*;
caddr_t *AuxiliaryID*;
caddr_t *UData*;

Description

The **IMAuxDestroy** subroutine is called by the Input Method when the auxiliary area should be destroyed.

Parameters

IM	Indicates the Input Method instance.
AuxiliaryID	Identifes the Auxiliary area to be destroyed.
UData	An application datum specified in the parameter of the **IMCreate** subroutine.

Return Values

If an error happens, the **IMAuxDestroy** subroutine returns **IMError**. Otherwise, **IMNoError** is returned.

Implementation Specifics

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

Related Information

The **IMCreate** subroutine.

AIX Input Method Overview in *General Programming Concepts*.

IMAuxDraw Subroutine

Purpose

Callback function that 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 when the Auxiliary area should be drawn. The Auxiliary area should also be created if it has not previously been done.

Parameters

IM	Indicates the Input Method instance.
AuxiliaryID	Identifies the auxiliary area.
AuxiliaryInformation	Points to the **IMAUXINFO** structure.
UData	Application datum specified in the parameter of the **IMCreate** subroutine.

Return Values

If an error happens, the **IMAuxDraw** subroutine returns **IMError**. Otherwise, **IMNoError** is returned.

Implementation Specifics

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

Related Information

The **IMAuxCreate** subroutine.

AIX Input Method Overview in *General Programming Concepts.*

IMAuxHide Subroutine

Purpose

Callback function that tells the application program to hide the 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 when the Auxiliary area should be hidden.

Parameters

IM	Indicates the Input Method instance.
AuxiliaryID	Identifies the Auxiliary area to be hidden.
UData	An application datum specified in the parameter of the **IMCreate** subroutine.

Return Values

If an error occurs, the **IMAuxHide** subroutine returns **IMError**. Otherwise, **IMNoError** is returned.

Implementation Specifics

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

Related Information

The **IMAuxCreate** subroutine.

AIX Input Method Overview in *General Programming Concepts.*

IMBeep Subroutine

Purpose

Callback function that 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 and the value –100 means no beep.
UData	An application datum specified by the parameter to the **IMCreate** subroutine.

Return Values

If an error happens, the **IMBeep** subroutine returns **IMError**. Otherwise, **IMNoError** is returned.

Implementation Specifics

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

Related Information

The **IMCreate** subroutine.

AIX Input Method Overview in *General Programming Concepts.*

imcalloc Subroutine

Purpose

Allocates space for an array.

Library

Input Method Library (**libIM.a**)

Syntax

caddr_t imcalloc(*NumberOfElements***,** *ElementSize***)**
uint *NumberOfElements***,** *ElementSize***;**

Description

The **imcalloc** subroutine allocates space for an array with the number of elements specified by the *NumberOfElements* parameter. Each element is of the size specified by the *ElementSize* parameter. The space is initialized to 0's.

Parameters

NumberOfElements	Specifies the number of elements in the array.
ElementSize	Specifies the size of each element.

Return Values

If an error happens during the **imcalloc** subroutine the **abort** subroutine is called.

Implementation Specifics

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

Related Information

The **abort** subroutine, **imfree** subroutine, **immalloc** subroutine, **imrealloc** subroutine.

AIX Input Method Overview in *General Programming Concepts*.

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. All Input Method instances, previously created, must be destroyed using the **IMDestroy** subroutine before calling the **IMClose** subroutine or memory will not be cleared.

Parameters

Imfep	Specifies the Input Method.

Implementation Specifics

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

Related Information

The **IMDestroy** subroutine.

AIX Input Method Overview in *General Programming Concepts.*

IMCreate Subroutine

Purpose

Creates one instance of an IMObject 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	A pointer to the caller supplied **IMCallback** structure.
UData	The optional *UData* parameter may be used to pass an application own information to the CALLBACK functions. Using this, the application can avoid the external references from the CALLBACK functions. The Input Method never changes this parameter, it merely passes it to the CALLBACK functions. The *UData* parameter is usually a pointer to the application data structure which may contain 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 fails, **NULL** is returned and the global variable **imerrno** is set to indicate the error.

Implementation Specifics

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

Related Information

The **IMProcess** subroutine, **IMDestroy** subroutine.

AIX Input Method Overview in *General Programming Concepts.*

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 AIX Base Operating System (BOS) Runtime.

Related Information

The **IMCreate** subroutine, **IMClose** subroutine.

AIX Input Method Overview in *General Programming Concepts.*

imfree Subroutine

Purpose

Frees a block of memory.

Library

Input Method Library (**libIM.a**)

Syntax

void imfree(*Pointer***)**
caddr_t *Pointer*;

Description

The **imfree** subroutine frees the block of memory pointed to by the *Pointer* parameter.

Parameter

Pointer	Points to a block of memory. The block pointed to by the *Pointer* parameter must have been previously allocated by the **imcalloc** subroutine.

Implementation Specifics

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

Related Information

The **imcalloc** subroutine, **immalloc** subroutine, **imrealloc** subroutine.

AIX Input Method Overview in *General Programming Concepts*.

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.

Parameter

IMMap	Identifies the keymap.

Implementation Specifics

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

Related Information

The **IMInitializeKeymap** subroutine.

AIX Input Method Overview in *General Programming Concepts.*

IMIndicatorDraw Subroutne

Purpose

Callback function that tells the application program to draw the indicator.

Syntax

int IMIndicatorDraw(*IM*, *IndicatorInformation*, *UData***)**
IMObject *IM*;
IMIndicatorInfo **IndicatorInformation*;
caddr_t *UData*;

Description

The **IMIndicatorDraw** subroutne is called by the Input Method when the value of the indicator is changed.

Parameters

IM Indicates the Input Method instance.

IndicatorInformation
Points to the **IMIndicatorInfo** structure that hold the current value of the Indicator. However, the interpretation of this value varies among (phonic) languages. The Input Method provides a function to interpret this value.

UData An application datum specified by the parameter to the **IMCreate** subroutine.

Return Values

If an error happens, the **IMIndicatorDraw** subroutne returns **IMError**. Otherwise, **IMNoError** is returned.

Implementation Specifics

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

Related Information

The **IMIndicatorHide** subroutne.

AIX Input Method Overview in *General Programming Concepts*.

IMIndicatorHide Subroutine

Purpose

Callback function that 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 when the Indicator should be hidden.

Parameters

IM	Indicates the Input Method instance.
UData	An application datum specified by the parameter to the **IMCreate** subroutine.

Return Values

If an error happens, the **IMIndicatorHide** subroutine returns **IMError**. Otherwise, **IMNoError** is returned.

Implementation Specifics

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

Related Information

The **IMIndicatorDraw** subroutine.

AIX Input Method Overview in *General Programming Concepts.*

IMInitialize Subroutine

Purpose

Initializes the Input Method for a particular language.

Library

Input Method Library (**libIM.a**)

Syntax

IMFep IMInitialize(*Language***)**
IMLanguage *Language*;

Description

The **IMInitialize** subroutine initializes the Input Method for a particular language. Each Input Method can produce one or more Input Method instances, which are created by calling the **IMCreate** subroutine.

Parameters

Language Specifies the language to be used. Each Input Method is dynamically linked to the application program.

Return Values

If **IMInitialize** succeeds, it returns a handle of type **IMFep**. Otherwise, **NULL** is returned and the global variable **imerrno** is set to indicate the error.

Implementation Specifics

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

Related Information

The **IMCreate** subroutine.

AIX Input Method Overview in *General Programming Concepts.*

IMInitializeKeymap Subroutine

Purpose

Initializes the keymap associated to the specified language.

Library

Input Method Library (**libIM.a**)

Syntax

IMMap IMInitalizeKeymap(*Language***)**
IMLanguage *Language***;**

Description

The **IMInitializeKeymap** subroutine initializes the keymap associated to the specified language. The Keyboard Mapping Table defines the keymap searching order.

Parameter

Language	Specifies the language to be used.

Return Values

The **IMInitializeKeymap** subroutine returns an identifier of type **IMMap**. Returning **NULL** means the occurrence of an error. **IMMap** is type defined in the **im.h** as **caddr_t**. This identifier is used for keymap manipulation functions.

Implementation Specifics

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

Related Information

The **IMQueryLanguage** subroutine, **IMFreeKeymap** subroutine.

AIX Input Method Overview in *General Programming Concepts.*

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, the **IMIoctl** subroutine can be used to control unique function of each language Input Method.

Parameters

IM Specifies the Input Method instance.

Operation Specifies the operation.

Argument The use of this parameter depends on the particular operation performed.

The following operations are defined across languages.

IM_Refresh Refresh the text area, Auxiliary area and Indicator by calling the needed callback functions if these area's contents are not empty. The *Argument* parameter is not used.

IM_GetString The application can use this operation to get the current pre-editing string. The *Argument* parameter is an 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 area and the Auxiliary area if they exist. If the *Argument* parameter is not NULL, this operation will invoke the callback functions to clear the screen.

IM_Reset Clears the Auxiliary area if it currently exists. If the *Argument* parameter is NULL, it clears only the Input Method's internal buffer, otherwise, the required callback functions are invoked.

IM_ChangeLength
Used to change the maximum length of the pre–editing string.

IM_QueryState This operation returns the status of the text area, the Auxiliary area and the Indicator. It also returns 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 the 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 the 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 the detailed information about the Indicator. The results are stored in the caller supplied **IMQueryIndicator** structure pointed to by the *Argument* parameter.

IM_QueryIndicatorString
Returns the Indicator string corresponding to the current indicator. Results are stored into the caller supplied **IMQueryIndicatorString** structure pointed to by the *Argument* parameter. The caller can request either short form or long form by specifying in the format member of the **IMQueryIndicatorString** structure.

Return Values

The **IMIoctl** subroutine returns **IMError** if the error happens. In this case, the global variable **imerror** is set to indicate the error.

Implementation Specifics

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

Related Information

The **IMProcess** subroutine.

AIX Input Method Overview in *General Programming Concepts.*

immalloc Subroutine

Purpose

Returns a pointer to a block of memory of at least the number of bytes specified by the *Size* parameter.

Library

Input Method Library (**libIM.a**)

Syntax

caddr_t immalloc(*Size***)**
uint *Size***;**

Description

The **immalloc** 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.

Parameter

Size	Specifies the size, in bytes, of the memory block.

Return Values

If an error happens during the **immalloc** subroutine, the subroutine calls the **abort** subroutine.

Implementation Specifics

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

Related Information

The **imcalloc** subroutine, **imfree** subroutine, **imrealloc** subroutine.

The **abort** subroutine.

AIX Input Method Overview in *General Programming Concepts.*

IMProcess Subroutine

Purpose

Processes keyboard events and does the language specific input processing.

Library

Input Method Library (**libIM.a**)

Syntax

int IMProcess (*IM, KeySymbol, State, String, Length***)**
IMObject *IM***;**
KeySym *KeySymbol***;**
uint *State***;**
caddr_t **String***;**
uint **Length***;**

Description

This is a main entry points to the Input Method.

The **IMProcess** subroutine processes one keyboard event at a time.

Processing of the **IMProcess** subroutine may look like the following:

1. Validates the *IM* parameter.
2. Keyboard translation for all its supported modifier states.
3. Invokes internal function to do language dependent processing.
4. Performs any necessary Callback functions depending on the internal state.
5. 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 State of the keyboard.

String Holds the returned string. Returning **NULL** 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 of the *String* parameter in bytes.

Return Values

The return code for the **IMProcess** subroutine has one of the following meanings:

IMError Error caused 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.

This function returns **IMError** if the error happens. In this case, the global variable **imerrno** is set to indicate the error.

Implementation Specifics

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

Related Information

The **IMCreate** subroutine, **IMClose** subroutine.

AIX Input Method Overview in *General Programming Concepts*.

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***)**
IMObject *IM*;
caddr_t *AuxiliaryID*;
uint *Button*;
uint *PanelRow*;
uint *PanelColumn*;
uint *ItemRow*;
uint *ItemColumn*;

Description

The **IMProcessAuxiliary** subroutine is used to notify the Input Method instance of input for an Auxiliary area.

Parameters

IM Specifies the Input Method instance.

AuxiliaryID Identifies the Auxiliary area that has process.

Button Tells the type of input

IM_OK	OK button is pushed.
IM_CANCEL	CANCEL button is pushed.
IM_ENTER	ENTER button is pushed.
IM_RETRY	RETRY button is pushed.
IM_ABORT	ABORT button is pushed.
IM_IGNORE	IGNORE button is pushed.
IM_YES	YES button is pushed.
IM_NO	NO button is pushed.
IM_HELP	HELP button is pushed.
IM_SELECTED	Selection has been made. Only in this case, the *PanelRow*, *PanelColumn*, *ItemRow*, and *ItemColumn* parameters have meaningful values.

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.

Implementation Specifics

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

Related Information

The **IMAuxCreate** subroutine.

AIX Input Method Overview in *General Programming Concepts*.

IMQueryLanguage Subroutine

Purpose

Checks to see if the specified (phonic) language is supported.

Library

Input Method Library (**libIM.a**)

Syntax

uint IMQueryLanguage(*Language***)**
IMLanguage *Language***;**

Description

The **IMQueryLanguage** subroutine checks to see if the specified (phonic) language specified by the *Language* parameter is supported.

The keyboard mapping table in the Understanding Keyboard Mapping article in *General Programming Concepts* contains a listing of supported languages and their names.

Parameter

Language	The specified (phonic) language.

Return Values

The **IMQueryLanguage** subroutine returns true if the specified language is supported. Otherwise, false is returned.

Implementation Specifics

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

Related Information

The **IMInitialize** subroutine, **IMClose** subroutine.

AIX Input Method Overview in *General Programming Concepts.*

imrealloc Subroutine

Purpose

Changes the size of a block of memory.

Library

Input Method Library (**libIM.a**)

Syntax

caddr_t imrealloc(*Pointer*, *Size***)**
caddr_t *Pointer*;
uint *Size*;

Description

The **imrealloc** 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 then it returns a pointer to the block. The contents of the block remain unchanged up to the lesser of the old and new sizes.

Parameters

Pointer	Points to a block of memory.
Size	Specifies, in bytes, the new size of the block.

Return Values

If an error happens during the **imrealloc** subroutine, the subroutine calls the **abort** subroutine.

Implementation Specifics

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

Related Information

The **imcalloc** subroutine, **imfree** subroutine, **immalloc** subroutine.

The **abort** subroutine.

AIX Input Method Overview in *General Programming Concepts*.

IMRebindCode Subroutine

Purpose

Rebinds the string to the specified *KeySymbol* and *State* pair.

Library

Input Method Library (**libIM.a**)

Syntax

IMRebindCode(*IMMap, KeySymbol, State, String, NBytes***)**
IMMap *IMMap*;
KeySym *KeySymbol*;
uint *State*;
caddr_t *String*;
int *NBytes*;

Description

The **IMRebindCode** subroutine can be used to rebind the string to the specified *KeySymbol* and *State* pair. It changes the binding of the keyboard temporarily. After issuing the **IMRebindCode** subroutine, subsequent calls to the **IMAIXMapping** or **IMSimpleMapping** subroutines return the supplied string instead of the string found in the keymap file.

If the *NBytes* parameter is zero and the *String* parameter is not NULL, then the *String* parameter points to a 2–byte array that contains the code page and code points of a dead key. If the *String* parameter is NULL and *NBytes* is not zero, then *NBytes* defines a function ID.

Parameters

IMMap	Specifies the keymap.
KeySymbol	Key symbol to which the *String* parameter is bound.
State	State to which the *String* parameter is bound.
String	Rebinding string.
NBytes	Length of the rebinding string.

Implementation Specifics

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

Related Information

The **IMInitializeKeymap** subroutine, **IMFreeKeymap** subroutine, **IMSimpleMapping** subroutine, **IMAIXMapping** subroutine.

AIX Input Method Overview in *General Programming Concepts.*

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. All the parameters have the same meaning as those in the **IMAIXMapping** subroutine.

The **IMSimpleMapping** subroutine differs from the **IMAIXMapping** subroutine in that this function does not support the diacritic character sequence or the ALT NumPad sequence.

Parameters

IMMap	Identifies the keymap
KeySymbol	Key symbol to which the string is mapped.
State	State to which the string is mapped.
NBytes	Returns the length of the returning string.

Implementation Specifics

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

Related Information

The **IMAIXMapping** subroutine, **IMFreeKeymap** subroutine, **IMInitializeKeymap** subroutine.

AIX Input Method Overview in *General Programming Concepts.*

IMTextCursor Subroutine

Purpose

Callback function that sets the new display cursor position.

Syntax

int IMTextCursor(*IM*, *Direction*,
Cursor, *UData***)**
IMObject *IM*;
uint *Direction*;
int **Cursor*;
caddr_t *UData*;

Description

The **IMTextCursor** subroutine is invoked by Input Method when the cursor up or down key is input to the **IMProcess** subroutine.

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 or –1 if the cursor should not be moved.

This subroutine is a hook of the Input Method which always treats a text string as one dimensional because the Input Method does not know about actual screen. However, in the terminal emulator, text string sometimes wraps to the next line, namely, it occupies multiline. This single– to multi– line conversion is done in this subroutine. So the cursor up or down should be interpreted by the subroutine, and the subroutine informs the corresponding cursor position relative to the text string to the AIX Input Method.

Parameters

IM	Indicates the Input Method instance.
Direction	Specifies Up or Down.
Cursor	The new cursor position or –1.
UData	An application datum specified in the parameter of the **IMCreate** function.

Return Values

If an error happens, the **IMTextCursor** subroutine returns **IMError**. Otherwise, **IMNoError** is returned.

Implementation Specifics

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

Related Information

The **IMTextDraw** subroutine.

AIX Input Method Overview in *General Programming Concepts*.

IMTextDraw Subroutine

Purpose

Callback function that 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.

Parameters

IM	Indicates the Input Method instance.
TextInfo	Points to the **IMTextInfo** structure.
UData	An application datum specified in the parameter of the **IMCreate** subroutine.

Return Values

If an error happens, the **IMTextDraw** subroutine returns **IMError**. Otherwise, **IMNoError** is returned.

Implementation Specifics

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

Related Information

The **IMCreate** subroutine.

AIX Input Method Overview in *General Programming Concepts*

IMTextHide Subroutine

Purpose

Callback function that tells the application program to hide the text area.

Syntax

int IMTextHide(*IM*, *UData***)**
IMObject *IM***;**
caddr_t *UData***;**

Description

The **IMTextHide** subroutine is invoked by the Input Method when the text area should be cleared.

Parameters

IM	Indicates the Input Method instance.
UData	An application datum specified in the parameter of the **IMCreate** subroutine.

Return Values

If an error happens, the **IMTextHide** subroutine returns **IMError**. Otherwise, **IMNoError** is returned.

Implementation Specifics

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

Related Information

The **IMTextDraw** subroutine.

AIX Input Method Overview in *General Programming Concepts*.

IMTextStart Subroutine

Purpose

Callback function that 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 invoked by the Input Method when the pre–editing is started, prior to drawing anything. The purpose of this function is to notify the Input Method of the length of the pre–editing space. This function sets the length of the available space (>=0) on the display to the integer pointed to by the *Space* parameter. Setting a value of –1 is acceptable to indicate that the pre–editing space is dynamic.

For example, if the Text area where the pre–editing string is drawn to has a fixed length and growing the pre–editing string beyond the right–most boundary wouldn't be expected, changing the maximum length of the pre–editing string must be possible because ususally pre–editing starts at the current cursor position.

Parameters

IM	Indicates the Input Method instance.
Space	Maximum length of pre-editing string.
UData	An application datum specified in the parameter of the **IMCreate** subroutine.

Return Values

If an error happens, the **IMTextStart** subroutine returns **IMError**. Otherwise, **IMNoError** is returned.

Implementation Specifics

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

Related Information

The **IMCreate** subroutine.

AIX Input Method Overview in *General Programming Concepts*.

initgroups Subroutine

Purpose

Initializes concurrent group set.

Library

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

Syntax

int initgroups (*User*, *Basegid***)**
**char ****User*;
gid_t *Basegid*;

Description

The **initgroups** subroutine reads the defined group membership of the specified *User* and sets the concurrent group set of the current process to that value. The *Basegid* parameter is always included in the concurrent group set. It is normally the principal user's group. If the user is in more that **NGROUPS_MAX** groups, only **NGROUPS_MAX** groups are set, including the *Basegid* group.

Warning: The **initgroups** subroutine uses the **getgrent** subroutines. If the program that invokes **initgroups** uses any of these subroutines, then calling **initgroups** overwrites the static group structure.

Parameters

User Specifies the user whose groups are to be used to initialize the group set.

Basegid Specifies an additional group to include in the group set.

Return Values

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

Implementation Specifics

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

Related Information

The **getgroups** subroutine, **getgidx** subroutine, **setgidx** subroutine, **setgroups** subroutine.

The **getgid** subroutine.

The **setgroups** command, **groups** command.

insque or remque Subroutine

Purpose

Inserts or removes an element in a queue.

Library

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

Syntax

```
struct qelem [
   struct  qelem *next;
   struct  qelem *prev;
   char    q_data[ ];
];
```

insque (*Element*, *Pred*)
struct qelem **Element*, **Pred*;

remque (*Element*)
struct qelem **Element*;

Description

The **insque** subroutine and **remque** subroutine 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 AIX Base Operating System (BOS) Runtime.

ioctl or ioctlx Subroutine

Purpose

Performs control functions associated with open file descriptors.

Syntax

#include <sys/ioctl.h>
#include <sys/types.h>

int ioctl (*file_descriptor*, *cmd*, *argument*)
int *file_descriptor*, *cmd*;
void **argument*;

int ioctlx (*file_descriptor*, *cmd*, *argument*, *ext*)
int *file_descriptor*, *cmd*;
void **argument*;
int *ext*;

Parameters

file_descriptor — Specifies the open file descriptor for which the control operation is to be performed.

cmd — Specifies the control function to be performed. The value of this parameter depends on which object is specified by the *file_descriptor* parameter.

argument — Specifies additional information required by the function requested in the *cmd* 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.

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, with sockets, or with 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.

Most AIX device drivers support a common **ioctl** operation, IOCINFO, that returns device information. This operation and the information returned is defined in the **<sys/devinfo.h>** header file. This header file should be included if the IOCINFO **ioctl** operation in to be used. The *argument* parameter for this operation should point to a caller–provided **devinfo** structure to be filled in by the device driver specified by the open file descriptor.

Specific device operations supported by the **ioctl** function are provided by the particular device driver, usually described with the relevant special file documentation. Refer to Understanding Socket Data Transfers for a description of the **ioctl** operations supported by socket objects.

Performing an **ioctl** function on a file descriptor associated with an ordinary file results in an error being returned.

Return Values

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

Error Codes

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

EBADF The *file_descriptor* parameter is not a valid open file descriptor.

ENOTTY The *file_descriptor* parameter is not associated with an object that accepts control functions.

ENODEV The *file_descriptor* parameter is associated with a valid character or block special file, but the supporting device driver does not support the **ioctl** function.

ENXIO The *file_descriptor* parameter is associated with a valid character or block special file, but the supporting device driver is not in the configured state.

EFAULT The *argument* or *ext* parameter is used to point to data outside of the process's address space.

EINVAL The *cmd* or *argument* parameter is not valid for the specified object.

EINTR A signal was caught during the **ioctl** or **ioctlx** subroutine and the process had not enabled re-startable subroutines for the signal.

Object–specific error codes are defined in the documentation for associated with the object.

Related Information

The **ddioctl** device driver entry point.

The **fp_ioctl** kernel service.

Understanding Socket Data Transfers.

Special Files Overview, in *General Programming Concepts.*

Understanding Block I/O Device Drivers, in *Kernel Extensions and Device Support Programming Concepts.*

Understanding Character I/O Device Drivers, in *Kernel Extensions and Device Support Programming Concepts.*

Sockets Overview, in *Communications Programming Concepts.*

termio General Terminal Interface in *General Programming Concepts.*

Japanese conv Subroutines

Purpose

Translates characters.

Library

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

Japanese Language Support Syntax

When running AIX with Japanese Language Support on your system, the following subroutines, stored in the **libc.a** library, are provided:

#include <jctype.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;*

int NCwunesc (*Pointer,CharacterPointer*)
NLchar **Pointer, * CharacterPointer;*

Description

The **NCwunesc** subroutine translate all characters, including extended characters, as code points. The other subroutines translate traditional ASCII characters only.

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 zero.

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.

The **NCwunesc** subroutine converts the escape sequence pointed to by the *Pointer* parameter to a single **NLchar** pointed to by *CharacterPointer*. **NCwunesc** returns the number of **NLchar** data types used in the translation.

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**.

Implementation Specifics

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

Related Information

The **ctype** subroutines, **conv** subroutines, **getc**, **fgetc**, **getchar, getw**, **getwc**, **fgetwc**, **getwchar** subroutines, **setlocale** subroutine.

National Language Support Overview in *General Programming Concepts.*

Japanese ctype Subroutines

Purpose

Classifies characters.

Library

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

Japanese Language Support Syntax

When running AIX with Japanese Language Support on your system, the following subroutines, stored in the **libc.a** library, are provided:

#include <jctype.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*);
int*Character;*

int isjprint (*Character*)
int *Character;*

int isjgraph (*Character*)
int *Character;*

int isjis (*Character*)

Japanese ctype

int *Character;*

int isjhira (*Character*)
int *Character;*

int isjkanji (*Character*)
int *Character;*

int isjkata (*Character*)
int *Character;*

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**.

The following list shows the classification functiosn for character sets within SJIS (SJIS).

isjis *Character* is an SJIS character.

```
0xA0   – 0xDF
0x8140 – 0x817E  0x8180 – 0x81FC
   |        |        |        |
0x9F40 – 0x9F7E  0x9F80 – 0x9FFC
0xE040 – 0xE07A  0xE080 – 0xE0FC
   |        |        |        |
0xFC40 – 0xFC7E  0xFC80 – 0xFCFC
```

isjhira *Character* is a hiragana character.

```
0x829F – 0x82F1
```

isjkata *Character* is a katakana character.

```
0x8340 – 0x837E  0x8380 – 0x8396
0xA0   – 0xDF
```

isjkanji *Character* is a kanji character.

```
0x889F – 0x88FC
0x8940 – 0x897E  0x8980 – 0x89FC
   |        |        |        |
0x9740 – 0x977E  0x9780 – 0x97FC
0x9840 – 0x9872  0x989F – 0x98FC
0x9940 – 0x997E  0x9980 – 0x99FC
   |        |        |        |
0x9F40 – 0x9F7E  0x9F80 – 0x9FFC
0xE040 – 0xE07E  0xE080 – 0xE0FC
   |        |        |        |
0xE940 – 0xE97E  0xE980 – 0xE9FC
0xEA40 – 0xEA7E  0xEA80 – 0xEAA2
0xFA5C – 0xFA7E  0xFA80 – 0xFAFC
0xFB40 – 0xFB7E  0xFB80 – 0xFBFC
0xFC40 – 0xFC4B
```

The following list shows the classification functions for double–width equivalents within SJIS.

isjalpha *Character* is an alphabetic SJIS character.

```
0x8260 – 0x8279  0x8281 – 0x829A
```

isjspace *Character* is a space SJIS character.

0x8140

isjpunct *Character* is a punctuation SJIS character (neither a control character nor an alphanumeric character).

0x8141– 0x8151 0x815A – 0x8198 0x81F5 – 0x81

isjparent and isjparen
Character is a bracketing SJIS character.

0x8169 – 0x817A

isjdigit *Character* is a digit SJIS character in the range [0–9].

0x824F – 0x8258

isjxdigit *Character* is an Arabic hexadecimal SJIS character in the range [0–9], [A–F], or [a–f].

0x824F – 0x8258
0x8260 – 0x8265
0x8281 – 0x8286

isjupper *Character* is an uppercase SJIS character.

0x8260 – 0x8279

isjlower *Character* is a lowercase SJIS character.

0x8281 – 0x829A

isjprint *Character* is a printing SJIS character, including the space character.

8140 – 817E
8180 – 81AC
81B8 – 81BF
81C8 – 81C9
81CB – 81CE
81DA – 81E5
81E7 – 81E8
81F0 – 81F7
81FC
824F – 8258
8260 – 8279
8281 – 829A
829F – 82F1
8340 – 837E
8380 – 8396
839F – 83B6
83BF – 83D6
8440 – 8460
8470 – 847E
8480 – 8491
849F – 84BE
889F – 88FC
8940 – 897E 8980 – 89FC
| | | |

9740 – 977E 9780 – 97FC
9840 – 9872 989F – 98FC
9940 – 997E 9980 – 99FC
| | | |
9F40 – 9F7E 9F80 – 9FFC
E040 – E07E E080 – E0FC
| | | |
E940 – E97E E980 – E9FC
EA40 – EA7E EA80 – EAA2
FA40 – FA7E FA80 – FAFC
FB40 – FB7E FB80 – FBFC
FC40 – FC4B

isjgraph *Character* is a printing SJIS character, excluding the space character.

8141 – 817E
8180 – 81AC
81B8 – 81BF
81C8 – 81C9
81CB – 81CE
81DA – 81E5
81E7 – 81E8
81F0 – 81F7
81FC
824F – 8258
8260 – 8279
8281 – 829A
829F – 82F1
8380 – 8396
839F – 83B6
83BF – 83D6
8440 – 8460
8470 – 847E
8480 – 8491
849F – 84BE
889F – 88FC
8940 – 897E 8980 – 89FC
| | | |
9740 – 977E 9780 – 97FC
9840 – 9872 989F – 98FC
9940 – 997E 9980 – 99FC
| | | |
9F40 – 9F7E 9F80 – 9FFC
E040 – E07E E080 – E0FC
| | | |
E940 – E97E E980 – E9FC
EA40 – EA7E EA80 – EAA2
FA40 – FA7E FA80 – FAFC
FB40 – FB7E FB80 – FBFC
FC40 – FC4B8340 – 837E

Parameter

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 AIX Base Operating System (BOS) Runtime.

Related Information

The **ctype** subroutines, **NCctype** subroutines and **setlocale** subroutine.

National Language Support Overview in *General Programming Concepts*.

jcode Subroutines

Purpose

Perform string conversion on 8–bit processing codes.

Library

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

Japanese Language Support Syntax

When running AIX with Japanese Language Support on your system, the following subroutines, stored in **libc.a**, are provided:

#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 of 8–bit processing JIS Kanji characters. A character is removed from the *String2* parameter, 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 **cjistos**j 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 — String or code to be converted.

Implementation Specifics

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

Related Information

The **Japanese conv** subroutines, **Japanese ctype** subroutines.

kill or killpg Subroutine

Purpose

Sends a signal to a process or to a group of processes.

Library

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

Syntax

int kill(*Process***,** *Signal***)**
pid_t *Process***;**
int *Signal***;**

killpg(*ProcessGroup***,** *Signal***)**
int *ProcessGroup***;**
int *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 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 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 that has a process ID equal to the value of the *Process* parameter.

If the *Process* parameter is 0, the signal specified by the *Signal* parameter is sent to all of the processes, excluding proc0 and proc1, whose process group ID is equal to the process group ID of the sender.

If the *Process* parameter is –1, the signal specified by the *Signal* parameter is sent to all of the processes, excluding proc0 and proc1, if the calling process passes the access checks for the process to be signalled. If the calling process effective user ID has root user authority, all processes, excluding proc0 and proc1, are signalled.

If the *Process* parameter is negative but not –1, the signal specified by the *Signal* parameter is sent to all of the processes which have a

	process group ID equal to the absolute value of the *Process* parameter.
Signal	Specifies the signal. If the *Signal* parameter is 0 (the null signal), error checking is performed but no signal is sent. This can be 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 global variable **errno** is set to indicate the error.

Error Codes

The **kill** subroutine fails 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 is **SIGKILL**, **SIGSTOP**, **SIGTSTP** or **SIGCONT** 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 the calling process does not have root user authority.

Implementation Specifics

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

The following interface is provided for BSD Compatibility:

```
killpg(ProcessGroup, Signal)
int ProcessGroup;
int Signal;
```

is equivalent to:

```
if (ProcessGroup < 0)
{
  errno = ESRCH;
   return (–1);
}
return (kill(–ProcessGroup, Signal));
```

Related Information

The **getpid**, **getpgrp**, **getppid** subroutines, **setpgid**, **setpgrp**subroutines, **sigaction, signal**, **sigvec** subroutines.

The **kill** command.

kleenup Subroutine

Purpose

Cleans up the run–time environment of a process.

Library

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

Syntax

int kleenup (*FileDescriptor*, *SigIgn*, *SigKeep*)
int *FileDescriptor*;
int *SigIgn*[ ];
int *SigKeep*[ ];

Description

The **kleenup** subroutine initializes the run–time environment for a trusted process by:

- Closing unnecessary file descriptors.
- Resetting the alarm time.
- Resetting signal handlers.
- Turning off **UCOMPAT_DIRSYS5.**
- Resetting the **ulimit** value, if it is less than a reasonable value (8192).

Parameters

FileDescriptor	A file descriptor; the **kleenup** subroutine closes all file descriptors greater than or equal to the *FileDescriptor* parameter.
SigIgn, SigKeep	Pointers to lists of signal numbers. If non–**NULL**, these lists are terminated by zeros. The handling of any signals specified by the *SigKeep* parameter is left unchanged. Any signals specified by the *SigIgn* parameter are set to **SIG_IGN**. The handling of all signals not specified by either list is set to **SIG_DFL**. Some signals cannot be reset and are left unchanged.

Return Value

The **kleenup** subroutine is always successful and always returns a value of 0. Errors in closing files are not reported, and 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 AIX Base Operating System (BOS) Runtime.

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 remaining fields are filled in by **knlist**. The **nlist** structure consists of the following fields:

char *n_name	The name of the symbol whose attributes are to be retrieved.
long n_seg	A descriptor for the segment in which the object named by the symbol resides. The only use of this descriptor is as the *Extension* parameter on a subroutine against /**dev**/**mem**.
long n_value	The offset of the object in this segment.
unsigned short n_type	Symbol type.
short n_scnum	Section number.
char n_sclass	Storage class.

If the name is not found, both the n_value and n_type fields are set to 0.

The **nlist.h** header file is automatically included by **a.out.h** for compatibility. However, do not include **a.out.h** if you only need the information necessary to use the **knlist** subroutine. If you do include **a.out.h**, follow the **#include** statement with the line:

```
#undef 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 global variable **errno** is set to indicate the error.

Error Code

The **knlist** subroutine fails when the following is true:

EFAULT The *NList* parameter points outside the limit of the array of **nlist** structures.

Implementation Specifics

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

Related Information

The **nlist** subroutine.

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.

Warning: The numerical values of the long integers are machine-dependent because of possible differences in byte ordering.

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 AIX Base Operating System (BOS) Runtime.

Related Information

The **fs** file format.

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 <filehdr.h>
#include <ldfcn.h>

int ldahread (*ldPointer*, *ArchiveHeader*)
LDFILE **ldPointer*;
ARCHDR **ArchiveReader*;

Description

If TYPE(*ldPointer*) is the archive file magic number, the **ldahread** routine reads the archive header of the common object file currently associated with *ldPointer* into the area of memory beginning at *ArchiveHeader.*

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 FILHDR structure.

Return Values

The **ldahread** subroutine returns SUCCESS or FAILURE.

Error Codes

The **ldahread** routine fails if TYPE(*ldPointer*) does not represent an archive file, or if it cannot read the archive header.

Implementation Specifics

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

Related Information

The **ldfhread** subroutine, **ldlread, ldlinit, ldlitem** subroutines, **ldshread, ldnshread** subroutines, **ldtbread** subroutine, **ldgetname** subroutine.

ldclose or ldaclose Subroutine

Purpose

Closes a common object file.

Library

Object File Access Routine Library (**libld.a**)

Syntax

#include <stdio.h>
#include <filehdr.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 TYPE(*ldPointer*) is the magic number of an archive file, and if there are any more files in the archive, the **ldclose** subroutine reinitializes OFFSET(*ldPointer)* to the file address of the next archive member and returns FAILURE. The **ldfile** structure is prepared for a subsequent **ldopen.**

If TYPE(*ldPointer*) 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 *ldPointer* regardless of the value of TYPE*(ldPointer).*

Parameter

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 SUCCESS or FAILURE.

The **ldaclose** subroutine always returns SUCCESS, and is often used in conjunction with the **ldaopen** subroutine.

Error Code

The **ldclose** subroutine returns FAILURE if there are more files to archive.

Implementation Specifics

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

Related Information

The **ldopen, ldaopen** subroutines.

ldfhread Subroutine

Purpose

Reads the file header of a common object file.

Library

Object File Access Routine Library (**libld.a**)

Syntax

#include <stdio.h>
#include <filehdr.h>
#include <ldfcn.h>

int ldfhread *(ldPointer, FileHeader)*
LDFILE **ldPointer;*
FILHDR **FileHeader;*

Description

The **ldfhread** subroutine reads the file header of the common object file currently associated with *ldPointer* into the area of memory beginning at *FileHeader.*

Parameters

ldPointer	Pointer to the LDFILE structure that was returned as the result of a successful call to **ldopen** or **ldaopen**.
FileHeader	Pointer to a FILHDR structure.

Return Values

The **ldfhread** subroutine returns SUCCESS or FAILURE.

Error Codes

The **ldfhread** subroutine fails if if it cannot read the file header.

Note: In most cases, the use of **ldfhread** can be avoided by using the macro **header**(*ldPointer*) defined in **ldfcn.h.** The information in any field or fieldname of the file header may be accessed using **header** *(dPointer)* **fieldname.**

Implementation Specifics

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

Related Information

The **ldahread** subroutine,**ldlread, ldlinit, ldlitem** subroutines, **ldshread, ldnshread** subroutines, **ldtbread** subroutine, **ldgetname** subroutine.

ldgetname Subroutine

Purpose

Retrieves symbol name for common object file symbol table entry.

Library

Object File Access Routine Library (**libld.a**)

Syntax

#include <stdio.h>
#include <filehdr.h>
#include <syms.h>
#include <ldfcn.h>

char *ldgetname *(ldPointer, Symbol)*
LDFILE **ldPointer;*
SYMENT **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 **ldgetname** that is overwritten by each call to **ldgetname,** and therefore, must 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.

Parameters

ldPointer Points to the LDFILE structure that was returned as the result of a successful call to **ldopen** or **ldaopen**.

Symbol Points to an initialized SYMENT structure.

Error Codes

The **ldgetname** subroutine returns NULL (defined in the **stdio.h file)** for a COFF-format object file if the name cannot be retrieved. This situation can occur:

if the string table cannot be found,

if not enough memory can be allocated for the string table,

if the string table appears not to be a string table (for example, if an auxiliary entry is handed to **ldgetname** that looks like a reference to a name in a non-existent string table), or

if 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.

Implementation Specifics

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

Related Information

The **ldahread** subroutine, **ldfhread** subroutine, **ldlread, ldlinit, ldlitem** subroutines, **ldshread, ldnshread** subroutines, **ldtbread** subroutine.

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 <filehdr.h>
#include <linenum.h>
#include <ldfcn.h>

int ldlread (*ldPointer*, *LineNumber*, *LineEntry*)
LDFILE **ldPointer;*
long *FunctionIndex*;
unsigned short *LineNumber;*
LINENO *LineEntry;*

int ldlinit (*ldPointer*, *FunctionIndex*)
LDFILE **ldPointer*;
long *FunctionIndex;*

int ldlitem (*ldPointer*, *LineNumber*, *LineEntry*)
LDFILE **ldPointer*;
unsigned short *LineNumber;*
LINENO *LineEntry*;

Description

The **ldlread** subroutine searches the line number entries of the common object file currently associated with *ldPointer*. 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 *FunctionIndex*, 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 LineNumber into the memory beginning at *LineEntry*.

The **ldlinit** subroutine and **ldlitem** subroutine together perform exactly the same function as the **ldlread** routine. After an initial call to **ldlread** or **ldlinit**, **ldlitem** may be used to retrieve a series of line number entries associated with a single function. The **ldlinit** subroutine simply locates the line number entries for the function identified by *FunctionIndex*. The **ldlitem** subroutine finds and reads the entry with the smallest line number equal to or greater than *LineNumber* into the memory beginning at *LineEntry*.

Parameters

ldPointer	Points to the LDFILE structure that was returned as the result of a successful call to **ldopen** or **ldaopen**.
LineNumber	Specifies the index of the first *LineNumber* entry to be read.
LineEntry	Points to a LINENO structure.
FunctionIndex	Points to the symbol table index of a function.

Return Values

The **ldlread**, **ldlinit**, and **ldlitem** subroutines return SUCCESS or FAILURE.

Error Codes

The **ldlread** subroutine fails if there are no line number entries in the object file, if *FunctionIndex* does not index a function entry in the symbol table, or if it finds no line number equal to or greater than *LineNumber*. The **ldlinit** subroutine fails if there are no line number entries in the object file or if *FunctionIndex* 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 *LineNumber*.

Implementation Specifics

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

Related Information

The **ldahread** subroutine, **ldfhread** subroutine, **ldshread, ldnshread** subroutines, **ldtbread** subroutine, **ldgetname** subroutine.

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 <filehdr.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 *SectionIndex* of the common object file currently associated with *ldPointer*. The first section has an index of 1.

The **ldnlseek** subroutine seeks to the line number entries of the section specified by *SectionName*.

Parameters

ldPointer	Points to the LDFILE structure that was returned as the result of a successful call to **ldopen** or **ldaopen**.
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 SUCCESS or FAILURE.

Error Codes

The **ldlseek** subroutine fails if *SectionIndex* is greater than the number of sections in the object file; the **ldnlseek** subroutine fails if there is no section name corresponding with *SectionName*. 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 AIX Base Operating System (BOS) Runtime.

Related Information

The **ldohseek** subroutine, **ldsseek, ldnsseek** subroutines, **ldtbseek** subroutine, **ldrseek, ldnrseek** subroutines.

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 <filehdr.h>
#include <ldfcn.h>

int ldohseek (*ldPointer*)
LDFILE **ldPointer*;

Description

The **ldohseek** subroutine seeks to the optional file header of the common object file currently associated with *ldPointer*.

Parameter

ldPointer Points to the LDFILE structure that was returned as the result of a successful call to **ldopen** or **ldaopen**.

Return Values

The **ldohseek** subroutine returns SUCCESS or FAILURE.

Error Codes

The **ldohseek** subroutine fails if the object file has no optional header or if it cannot seek to the optional header.

Implementation Specifics

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

Related Information

The **ldsseek, ldnsseek** subroutines, **ldtbseek** subroutine, **ldrseek,ldnrseek** subroutines, **ldlseek, ldnlseek** subroutines.

ldopen or ldaopen Subroutine

Purpose

Opens a common object file for reading.

Library

Object File Access Routine Library (**libld.a**)

Syntax

#include <stdio.h>
#include <filehdr.h>
#include <ldfcn.h>

ldfile *ldopen(*FileName*, *ldPointer***)**
char **FileName*;
LDFILE **ldPointer*;

LDFILE *ldaopen(*FileName, ldPointer***)**
char **FileName*;
LDFILE **ldPointer*;

Description

The **ldopen** subroutine and **ldclose** subroutine 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 TYPE(*ldPointer*) has the value NULL, the **ldopen** subroutine opens *FileName* and allocate and initialize the **ldfile** structure, and returns a pointer to the structure to the calling program.

If *ldPointer* is valid and if TYPE(*ldPointer)* is the archive magic number, the **ldopen** subroutine reinitializes the **ldfile** structure for the next archive member of *FileName.*

The **ldopen** and **ldclose** subroutines are designed to work in concert. The **ldclose** subroutine returns FAILURE only when TYPE(*ldPointer*) is the archive magic number and there is another file in the archive to be processed. Only then should **ldopen** be called with the current value of *ldPointer.* In all other cases, in particular whenever a new *FileName* is opened, **ldopen** should be called with a NULL *ldPointer* argument.

The following is an example for the use of **ldopen** and **ldclose**:

```
/* 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 );
```

If the value of *ldPointer* is not NULL, the **ldaopen** subroutine opens *FileName* again and allocate and initializes a new **ldfile** structure, copying the TYPE, OFFSET, and HEADER fields from *ldPointer*. 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.

Parameters

ldPointer	Pointer to the LDFILE structure.
FileName	Specifies the file name of an object file or archive of object files.

Error Codes

Both **ldopen** and **ldaopen** open *FileName* for reading. Both functions return NULL if *FileName* cannot be opened, or if memory for the **ldfile** structure cannot be allocated. A successful open does not insure that the given file is a common object file or an archived object file.

Implementation Specifics

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

Related Information

The **ldclose, ldaclose** subroutines.

The Extended Common Object File Format (XCOFF).

ldrseek or ldnrseek Subroutine

Purpose

Seeks to relocation entries of a section of a common object file.

Library

Object File Access Routine Library (**libld.a**)

Syntax

#include <stdio.h>
#include <filehdr.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 *SectionIndex* of the common object file currently associated with *ldPointer.*

The **ldnrseek** subroutine seeks to the relocation entries of the section specified by *SectionName.*

Parameters

ldPointer	Points to the LDFILE structure that was returned as the result of a successful call to **ldopen** or **ldaopen**.
SectionIndex	Specifies an index of the section whose relocation entries are to be seeked to.
SectionName	Specifies the name of the section whose relocation entries are to be seeked to.

Return Values

The **ldrseek** and **ldnrseek** subroutines return SUCCESS or FAILURE.

Error Codes

The **ldrseek** subroutine fails if *SectionIndex* is greater than the number of sections in the object file; **ldnrseek** fails if there is no section name corresponding with **SectionName.* Either function fails if the specified section has no relocation entries or if it cannot seek to the specified relocation entries. Note that the first section has an index of 1.

Implementation Specifics

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

Related Information

The **ldohseek** subroutine, **ldtbseek** subroutine, **ldsseek, ldnsseek** subroutines, **ldlseek, ldnlseek** subroutines.

ldshread or ldnshread Subroutine

Purpose

Reads an indexed/named section header of a common object file.

Library

Object File Access Routine Library (**libld.a**)

Syntax

#include <stdio.h>
#include <filehdr.h>
#include <ldfcn.h>
#include <ldfcn.h>

int ldshread (*ldPointer*, *SectionIndex*, *SectionHead*)
LDFILE **ldPointer;*
unsigned short *SectionIndex;*
SCNHDR **SectionHead;*

int ldnshread (*ldPointer, SectionName, SectionHead*)
LDFILE **ldPointer;*
char **SectionName;*
SCNHDR **SectionHead;*

Description

The **ldshread** subroutine reads the section header specified by *SectionIndex* of the common object file currently associated with *ldPointer* into the area of memory beginning at *SectionHead.*

The **ldnshread** subroutine reads the section header specified by *SectionName* into the area of memory beginning at *SectionHead.*

Parameters

ldPointer	Points to the LDFILE structure that was returned as the result of a successful call to **ldopen** or **ldaopen**.
SectionIndex	Specifies the index of the section header to be read.
SectionHead	Specifies the name of the section header to be read.
SectionName	Points to an SCNHDR structure.

Return Values

The **ldshread** and **ldnshread** subroutines return SUCCESS or FAILURE.

Error Codes

The **ldshread** subroutine fails if *SectionIndex* is greater than the number of sections in the object file; the **ldnshread** subroutine fails if there is no section name corresponding with *SectionName.* Either function fails if it cannot read the specified section header. Note that the first section has an index of 1.

Implementation Specifics

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

Related Information

The **ldahread** subroutine, **ldfhread** subroutine, **ldlread, ldlinit, ldlitem** subroutines, **ldtbread** subroutine, **ldgetname** subroutine.

ldsseek or ldnsseek Subroutine

Purpose

Seeks to an indexed/named section of a common object file.

Library

Object File Access Routine Library (**libld.a**)

Syntax

#include <stdio.h>
#include <filehdr.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 *SectionIndex* of the common object file currently associated with *ldPointer.*

The **ldnsseek** subroutine seeks to the section specified by *SectionName.*

Parameters

ldPointer — Points to the LDFILE structure that was returned as the result of a successful call to **ldopen** or **ldaopen**.

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 SUCCESS or FAILURE.

Error Codes

The **ldsseek** subroutine fails if *SectionIndex* is greater than the number of sections in the object file; **ldnsseek** fails if there is no section name corresponding with *SectionName.* Either function fails if there is no section data for the specified section or if it cannot seek to the specified section. Note that the first section has an index of 1.

Implementation Specifics

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

Related Information

The **ldohseek** subroutines, **ldtbseek** subroutine, **ldrseek**, **ldnrseek** subroutines, **ldlseek**, **ldnlseek** subroutines.

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 <filehdr.h>
#include <syms.h>
#include <ldfcn.h>

long ldtbindex (*ldPointer*)
LDFILE **ldPointer*;

Description

The **ldtbindex** subroutine returns the (LONG) index of the symbol table entry at the current position of the common object file associated with *ldPointer.*

The index returned by **ldtbindex** may be used in subsequent calls to **ldtbread.** However, since **ldtbindex** returns the index of the symbol table entry that begins at the current position of the object file, if **ldtbindex** is called immediately after a particular symbol table entry has been read, it returns the index of the next entry.

Parameter

ldPointer	Points to the LDFILE structure that was returned as a result of a successful call to **ldopen** or **ldaopen**.

Error Codes

The **ldtbindex** routine 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 that the first symbol in the symbol table has an index of 0.

Implementation Specifics

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

Related Information

The **ldtbseek** subroutine, **ldtbread** subroutine.

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 <filehdr.h>
#include <syms.h>
#include <ldfcn.h>

int ldtbread (*ldPointer, SymbolIndex, Symbol*)
LDFILE **ldPointer;*
long *SymbolIndex;*
SYMENT **Symbol;*

Description

The **ldtbread** subroutine reads the symbol table entry specified by *SymbolIndex* of the common object file currently associated with *ldPointer* into the area of memory beginning at *Symbol.*

Parameters

ldPointer	Points to the LDFILE structure that was returned as the result of a successful call to **ldopen** or **ldaopen**.
SymbolIndex	Specifies the index of the symbol table entry to be read.
Symbol	Points to a SYMENT structure.

Return Values

The **ldtbread** subroutine returns SUCCESS or FAILURE.

Error Codes

The **ldtbread** subroutine fails if *SymbolIndex* is greater than or equal to number of symbols in the object file, or if it cannot read the specified symbol table entry. Note that the first symbol in the symbol table has an index of 0.

Implementation Specifics

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

Related Information

The **ldahread** subroutine, **ldfhread** subroutine, **ldlread, ldlinit, ldlitem** subroutines, **ldshread, ldnshread** subroutines, **ldgetname** subroutine.

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 <filehdr.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 *ldPointer.*

Parameter

ldPointer	Points to the LDFILE structure that was returned as the result of a successful call to **ldopen** or **ldaopen.**

Return Values

The **ldtbseek** subroutine returns SUCCESS or FAILURE.

Error Codes

The **ldtbseek** subroutine fails if the symbol table has been stripped from the object file, or if it cannot seek to the symbol table.

Implementation Specifics

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

Related Information

The **ldohseek** subroutine, **ldrseek, ldnrseek** subroutines, **ldsseek, ldnsseek** subroutines, **ldlseek, ldnlseek** subroutines.

lgamma or gamma Subroutine

Purpose

Computes the natural logarithm of the gamma function. The subroutine names **lgamma** and **gamma** are different names for the same function.

Library

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

Syntax

#include <math.h>

extern int signgam;

double lgamma (*x*)
double *x*;

double gamma (*x*)
double *x*;

Description

The **lgamma** subroutine returns the natural logarithm of the absolute value of the gamma function of the *x* parameter, where the gamma function of x is defined as:

```
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;
```

because the variable signgam can be relied on only after **lgamma** has finished execution.

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

```
cc lgamma.c -lm
```

Parameter

x Specifies some double-precision floating-point value.

Error Codes

When using **libm.a** (**-lm**):

For non-positive integer arguments, the **lgamma** function returns NaNQ and sets the division-by-zero bit in the floating-point exception status.

If the correct value overflows, **lgamma** returns HUGE_VAL. If the correct value underflows, **lgamma** returns 0.

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

For non-positive integer arguments, the **lgamma** function returns HUGE_VAL, and sets the global variable **errno** is set to EDOM. A message indicating SING error is printed on the standard error output.

If the correct value overflows, **lgamma** returns HUGE_VAL, and sets the global variable **errno** is set to ERANGE.

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

Implementation Specifics

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

Related Information

The **exp, expm1, log, log10, log1p, pow** subroutines, **matherr** subroutine.

link Subroutine

Purpose

Creates an additional directory entry for an existing file.

Library

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

Syntax

int link (*Path1*, *Path2***)**
char **Path1*, **Path2*;

Description

The **link** subroutine creates an additional hard link (directory entry) for an existing file. Both the old and the new link 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 for the new directory entry to be created.

If Network File System is installed on your system, these paths can cross into another node.

With hard links, both the *Path1* and *Path2* parameters must reside on the same file system. 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 global variable **errno** is set to indicate the error.

Error Codes

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

ENOENT	The file named by the *Path1* parameter does not exist.
EEXIST	The link named by the *Path2* parameter already exists.
EPERM	The file named by the *Path1* parameter is a directory and the calling process does not have root user authority.
EXDEV	The link named by the *Path2* parameter and the file named by the *Path1* parameter are on different file systems.
EACCES	The requested link requires writing in a directory with a mode that denies write permission.
EMLINK	The file already has the maximum number of links.
EROFS	The requested link requires writing in a directory on a read–only file system.

ENOSPC 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.

EDQUOT The directory in which the entry for the new link is being placed cannot be extended because the user's quota of disk blocks on the file system containing the directory has been exhausted.

The **link** subroutine can also fail if additional errors on page A–1 occur.

If Network File System is installed on the system, the **link** system call can also fail if the following is true:

ETIMEDOUT The connection timed out.

Implementation Specifics

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

Related Information

The **unlink** subroutine.

The **link** command, **ln** command, **rm** command.

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 object file for the program into the calling process. Unlike the **exec** subroutine, **load** does not replace the current program with the new one. Instead, it loads the new program into the process private segment at the current break value and the break value is updated to point past the new program.

The **exec** subroutine is similar to the **load** subroutine, except that **exec** does not have an explicit library path parameter; it has only the LIBPATH environment variable. Also, LIBPATH is ignored when the **exec**'d program 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, once the file is loaded, the space is unusable by the process except through the **load** subroutine. If the kernel finds an unused space created by a previous unload, rather than load the program at the break value, it loads the program into this unused space. 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 object files in one of two ways that allows execution within the same process. The first way is to create each of the application's object files separately and use **load** to explicitly load an object when it is needed. The other way is to specify the relationship between the object files when they are created by defining imported and exported symbols.

Object files can import symbols from other object files. Whenever symbols are imported from one or more other object files, these object files are automatically loaded to resolve the symbol references if the required object files are not already loaded, and if the imported symbols are not specified as "deferred resolution". These object files 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 object files (typically members of a shared library archive) are loaded into the shared library region, when their access permissions are such that sharing is acceptable. Shared object files without the required permissions for sharing and private object files are loaded into the process private region.

When the loader resolves a symbol it uses the filename recorded with that symbol to find the object file that exports the symbol. If the file name contains any "/" characters, it is used directly and must name an appropriate object file. However, if the filename is a basename (contains no "/" characters), the loader searches the directories specified in the default library path for an object file with that basename.

The library path is a string containing one or more directory path names separated by a colon. If the basename is not found the search continues, using the library path specfied 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 basename found is used. An error occurs if this object file 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 fro the LIBPATH environment variable or from the object file specified by the *FilePath* parameter.

Programs loaded by this subroutine are automatically unloaded when the process terminates or when **exec** is executed. They are explicitly unloaded by calling the **unload** subroutine.

Parameters

**FilePath*
: A pointer to the name of the object file to be loaded. If the *FilePath* name contains no "/" symbols, it is treated as a basename, and should be in one of the directories listed in the library path.

  The library path is either the value of *LibraryPath* (if not NULL), or the value of LIBPATH (if set). If no library path is provided, the object file should be in the current directory.

  If *FilePath* is not a basename (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.

Flags
: Used to modify the behaviour of the **load** service as follows (see the **ldr.h** file):

  1 — The typical value for loading modules.

  L_NOAUTODEFER –Specifies that any unresolved imports (designated for deferred resolution) 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 object module. If this flag is not specified, unresolved imports (marked for deferred resolution) are resolved at the earliest opportunity when any module is loaded that has exported symbols matching unresolved imports.

LibraryPath
: A pointer to a character string that specifies the default library search path.

  If *LibraryPath* is NULL and LIBPATH 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 *FilePath* is used as the default library path.

  If the object file is not in *LibraryPath* or LIBPATH (if *LibraryPath* was NULL), then the library path specified in the loader section of the object file importing the symbol is used, to locate the object file exporting the required symbol. The library path in the importing object file was specified when the object file was link edited (by the **ld** command).

  The library path search is not performed when either a relative or an absolute pathname is specified for the object file exporting the symbol.

Return Values

Upon successful completion, the **load** subroutine returns the pointer to function for the main entry point of the program.

Error Codes

If the **load** subroutine fails, a NULL pointer is returned, the program is not loaded, and **errno** is set to indicate the error. The **load** subroutine fails if one or more of the following are true of an object file to be explicitly or automatically loaded:

EACCES The program 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 The program 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.

ELOOP Too many symbolic links were encountered in translating the pathname.

ENOEXEC An error occurred when loading or resolving symbols for the specified object file. This can be due to an attempt to load an object file with an invalid XCOFF header, a failure to resolve symbols that were not specified as "deferred resolution" or several other load time related problems. The **loadquery** subroutine can be used to return more information about the load failure.

ENOMEM The program requires more memory than is allowed by the system-imposed maximum.

ETXTBSY The program file is currently open for writing by some process.

ENAMETOOLONG

A component of a path name exceeded 255 characters, or an entire path name exceeded 1023 characters.

ENOENT A component of the path prefix does not exist, or the path name is NULL.

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

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

Implementation Specifics

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

Related Information

The **ld** command.

The **exec** subroutine, **unload** subroutine, **ldbind** subroutine, **loadquery** subroutine.

loadbind Subroutine

Purpose

Provides specific runtime resolution of a module's deferred symbols.

Syntax

int loadbind(*Flag*, *ExportPointer*, *ImportPointer***)**
int *Flag*;
void ******ExportPointer*, ******ImportPointer*;

Description

The **loadbind** subroutine controls the runtime resolution of a previously loaded object module's unresolved imported symbols.

The **loadbind** subroutine is used when the following occurs: two modules are loaded. Module A, an object module loaded at runtime 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.

Parameters

Flag	Currently not used.
ExportPointer	Set to the function pointer returned by the **load** subroutine when module B was loaded.
ImportPointer	Set to the function pointer returned by the **load** subroutine when module A was loaded.

The *ImportPointer* or *ExportPointer* parameters 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	Either the *ImportPointer* or *ExportPointer* is not valid (the pointer to *ExportPointer* or *ImportPointer* does not correspond to a loaded program module or library).
ENOMEM	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 AIX Base Operating System (BOS) Runtime.

Related Information

The **load** subroutine, **unload** subroutine, **loadquery** subroutine.

The **ld** command.

loadquery Subroutine

Purpose

Returns error information from the **load** subroutine 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** subroutine 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.

Parameters

Buffer — Points to a *Buffer* in which to store error message or object file information.

BufferLength — Specifies the number of bytes available in *Buffer*.

Flags — Specifies the action of the **loadquery** function as follows:

L_GETINFO – Returns a list of all object files loaded for the current process, and stores the list in *Buffer*. The object file information is contained in a sequence of LD_INFO structures as defined in the **sys/ldr.h** header file. Each structure contains the module location in virtual memory and the pathname 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_GETMESSAGES – 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** header file.

You can format the error messages returned by the L_GETMESSAGES function and write them to standard error using the standard system command **/etc/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("/etc/execerror",buffer);
```

This sample code causes the application to terminate after the messages are written to standard error.

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 global variable **errno** is set to one of the following when an error condition is detected:

ENOMEM 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 The function specified in the *Flags* parameter is not valid or an error occurred when accessing the caller's buffer.

Implementation Specifics

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

Related Information

The **exec** subroutine, **load** subroutine, **unload** subroutine, **loadbind** subroutine.

The **ld** command.

localeconv Subroutine

Purpose

Sets the locale dependent conventions of an object.

Library

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

Syntax

#include <locale.h>
struct lconv *localeconv ( )

Description

The **localeconv** subroutine sets the components of an object using the **lconv** structure. The **lconv** structure contains values appropriate for the formatting of numeric quantities (monetary and otherwise) according to the rules of the current locale.

The members of the structure with the type **char *** are strings, any of which (except **decimal_point**) can point to a **NULL** string, to indicate that the value is not available in the current locale or is of zero length. The members with type **char** are nonnegative numbers, any of which can be **CHAR_MAX** to indicate that the value is not available in the current locale. The members 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.

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.

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 or 0 if the **currency_symbol** respectively precedes or succeeds the value for a nonnegative formatted monetary quantity.
char p_sep_by_space	Set to 1 or 0 if the **currency_symbol** respectively is or is not separated by a space from the value for a nonnegative formatted monetary quantity.
char n_cs_precedes	Set to 1 or 0 if the **currency_symbol** respectively precedes or succeeds the value for a negative formatted monetary quantity.
char n_sep_by_space	Set to 1 or 0 if the **currency_symbol** respectively is or is not separated by a space from the value for a negative formatted monetary quantity.
char p_sign_posn	Set to a value indicating the positioning of the **positive_sign** for nonnegative formatted monetary quantity.
char n_sign_posn	Set to a value indicating the positioning of the **negative_sign** for a negative formatted monetary quantity.

The elements of **grouping** and **mon_grouping** are 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.

The value of **p_sign_posn** and **n_sign_posn** is interpreted according to the following:

0	Parenthesis surround the quantity and **currency_symbol**.
1	The sign string precedes the quantity and **currency_symbol**.
2	The sign string succeeds the quantity and **currency_symbol**.
3	The sign string immediately precedes the **currency_symbol**.
4	The sign string immediately succeeds the **currency_symbol**.

Return Values

A pointer to the filled–in object is returned. The structure pointed to by the return value shall not be modified by the program, but may be overwritten by a subsequent call to **localeconv**.

In addition, calls to **setlocale** with categories **LC_ALL**, **LC_MONETARY** or **LC_NUMERIC** may cause subsequent calls to **localeconv** to return different values based on the selection of the locale.

Implementation Specifics

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

Related Information

The **setlocale** subroutine.

National Language Support Overview in *General Programming Concepts*.

lockfx, lockf or flock Subroutine

Purpose

Controls open file descriptors.

Library

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

Syntax

#include <fcntl.h>

int lockfx (*FileDescriptor, Command, Argument*)
int *FileDescriptor*;
int *Command*;
struct flock **Argument*;

#include <sys/lockf.h>

int lockf(*FileDescriptor, Request, Size*)
int *FileDescriptor*;
int *Request*;
off_t *Size*;

#include <sys/file.h>

int flock(*FileDescriptor, Operation*)
int *FileDescriptor*;
int *Operation*;

Description

The **lockfx** subroutine is used to lock and unlock sections of an open file. **lockfx** provides a subset of 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, the implementations are fully integrated. Therefore, locks obtained from one subroutine are honored and enforced by any of the lock subroutines.

Warning: 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.

A parameter to the **lockfx** subroutine that 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.

Parameters

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.

Command
One of the following constants for **lockfx**:

- **F_SETLK**: Sets or clears a file lock. The l_type field of the **flock** structure indicates whether to establish a read or write lock, or to remove either type of 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** except that if a read or write lock is blocked by existing locks, 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**.

Argument
A pointer to a structure of type **flock**, defined in the **flock.h** header file.

Request
One of the following constants for **lockf**:

- **F_ULOCK**: Unlocks a previously locked region in the file.
- **F_LOCK**: Locks the region for exclusive use. This request causes the calling process to sleep if the region overlaps a locked region, and to resume when it is 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 global variable **errno** is set to **EACCES**.

Size
The number of bytes to be locked or unlocked for **lockf**. The region starts at the current location in the open file and extends forward if *Size* is positive and backward if *Size* is negative. If the *Size* parameter 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.

Operation
One of the following constants for **flock**:

- **LOCK_SH**: Apply a shared lock.
- **LOCK_EX**: Apply an exclusive 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.

Return Values

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

Error Codes

The **lockfx**, **lockf**, and **flock** subroutines fail if one or more of the following are true:

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

EINVAL The request is not valid.

EDEADLK The lock is blocked by some lock from another process. Putting the calling process to sleep while waiting for that lock to become free would cause a deadlock.

ENOLCK The lock table is full. Too many regions are already locked.

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

EAGAIN 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.

EAGAIN 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 **lockf** 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 AIX 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. In order to apply a shared lock, the file must be opened for reading, and to apply an exclusive lock, it must be opened for writing. Also, locks are not inherited; therefore, a child process cannot unlock a file locked by the parent process.

Related Information

The **close** subroutine, **execve** subroutine, **fcntl** subroutine, **fork** subroutine, **open** subroutine.

The **flock.h** header file, **sys/file.h** header file.

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***)**
**void ****Key*, *Base*;
size_t *Width*, *NumberOfElementsPointer*;
int (*ComparisonPointer***) ( );**

void *lfind (*Key***,** *Base***,***NumberOfElementsPointer***,** *Width*, *ComparisonPointer***)**
**void ****Key*, *Base*;
size_t *Width*, *NumberOfElementsPointer*;
int (*ComparisonPointer***) ( );**

Description

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.

Warning: Undefined results can occur if there is not enough room in the table for the **lsearch** subroutine to add a new item.

Parameters

Key Specifies the data to be sought in the table.

Base Points to the first element 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.

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.

Width Specifies the size of an element in bytes.

Return Values

The comparison function compares its parameters and return 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.

Implementation Specifics

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

Related Information

The **bsearch** subroutine, **hsearch** subroutine, **tsearch** subroutine, **qsort** subroutine.

Donald E. Knuth's *The Art of Computer Programming*, Volume 3, 6.1, Algorithm S. This book was published in Reading, Massachusetts by Addison-Wesley, 1981.

lseek Subroutine

Purpose

Moves read–write file pointer.

Library

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

Syntax

#include <sys/types.h>
#include <unistd.h>

off_t lseek (*FileDescritpor*, *Offset*, *Whence*)
int *FileDescriptor*;
off_t *Offset*;
int *Whence*;

Description

The **lseek** subroutine sets the file pointer for the open file specified by the *FileDescriptor* parameter.

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. The resulting file position may also be negative.

Whence — Specifies how to interpret the *Offset* in setting the file pointer associated with the *FileDescriptor* parameter, as follows:

- **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 the **lseek** system call fails, a value of –1 is returned and the global variable **errno** is set to indicate the error.

Error Codes

The **lseek** subroutine fails 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 *Whence* parameter is an invalid value.

EINVAL The resulting offset would be negative.

Implementation Specifics

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

Related Information

The **fcntl** subroutine, **fseek, rewind, ftell, fgetpos, fsetpos** subroutines, **open** subroutine, **read** subroutine, **write** subroutine.

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

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** header file and contains the following members:

```
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;}
```

The **lv_id** specifies the logical volume to be changed. The **lvname** specifies either the full path name of the logical volume, or a single file name that must reside in the /**dev** directory, (i.e., **rhd1**). This field must be a null–terminated string of from 1 to **LVM_NAMESIZ** bytes, including the null byte, and must be the name of a raw/character device. If a raw/character device is not specified for the **lvname** field, 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 Logical Volume Manager will return the **LVM_NOTCHARDEV** error code. The **maxsize** field 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. The **permissions field** specifies the permission assigned to the logical volume, either read only, or read/write, and the **bb_relocation** field specifies if bad block relocation is desired. The **mirror_policy** field specifies how the copies of the logical partition should be written. This field can be either **LVM_SEQUENTIAL** or **LVM_PARALLEL**. The **write_verify** field specifies if writes to the logical volume should be checked for successful completion. The values for this field are either **LVM_VERIFY** or **LVM_NOVERIFY**. Any other fields in the parameter list that are not to be changed should either contain a zero (0), or be set to null if they are pointers.

The **mirwrt_consist** field tells whether mirror write consistency recovery will be performed for this logical volume. The Logical Volume Manger always insures data consistency among mirrored copies of a logical volume during normal I/O processing. For every write to a logical volume, the Logical Volume Manager 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 Logical Volume Manager 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.

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.

Parameter

Changelv	A pointer to the **changelv** structure.

Return Value

Upon successful completion, a value of 0 is returned.

Error Codes

If the **changelv** subroutine fails, then it returns one of the following values.

LVM_OFFLINE	A routine that requires a volume group to be on–line has encountered one that is off–line.
LVM_INVALID_PARAM	A field in the **changelv** structure is invalid or the pointer to the **changelv** structure is invalid.
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_MAPFSHMAT	An error occurred while trying to attach the mapped file.
LVM_MAPFRDWR	An error occurred while trying to read or write the mapped file.
LVM_DALVOPN	The volume group reserved logical volume could not be opened.
LVM_LVOPEN	The logical volume was open. It must be closed to change the **permissions**, **bb_relocation**, **write_verify**, **mirror_policy,** or **mirwrt_consist** fields.
LVM_INV_DEVENT	The logical volume device entry is invalid and cannot be checked to determine if it is raw.
LVM_ALLOCERR	A memory allocation error occurred.
LVM_NOTCHARDEV	The device is not a raw/character device.
LVM_INVALID_MIN_NUM	An invalid minor number was received.

LVM_LVEXIST A logical volume already exists with the name 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_WRTDAERR An error occurred while trying to write the volume group descriptor area to the logical volume.

Implementation Specifics

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

Related Information

The **lvm_querylv** subroutine, **lvm_varyonvg** subroutine.

Logical Volume Programming Overview in *General Programming Concepts.*

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

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** header file and contains the following members:

```
struct changepv{
   struct unique_id vg_id;
   struct unique_id pv_id;
   long rem_ret;
   long allocation;}
```

The **lvm_changepv** subroutine changes the state of the physical volume specified by the **pv_id** field. The **rem_ret** field should be set to **LVM_REMOVEPV** to temporarily remove the physical volume from the volume group, or **LVM_RETURNPV** to return 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 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 and therefore 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 zero 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 on–line, or an error is returned.

Parameter

Changepv	Pointer to the **changepv** structure.

Return Value

Upon successful completion, a value of 0 is returned.

Error Codes

If the **lvm_changepv** subroutine fails, then it returns one of the following values.

LVM_OFFLINE
The volume group containing the physical volume to be changed is off–line and should be on–line.

LVM_INVALID_PARAM
A field in the **changepv** structure is invalid, or the pointer to the **changepv** structure is invalid.

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_MAPFSHMAT
An error occurred while trying to attach the mapped file.

LVM_MAPFRDWR
An error occurred while trying to read or write the mapped file.

LVM_DALVOPN
The volume group reserved logical volume could not be opened.

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_INV_DEVENT
The device entry for the physical volume is invalid and cannot be checked to determine if it is raw.

LVM_NOTCHARDEV
The device specified is not a character/raw device.

LVM_WRTDAERR
An error occurred while trying to write the volume group descriptor area to the physical volume.

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_BADBBDIR
The bad block directory on the physical volume could not be read from and/or written to.

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_LVMRECERR
The lvm record could not be read or written.

LVM_PVDAREAD
An error occurred while trying to read the volume group descriptor area from the specified physical volume.

Implementation Specifics

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

Related Information

The **lvm_querypv** subroutine.

Logical Volume Programming Overview in *General Programming Concepts.*

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

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** header file and contains the following members:

```
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{
    unsigned long    word1;
    unsigned long    word2;
    unsigned long    word3;
    unsigned long    word4;
}
```

The **lvname** field 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 (e.g., **rhd1**). All name fields must be null–terminated strings of from 1 to **LVM_NAMESIZ** bytes, including the null byte. If a raw/character device is not specified for the **lvname** field, 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 Logical Volume Manager will return the **LVM_NOTCHARDEV** error code. The **vg_id** field specifies the unique ID of the volume group that will contain the logical volume. The **minor_num** field must be in the range from 1 to **maxlvs**. The **maxlvs** field is set when a volume group is created and is returned by the **lvm_queryvg** subroutine. The **maxsize** field is the maximum size in logical partitions for the logical volume and must be in the range of 1 to **LVM_MAXLPS**. The **mirror_policy** field specifies how the physical copies will be written. The **mirror_policy** 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. The **permissions** field indicates read/write or read only permission for the logical volume, and the **bb_relocation** field indicates that bad block relocation is desired. The **write_verify** field indicates that writes to the logical volume are to be verified as successful.

The **mirwrt_consist** field tells whether mirror write consistency recovery will be performed for this logical volume.

The Logical Volume Manger always insures data consistency among mirrored copies of a logical volume during normal I/O processing. For every write to a logical volume, the Logical Volume Manager 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 Logical Volume Manager 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.

Values for the **mirror_policy** field:

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.

Values for the **permissions** field:

LVM_RDONLY Create the logical volume with read only permission.

LVM_RDWR Create the logical volume with read/write permission.

Values for the **bb_relocation** field:

LVM_RELOC Bad block relocation is desired.

LVM_NORELOC Bad block relocation is not desired.

Values for the **write_verify** field:

LVM_VERIFY Write verification is desired.

LVM_NOVERIFY Write verification is not desired.

Values for the **mirwrt_consist** field:

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.

Parameter

Createlv	A pointer to the **createlv** structure.

Return Value

Upon successful completion, a value of 0 is returned.

Error Codes

If the **lvm_createlv** subroutine fails, then it returns one of the following values.

LVM_INV_DEVENT	The logical volume device entry is invalid and cannot be checked to determine if it is raw.
LVM_OFFLINE	A routine that requires a volume group to be on–line has encountered one that is off–line.
LVM_VGFULL	The volume group that the logical volume was requested to be a member of already has the maximum number of logical volumes.
LVM_INVALID_PARAM	A field in the **createlv** structure is invalid, or the pointer to the **createlv** structure is invalid.
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_MAPFSHMAT	An error occurred while trying to attach the mapped file.
LVM_MAPFRDWR	An error occurred while trying to read or write the mapped file.
LVM_DALVOPN	The descriptor area logical volume could not be opened.
LVM_INVALID_MIN_NUM	A minor number passed into the routine is invalid.
LVM_LVEXIST	A logical volume already exists with the name passed into the routine.
LVM_NOTCHARDEV	The lvname name given does not represent a raw/character device.
LVM_ALLOCERR	A memory allocation error has occurred.

Implementation Specifics

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

Related Information

The **lvm_extendlv** subroutine, **lvm_varyonvg** subroutine, **lvm_querylv** subroutine, **lvm_queryvg** subroutine.

Logical Volume Programming Overview in *General Programming Concepts.*

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

The **lvm_createvg** subroutine is used to create a new volume group and to install 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** header 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;
       };
```

The **kmid** field is the module id which 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.

The **vgname** field is the character special file name, which is either the full path name or a file name that resides in the /**dev** directory (e.g., **rvg13**), of the volume group device. This device is actually a logical volume with minor number 0 (zero), which is reserved for use by the Logical Volume Manager.

The **vg_major** field is the major number for the volume group which is to be created.

The **pvname** field is the character special file name, which is either the full path name or a single file name that resides in the /**dev** directory (e.g., **rhdisk0**) of the physical volume being installed in the new volume group.

The **maxlvs** field is the maximum minor number, which will be allowed for a logical volume in the volume group. The range is 1 to **LVM_MAXLVS**.

The **ppsize** field 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 **ppsize**.

The **vgda_size** field is the number of 512 byte blocks which are to be reserved for one copy of the volume group descriptor area. The range is **LVM_MINVGDASIZ** to **LVM_MAXVGDASIZ**. Twice this amount of space will be reserved on each physical volume in the volume group so that two copies of the volume group descriptor area may be saved when needed.

The **override** field specifies whether or not the **LVM_VGMEMBER** error code should be ignored. If the **override** field is **TRUE**, the Logical Volume Manager will create 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 Logical Volume Manager will return 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 varied off. If the **LVM_MEMACTVVG** or **LVM_VGMEMBER** error code is returned, the **vg_id** field will contain the ID of the volume group that the specified physical volume is a member of.

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 will contain 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 Logical Volume Manager to manage the logical volumes and physical volumes in the volume group.

Parameter

Createvg	Pointer to the **createvg** structure.

Return Value

Upon successful completion, a value of 0 is returned.

Error Codes

If the **lvm_createvg** subroutine fails, then it returns one of the following values.

LVM_INV_DEVENT	The device entry for the physical volume is invalid and cannot be checked to determine if it is raw.
LVM_NOTCHARDEV	The device specified is not a character/raw device.
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_INVALID_PARAM	A field in the **createvg** structure is invalid.
LVM_PVOPNERR	The physical volume device could not be opened.

LVM_LVMRECERR The LVM record, which contains information about the volume group descriptor area, could not be read or could not be written.

LVM_RDPVID The record, which contains the physical volume id, could not be read.

LVM_MEMACTVVG The physical volume specified is a member of another volume group that is varied on. This is returned only when the **override** field is set to **TRUE**.

LVM_WRTDAERR An error occurred while trying to write the volume group descriptor area to the physical volume.

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_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_ALLOCERR A memory allocation error occurred.

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_DALVOPN The volume group reserved logical volume could not be opened.

Implementation Specifics

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

Related Information

The **lvm_varyonvg** subroutine.

Logical Volume Programming Overview in *General Programming Concepts*.

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 on–line, 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.

Parameter

Lv_id	Specifies the logical volume to be deleted.

Return Value

Upon successful completion, a value of 0 is returned.

Error Codes

If the **lvm_deletelv** subroutine fails, then it returns one of the following values:

LVM_OFFLINE	A routine that requires a volume group to be on–line has encountered one that is off–line.
LVM_LVOPEN	An open logical volume was encountered when it should be closed.
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_NODELLV	The logical volume cannot be deleted because there are existing logical partitions.
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_MAPFSHMAT	An error occurred while trying to attach the mapped file.
LVM_MAPFRDWR	An error occurred while trying to read or write the mapped file.

LVM_DALVOPN	The volume group reserved logical volume could not be opened.
LVM_INVALID_MIN_NUM	An invalid minor number was received.
LVM_ALLOCERR	A memory allocation error occurred.
LVM_NOTCHARDEV	The device specified is not a character/raw device.
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_WRTDAERR	An error occurred while trying to write the volume group descriptor area to the physical volume.
LVM_INV_DEVENT	The device entry for the logical volume is invalid and cannot be checked to determine if it is raw.

Implementation Specifics

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

Related Information

The **lvm_varyonvg** subroutine.

Logical Volume Programming Overview in *General Programming Concepts*.

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.

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

Upon successful completion, one of the following positive return codes will be returned.

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, one of the following negative error codes will be returned.

LVM_OFFLINE	The volume group which contains the physical volume to be deleted is off–line and should be on–line.
LVM_INVALID_PARAM	An invalid parameter was passed into the routine.
LVM_PARTFND	This routine cannot delete the specified physical volume because it contains physical partitions allocated to a logical volume.

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_MAPFSHMAT	An error occurred while trying to attach the mapped file.
LVM_MAPFRDWR	An error occurred while trying to read or write the mapped file.
LVM_DALVOPN	The descriptor area logical volume could not be opened.
LVM_PVOPNERR	The physical volume device could not be opened.
LVM_LVMRECERR	The lvm record could not be read or written.
LVM_ALLOCERR	A memory allocation error occurred.
LVM_NOTCHARDEV	The physical volume to be deleted does not have a raw device entry.
LVM_INV_DEVENT	The physical volume specified has an invalid device entry and cannot be checked to determine if it is raw.
LVM_WRTDAERR	An error occurred while trying to write the volume group descriptor area to the physical volume.
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 AIX Base Operating System (BOS) Runtime.

Related Information

The **lvm_deletelv** subroutine, **lvm_varyonvg** subroutine, **lvm_reducelv** subroutine, **lvm_migratepp** subroutine, **lvm_queryvg** subroutine.

Logical Volume Programming Overview in *General Programming Concepts.*

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

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 **lvm.h** header file and contains the following members:

```
struct ext_redlv{
     long       size;
     struct pp *parts;
  }
 struct pp {
     struct unique_id pv_id;
     long             lp_num;
     long             pp_num;
  }
```

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 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 zero; if it is, an error will be returned. Within the **pp** structure is a **lp_num** field which is the number of the logical partition that you are extending. This number should be in the range of 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 be in the range of 1 to **LVM_MAXLPS**. Also in the **pp** structure are the **pp_num** and **pv_id**. The **pp_num** field is the number of the physical partition to be allocated as a copy of the logical partition. This number should be in the range of 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 is in the range 1 to **LVM_MAXPPS**). The physical partition specified by the **pp_num** should have a state of **LVM_PPFREE** (i.e., should not 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 extended. The volume group should be varied on, or an error is returned.

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 3 copies (physical partitions) can be allocated to the same logical partition, and an error will be returned if an attempt is made to add more. It is also possible to have entries with a valid **lp_num** 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 to simply 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 will be ignored since there were no entries for them in the array.

Parameters

Extendlv	Pointer to the **ext_redlv** structure.
Lv_id	Pointer to the **lv_id** structure, which specifies the logical volume to extend.

Return Value

Upon successful completion, a value of 0 is returned.

Error Codes

If the **lvm_extendlv** subroutine fails, then it returns one of the following values.

LVM_OFFLINE The volume group is off–line and should be on–line.

LVM_INVALID_PARAM Either 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 invalid.

LVM_NOALLOCLP The logical partition specified already has three copies.

LVM_LPNUM_INVAL A logical partition number passed in is invalid.

LVM_PPNUM_INVAL A physical partition number passed in is invalid.

LVM_PVSTATE_INVAL A physical volume id sent in specifies a physical volume with a state of **LVM_PVNOALLOC**.

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_MAPFSHMAT An error occurred while trying to attach the mapped file.

LVM_MAPFRDWR An error occurred while trying to read or write the mapped file.

LVM_DALVOPN The volume group reserved logical volume could not be opened.

LVM_INVALID_MIN_NUM An invalid minor number was received.

LVM_ALLOCERR A memory allocation error occurred.

LVM_INV_DEVENT The device entry for the physical volume is invalid and cannot be checked to determine if it is raw.

LVM_NOTCHARDEV The device specified is not a character/raw device.

LVM_INRESYNC The logical partition to be extended is being resynced, and cannot be extended while the resync is in progress.

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_WRTDAERR An error occurred while trying to write the volume group descriptor area to the physical volume.

Implementation Specifics

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

Related Information

The **lvm_changelv** subroutine, **lvm_createlv** subroutine, **lvm_reducelv** subroutine, **lvm_varyonvg** subroutine.

Logical Volume Programming Overview in *General Programming Concepts.*

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

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** header file and is defined as follows:

```
struct installpv
{
  char *pvname;
  struct unique_id vg_id;
  short int override;
  struct unique_id out_vg_id;
};
```

The **pvname** field is the character special file name, which can be either a full path name or a single file name that resides in the /**dev** directory (e.g., **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 of 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 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 Logical Volume Manager will return the **LVM_NOTCHARDEV** error code.

The **override** field specifies whether or not the **LVM_VGMEMBER** error code should be ignored. If the **override** field is **TRUE**, the Logical Volume Manager will install 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, the **LVM_MEMACTVVG** error code is returned. If the **override** field is **FALSE**, the **LVM_VGMEMBER** error code is returned if the physical volume belongs to another volume group wheter 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 no matter what the setting is of the **override** field.

The **out_vg_id** field contains the ID of the volume group that the physical volume is a member of if either the **LVM_MEMACTVVG** or the **LVM_VGMEMBER** error code is returned.

Each physical volume installed into a volume group will contain 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 Logical Volume Manager to manage the logical volumes and physical volumes in the volume group.

Parameter

Installpv Pointer to the **installpv** structure.

Return Values

Upon successful completion, a value of 0 is returned.

Error Codes

If the **lvm_installpv** subroutine fails, then it returns one of the following negative values.

LVM_ALRDYMEM The physical volume is already a member of the specified volume group.

LVM_OFFLINE A volume group specified is off–line. It must be varied–on to perform this operation.

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_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_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_PVOPNERR The physical volume device could not be opened.

LVM_LVMRECERR The LVM record, which contains information about the volume group descriptor area, could not be read or could not be written.

LVM_RDPVID The record which contains the physical volume id could not be read.

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_DALVOPN	The volume group reserved logical volume could not be opened.
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_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_ALLOCERR	A memory allocation error occurred.
LVM_INVALID_PARAM	An invalid parameter was passed into the routine.
LVM_NOTCHARDEV	The device specified is not a character/raw device.
LVM_INV_DEVENT	The device entry for the physical volume is invalid and cannot be checked to determine if it is raw.
LVM_MEMACTVVG	The physical volume specified is a member of another volume group that is varied on. This is returned when the **override** field is **TRUE**.
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_WRTDAERR	An error occurred while trying to write the volume group descriptor area to the physical volume.

Implementation Specifics

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

Related Information

The **lvm_varyonvg** subroutine.

Logical Volume Programming Overview in *General Programming Concepts*.

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

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** header file and contains the following members:

```
struct migratepp{
       struct unique_id vg_id;
       long    oldpp_num;
       long    newpp_num;
       struct unique_id oldpv_id;
       struct unique_id newpv_id;
   }
```

Parameter

Migratepp	Points to the **migratepp** structure.

Return Value

Upon successful completion of the **lvm_migratepp** subroutine a value of 0 is returned.

Error Codes

If the **lvm_migratepp** subroutine fails, then it returns one of the following values.

LVM_NOTSYNCED	The resync involving the physical partitions of the **migratepp** call was not complete.
LVM_OFFLINE	The volume group is off–line and should be on–line.
LVM_INVALID_PARAM	One of the parameters passed in did not have a valid value.
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_MAPFSHMAT	An error occurred while trying to attach the mapped file.
LVM_MAPFRDWR	An error occurred while trying to read or write the mapped file.
LVM_DALVOPN	The volume group reserved logical volume could not be opened.
LVM_NOALLOCLP	The logical partition specified already has three copies.
LVM_LPNUM_INVAL	A logical partition number is invalid.
LVM_PPNUM_INVAL	A physical partition number is invalid.
LVM_PVSTATE_INVAL	A physical volume specified has a state of **LVM_PVNOALLOC**.
LVM_ALLOCERR	A memory allocation error occurred.
LVM_NOTCHARDEV	A device is not a raw/character device.
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_INVLPRED	A reduction was requested that would leave a logical partition with no good copies.
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_WRTDAERR	An error occurred while trying to write the volume group descriptor area to the physical volume.
LVM_MIGRATE_FAIL	The migration failed due to an error in the resync phase.
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.

Implementation Specifics

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

Related Information

The **lvm_querypv** subroutine, **lvm_varyonvg** subroutine.

Logical Volume Programming Overview in *General Programming Concepts*.

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

The **lvm_querylv** subroutine returns information for the logical volume specified by the *Lv_id* parameter.

The **querylv** structure, found in the **lvm.h** header 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;
   }
```

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 most recent, in–memory copy of the descriptor area. If the query is done this way, the volume group does not have to be on–line; however, the data returned may reflect a back level descriptor area instead of the most recent one. 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 (e.g., **rhdisk1**). This parameter must be a **null** terminated string of from 1 to **LVM_NAMESIZ** bytes, including the null byte and must represent a raw device entry. If a raw/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 Logical Volume Manager will return the **LVM_NOTCHARDEV** error code.

If a pvname is specified, only the **minor_num** portion of the *Lv_id* parameter need be supplied. The Logical Volume Manager will fill in the **vg_id** portion and return it to the user. If the user wishes to query from the Logical Volume Manager'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 will be returned.

Note: As long as the *Pvname* is **not NULL**, the Logical Volume Manager will attempt a query from a physical volume and **not** its in–memory copy of data.

In addition to the *Pvname*, 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 Logical Volume Manager will allocate the space needed for the **querylv** structure and return the structure's address in the pointer variable passed in by the user.

The **lv_state** field specifies the current state of the logical volume and may 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.

The **currentsize** field is the current size in logical partitions of the logical volume. The **ppsize** specifies the size of the physical partitions of all physical volumes in the volume group. The size in bytes of every physical partition is 2 ** ppsize.

The **permissions** field specifies the permission assigned to the logical volume and may be one of the following:

LVM_RDONLY Access to this logical volume is read only.

LVM_RDWR Access to this logical volume is read/write.

The **bb_relocation** field specifies if bad block relocation is desired and will be one of the following:

LVM_RELOC Bad blocks will be relocated.

LVM_NORELOC Bad blocks will not be relocated.

The **write_verify** field specifies if write verification for the logical volume is desired and will be one of the following:

LVM_VERIFY Write verification is performed on all writes to the logical volume.

LVM_NOVERIFY Write verification is not performed for this logical volume.

The **mirwrt_consist** field tells whether mirror write consistency recovery will be performed for this logical volume.

The Logical Volume Manger always insures data consistency among mirrored copies of a logical volume during normal I/O processing. For every write to a logical volume, the Logical Volume Manager 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 Logical Volume Manager 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.

Values for the **mirwrt_consist** field:

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.

The **open_close** field specifies if the logical volume is opened or closed.

LVM_QLVOPEN The logical volume is opened by one or more processes.

LVM_QLV_NOTOPEN The logical volume is closed.

The **mirrors** field is an array of pointers to partition map lists (physical volume id, logical partition number, and physical partition number for each copy of the logical partitions for the logical volume). 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.

Parameters

Lv_id Pointer to an **lv_id** structure that specifies the logical volume to query.

Querylv Address of a pointer to the **querylv** structure.

Pvname Name of the physical volume from which to use the volume group descriptor for the query. (Can also be **NULL**).

Return Value

Upon successful completion, a value of 0 is returned.

Error Codes

If the **lvm_querylv** subroutine fails, then it returns one of the following values.

LVM_ALLOCERR The subroutine could not allocate enough space for the complete buffer.

LVM_OFFLINE The volume group containing the logical volume to query was off–line.

LVM_INVALID_PARAM An invalid parameter was passed into the routine.

LVM_INVALID_MIN_NUM The minor number of the logical volume is invalid.

LVM_NOTCHARDEV The physical volume name given does not represent a raw/character device.

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.

If the query is done from the varied on volume group's current volume group descriptor area, then one of the following negative return codes may be returned.

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_MAPFSHMAT	An error occurred while trying to attach the mapped file.
LVM_DALVOPN	The volume group reserved logical volume could not be opened.

If a physical volume name has been passed, requesting that the query come from a specific physical volume, then one of the following negative return codes may be returned.

LVM_PVOPNERR	The physical volume device could not be opened.
LVM_LVMRECERR	The LVM record, which contains information about the volume group descriptor area, could not be read.
LVM_PVDAREAD	An error occurred while trying to read the volume group descriptor area from the specified physical volume.
LVM_NOTVGMEM	The physical volume specified is not a member of a volume group.
LVM_NOPVVGDA	There are no volume group descriptor areas on the physical volume specified.
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.
LVM_BADBBDIR	The bad bock directory could not be read or written.

Implementation Specifics

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

Related Information

The **lvm_varyonvg** subroutine, **lvm_createlv** subroutine.

Logical Volume Programming Overview in *General Programming Concepts.*

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

The **lvm_querypv** subroutine returns information on the physical volume specified by the *Pv_id* parameter.

The **querypv** structure, defined in the **lvm.h** header file, contains the following members:

```
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;
     struct unique_id fst_alt_vol;
     long fst_alt_part;
     struct unique_id snd_alt_vol;
     long snd_alt_part;
   }
```

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 most recent, in–memory copy of the descriptor area. If the query is done this way, the volume group does not have to be on–line; however, the data returned may reflect a back level descriptor area instead of the most recent one. 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 (.e.g., **rhdisk1**). This field must be a null terminated string of from 1 to **LVM_NAMESIZ** bytes, including the null byte, and represent a raw/character device. If a raw/character device is not specified for the **pvname** field, 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 Logical Volume Manager will return the

LVM_NOTCHARDEV error code. If a *Pvname* is specified, the vg_id will be returned by the Logical Volume Manager through the *Vg_id* parameter passed in by the user. If the user wishes to query from the Logical Volume Manager'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 will be returned. **NOTE** As long as the *Pvname* is not NULL, the Logical Volume Manager 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 Logical Volume Manager will allocate enough space for the **querypv** structure and return the address to this structure in the *Querypv* pointer passed in.

The **pv_state** field will contain the current state of the physical volume. The **ppsize** field 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**. The **pp_count** field will contain the total number of physical partitions on the physical volume. The **alloc_ppcount** field will contain the number of allocated physical partitions on the physical volume. The **pvnum_vgdas** field contains the number of volume group descriptor areas (0, 1, or 2) that are on the specified physical volume. The **pp_map** field is a pointer 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**, the ID of the logical volume that it is a member of. Also in the struct **pp_map** array are 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. The **fst_alt_vol** and **fst_alt_part** fields will contain zeroes if the logical partition has only one physical copy. The **snd_alt_vol** and **snd_alt_part** fields will contain zeroes if the logical partition has only one or two physical copies.

Parameters

Vg_id	Pointer to a **unique_id** structure that specifies the volume group of which the physical volume to query is a member.
Pv_id	Pointer to a **unique_id** structure that specifies the physical volume to query.
Querypv	Address of a pointer to a **querypv** structure.
Pvname	Name of physical volume from which to use the volume group descriptor area for the query. This can also be **NULL**.

Return Value

Upon successful completion, a value of 0 is returned.

Error Codes

If the **lvm_querypv** subroutine fails, then it returns one of the following negative return codes.

LVM_INV_DEVENT	The device entry for the physical volume is invalid and cannot be checked to determine if it is raw.

LVM_ALLOCERR The routine cannot allocate enough space for a complete buffer.

LVM_OFFLINE The volume group specified is off–line and should be on–line.

LVM_INVALID_PARAM An invalid parameter was passed into the routine.

If the query is done from the varied–on volume group's current volume group descriptor area, then one of the following negative return codes may be returned.

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_MAPFSHMAT An error occurred while trying to attach the mapped file.

LVM_DALVOPN The volume group reserved logical volume could not be opened.

If a physical volume name has been passed, requesting that the query come from a specific physical volume, then one of the following negative return codes may be returned.

LVM_PVOPNERR The physical volume device could not be opened.

LVM_LVMRECERR The LVM record, which contains information about the volume group descriptor area, could not be read.

LVM_PVDAREAD An error occurred while trying to read the volume group descriptor area from the specified physical volume.

LVM_NOTVGMEM The physical volume is not a member of a volume group.

LVM_NOPVVGDA There are no volume group descriptor areas on this physical volume.

LVM_NOTCHARDEV A device is not a raw/character device.

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.

LVM_BADBBDIR The bad bock directory could not be read or written.

Implementation Specifics

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

Related Information

The **lvm_varyonvg** subroutine.

Logical Volume Programming Overview in *General Programming Concepts.*

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

The **lvm_queryvg** subroutine returns information on the volume group specified by the *Vg_id* parameter.

The **queryvg** structure, found in the **lvm.h** header file, contains the following members:

```
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];
 }
```

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 most recent, in–memory copy of the descriptor area. If the query is done this way, the volume group does not have to be on–line; however, the data returned may reflect a back level descriptor area instead of the most recent one. 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 (e.g., **rhdisk1**). The name must represent a raw device. If a raw/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 Logical Volume Manager will return 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 Logical Volume Manager will return the vg_id to the user through the *Vg_id* pointer passed in. If the user wishes to query from the Logical Volume Manager'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 will be returned.

Note: As long as the pvname is **not NULL**, the Logical Volume Manager 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 logical volume manager will allocate enough space for the structure and return the address of the completed structure in the *Queryvg* parameter passed in by the user.

The **maxlvs** field is the maximum number of logical volumes allowed in the volume group. The **ppsize** field 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. The **freespace** field contains the number of free physical partitions in this volume group. The number of logical volumes and the number of physical volumes will be returned in the **num_lvs** and **num_pvs** fields, respectively. The **total_vgdas** field specifies the total number of volume group descriptor areas for the entire volume group. The **lvs** field is a pointer to an array of unique ids, names. and states of the logical volumes in the volume group. The **pvs** field is a pointer 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.

Parameters

Vg_id — Pointer to a **unique_id** structure that specifies the volume group to be queried.

Queryvg — 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 Value

Upon successful completion, a value of 0 is returned.

Error Codes

If the **lvm_queryvg** subroutine fails, then it returns one of the following negative return codes.

LVM_ALLOCERR — The subroutine cannot allocate enough space for a complete buffer.

LVM_OFFLINE — The volume group is off–line and should be on–line.

LVM_INVALID_PARAM — An invalid parameter was passed into the routine.

If the query is done from the varied–on volume group's current volume group descriptor area, then one of the following negative return codes may be returned.

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_MAPFSHMAT	An error occurred while trying to attach the mapped file.
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_NOTCHARDEV	A device is not a raw/character device.

If a physical volume name has been passed, requesting that the query come from a specific physical volume, then one of the following negative return codes may be returned.

LVM_PVOPNERR	The physical volume device could not be opened.
LVM_LVMRECERR	The LVM record, which contains information about the volume group descriptor area, could not be read.
LVM_PVDAREAD	An error occurred while trying to read the volume group descriptor area from the specified physical volume.
LVM_NOTVGMEM	The physical volume is not a member of a volume group.
LVM_NOPVVGDA	There are no volume group descriptor areas on this physical volume.
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.
LVM_BADBBDIR	The bad bock directory could not be read or written.

Implementation Specifics

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

Related Information

The **lvm_varyonvg** subroutine.

Logical Volume Programming Overview in *General Programming Concepts.*

lvm_queryvgs Subroutine

Purpose

Queries the volume groups of the system and returns information for the volume groups that are on–line.

Library

Logical Volume Manager Library (**liblvm.a**)

Syntax

#include <lvm.h>

int lvm_queryvgs (*Queryvgs*, *Kmid*)
  struct queryvgs ***Queryvgs*;
  mid_t *Kmid*;

Description

The **lvm_queryvgs** subroutine returns the volume group ids and major numbers for all volume groups in the system that are on–line.

The caller passes the address of a pointer to a **queryvgs** structure and the logical volume manager 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];
 }
```

The **num_vgs** field contains the number of on–line volume groups on the system. The **vgs** is an array of the volume group IDs and major numbers of all on–line volume groups in the system.

Parameters

Queryvgs — Address of a pointer that is of the type struct **queryvgs**.

Kmid — Identifies the address of the entry point of the logical volume device driver module.

Return Value

Upon successful completion, a value of 0 is returned.

Error Codes

If the **lvm_queryvgs** subroutine fails, then it returns one of the following values.

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 AIX Base Operating System (BOS) Runtime.

Related Information

The **lvm_varyonvg** subroutine.

Logical Volume Programming Overview in *General Programming Concepts.*

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

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 a volume group that is on–line. 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 will 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** header 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 struct **pps**. 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 zero; if it is, an error will be returned. Within the **pp** structure is a **lp_num** field which is the number of the logical partition that you are reducing. This number should be in the range of 1 to 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** is the number of the physical partition to be deallocated as a copy of the logical partition. This number should be in the range of 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** should have a state of **LVM_PPALLOC** (i.e., 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

Reducelv	Pointer to the **ext_redlv** structure.
Lv_id	Specifies the logical volume to be reduced.

Return Value

Upon successful completion, a value of 0 is returned.

Error Codes

If the **lvm_reducelv** subroutine fails, then it returns one of the following values.

LVM_OFFLINE	The volume group is off–line and should be on–line.
LVM_INVALID_PARAM	One of the parameters passed in is invalid, or one of the fields in the structures pointed to by one of the parameters is invalid.
LVM_LVOPEN	The logical volume to be reduced was open and should be closed.
LVM_PPNUM_INVAL	A physical partition number passed in is invalid.
LVM_LPNUM_INVAL	A logical partition number passed in is invalid.
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_MAPFSHMAT	An error occurred while trying to attach the mapped file.
LVM_MAPFRDWR	An error occurred while trying to read or write the mapped file.
LVM_INVALID_MIN_NUM	An invalid minor number was received.
LVM_ALLOCERR	A memory allocation error occurred.

LVM_DALVOPN	The volume group reserved logical volume could not be opened.
LVM_INVLPRED	The reduction can not be completed because a logical partition would exist with only stale copies remaining.
LVM_INV_DEVENT	The device entry for the physical volume is invalid and cannot be checked to determine if it is raw.
LVM_NOTCHARDEV	The device specified is not a character/raw device.
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_WRTDAERR	An error occurred while trying to write the volume group descriptor area to the physical volume.

Implementation Specifics

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

Related Information

The **lvm_extendlv** subroutine, **lvm_createlv** subroutine, **lvm_deletelv** subroutine, **lvm_resynclp** subroutine, **lvm_resynclv** subroutine, **lvm_resyncpv** subroutine.

Logical Volume Programming Overview in *General Programming Concepts.*

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***)**
struct Lv_id **Lv_id*;
long *Lp_num*;

Description

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.

Parameters

Lv_id	Specifies the logical volume that contains the logical partition needing resynchronization.
Lp_num	The logical partition number within the logical volume to be resynchronized.

Return Value

Upon successful completion the **lvm_resynclp** subroutine returns a value of 0.

Error Codes

If the **lvm_resynclp** subroutine fails, then it returns one of the following values.

LVM_NOTSYNCED	The logical partition was not completely resynced.
LVM_OFFLINE	The volume group is off–line and should be on–line.
LVM_INVALID_PARAM	One of the fields passed in did not have a valid value.
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_MAPFSHMAT	An error occurred while trying to attach the mapped file.
LVM_MAPFRDWR	An error occurred while trying to read or write the mapped file.
LVM_DALVOPN	The volume group reserved logical volume could not be opened.

LVM_ALLOCERR	A memory allocation error occurred.
LVM_NOTCHARDEV	A device is not a raw/character device.
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_WRTDAERR	An error occurred while trying to write the volume group descriptor area to the physical volume.

Implementation Specifics

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

Related Information

The **lvm_resynclv** subroutine, **lvm_resyncpv** subroutine, **lvm_extendlv** subroutine, **lvm_varyonvg** subroutine.

Logical Volume Programming Overview in *General Programming Concepts.*

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***)**
**struct Lv_id ****Lv_id*;

Description

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.

Parameter

Lv_id	Specifies the logical volume name.

Return Value

Upon successful completion, the **lvm_resynclv** subroutine returns a value of 0.

Error Codes

If the **lvm_resynclv** subroutine fails, then it returns one of the following values.

LVM_OFFLINE	The volume group is off–line and should be on–line.
LVM_INVALID_PARAM	One of the fields passed in did not have a valid value.
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_MAPFSHMAT	An error occurred while trying to attach the mapped file.
LVM_MAPFRDWR	An error occurred while trying to read or write the mapped file.
LVM_DALVOPN	The volume group reserved logical volume could not be opened.
LVM_NOTSYNCED	The logical volume could not be completely resynced.
LVM_ALLOCERR	A memory allocation error occurred.
LVM_NOTCHARDEV	A device is not a raw/character device.
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_WRTDAERR An error occurred while trying to write the volume group descriptor area to the physical volume.

Implementation Specifics

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

Related Information

The **lvm_resyncpv** subroutine, **lvm_resynclp** subroutine, **lvm_varyonvg** subroutine.

Logical Volume Programming Overview in *General Programming Concepts*.

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)*
struct unique_id **Vg_id;*
struct unique_id **Pv_id;*

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 will be returned.

Note: The resync of the physical volume is done by resyncing **entire** logical partitions that any stale physical partitions belong to on the physical volume. Because a complete logical partition is resynced, other physical volumes besides the one specified may be partially or completely resynced.

Parameters

Vg_id	Specifies the volume group that contains the physical volume to be resynced.
Pv_id	Specifies the physical volume.

Return Value

Upon successful completion the **lvm_resyncpv** subroutine returns a value of 0.

Error Codes

If the **lvm_resyncpv** subroutine fails, then it returns one of the following values.

LVM_OFFLINE	The volume group is off–line and should be on–line.
LVM_INVALID_PARAM	One of the fields passed in did not have a valid value.
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_MAPFSHMAT	An error occurred while trying to attach the mapped file.
LVM_MAPFRDWR	An error occurred while trying to read or write the mapped file.

LVM_DALVOPN	The volume group reserved logical volume could not be opened.
LVM_NOTSYNCED	The physical volume could not be completely resynced.
LVM_ALLOCERR	A memory allocation error occurred.
LVM_NOTCHARDEV	A device is not a raw/character device.
LVM_INV_DEVENT	A device has a major number that does not correspond to the volume group being worked in.
LVM_WRTDAERR	An error occurred while trying to write the volume group descriptor area to the physical volume.

Implementation Specifics

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

Related Information

The **lvm_resynclv** subroutine, **lvm_resynclp** subroutine, **lvm_varyonvg** subroutine.

Logical Volume Programming Overview in *General Programming Concepts*.

lvm_varyoffvg Subroutine

Purpose

Varies a volume group off–line.

Library

Logical Volume Manager Library (**liblvm.a**)

Syntax

#include <lvm.h>

int lvm_varyoffvg (*Varyoffvg*)
struct varyoffvg * *Varyoffvg*;

Description

The **lvm_varyoffvg** subroutine varies a specified volume group off–line. All logical volumes in the volume group to be varied off–line **must be closed**.

The **varyoffvg** structure pointed to by the *Varyoffvg* parameter is found in the **lvm.h** header file and defined as follows:

```
struct varyoffvg
       {
        struct unique_id vg_id;
        long lvs_only;
       }  * Varyoffvg;
```

The **lvm_varyoffvg** subroutine varies the volume group specified by the **vg_id** field off–line.

The **lvs_only** flag is used to indicate whether the volume group is to be varied–off entirely or whether system management commands, which act on the volume group, will still be permitted. If the **lvs_only** flag is **TRUE**, then all logical volumes in the volume group will be varied–off, but the volume group will still be available for system management commands which act on the volume group. If the **lvs_only** flag 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**.

Parameter

Varyoffvg	Pointer to the **varyoffvg** structure.

Return Value

Upon successful completion, a value of 0 is returned.

Error Codes

If the **lvm_varyoffvg** subroutine fails, then it returns one of the following negative values.

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_MAPFSHMAT	An error occurred while trying to attach the mapped file.
LVM_MAPFRDWR	An error occurred while trying to write to the mapped file.
LVM_ALLOCERR	A memory allocation error occurred.
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 major number in the mapped file is invalid.
LVM_OFFLINE	The volume group specified is off–line. It must be varied–on to perform this operation.
LVM_INV_DEVENT	The device entry for the physical volume is invalid and cannot be checked to determine if it is raw.
LVM_NOTCHARDEV	The device specified is not a character/raw device.

Implementation Specifics

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

Related Information

The **lvm_varyonvg** subroutine.

Logical Volume Programming Overview in *General Programming Concepts.*

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

The **lvm_varyonvg** subroutine varies the specified volume group on–line. The **lvm_varyonvg** subroutine contacts the physical volumes in the volume group and performs recovery of the volume group descriptor area if necessary.

The **varyonvg** structure pointed to by the *Varyonvg* parameter is found in the **lvm.h** header file and 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;
          };
```

The **kmid** field is the module id that identifies the entry point of the logical volume device driver module.

The **vgname** field is the character special file name, which is either the full path name or a file name that resides in the /**dev** directory (e.g. **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.

The **vg_major** is the major number of the volume group to be varied on.

If the **noopen_lvs** flag 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 any other system management functions can be performed, but opens to the logical volumes in the volume group will not be allowed.

The **auto_resync** is a flag that should contain either **TRUE** or **FALSE**. If **auto_resync** 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** flag is **TRUE**.

The structure **vvg_in** 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 structure **vvg_out** contains output from the **lvm_varyonvg** subroutine to the user that describes the status of the physical volumes in the caller's input list and any additional physical volumes which are found to be in the volume group but were not included in the input list. The **num_pvs** 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 **pvname** field is the character special file name, which is either the full path name or a single file name that resides in the /**dev** directory (e.g., **rhdisk0**) of the physical volume being installed in the new volume group.

The **pv_status** field for each physical volume in the **vvg_out** structure will contain one of the following values if either the volume group is varied on successfully or if the **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 missing from the volume group.

LVM_PVREMOVED — This physical volume has been temporarily removed from the volume group by user request.

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** 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 which are found to be members of the specified volume group, the **pv_status** will contain the physical volume state of either **LVM_PVACTIVE**, **LVM_PVMISSING**, or **LVM_PVREMOVED**. If a physical volume which has the same **pv_id** has appeared previously in the input list, the **pv_status** field will contain **LVM_DUPPVID**. For physical volumes in the list which are not members of the volume group, the **pv_status** will be **LVM_INVPVID**.

In some cases, a physical volume that is a member of the volume group might have a **pv_status** of **LVM_LVMRECNMTCH**. This means that the LVM record on the physical volume has either invalid or non–matching 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** will be **LVM_NONAME** or **LVM_NAMIDNMTCH**. In this case the **pv_id** field will contain the ID of the physical volume, and the **pvname** field will contain a null pointer. An unsuccessful (negative) return code of **LVM_MISSPVNAME** will be returned to the caller unless the subroutine was called with a value of **TRUE** for the **missname_von** flag.

The **pv_status** field for each physical volume in the **vvg_out** structure will contain 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 record or the 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** flag contain a value of **FALSE** for the first call to the **lvm_varyonvg** subroutine since a value of **TRUE** will mean that any physical volume for which a name was not passed in the input list will be given a state of **LVM_PVMISSING**, and 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** flag is **TRUE**, the volume group will be varied on (provided a quorum exists) even if some of the physical volumes in the volume group have a state of **LVM_PVMISSING** or **LVM_PVREMOVED**. If the flag is **FALSE**, the volume group will be varied on only if all physical volumes in the volume group are in the active state (**LVM_PVACTIVE**). The value recommended for this flag is **TRUE**. For any physical volume that has a state of **LVM_PVMISSING** or **LVM_PVREMOVED** when the volume group is varied on, access to that physical volume will not be available through the Logical Volume Manager. 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 will again be available to the Logical Volume Manager, provided that it is not missing at the time.

If the **override** flag is **TRUE**, an attempt will be made to vary on the volume group even if access to a quorum (or majority) of volume group descriptor area copies cannot be obtained. Provided that there is at least one valid copy of the descriptor area, the vary on of the volume group will proceed with the latest available copy of the volume group descriptor area.

The recommended value for the override flag is **FALSE**. Note that if the user chooses to override the **LVM_NOQUORUM** error and artificially force a quorum, the Logical Volume Manger will not guarantee the data integrity of the data contained in the chosen copy of the volume group descriptor area.

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 will be 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 will be defined to the logical volume device driver, and access to those physical volumes will again be available through the Logical Volume Manager.

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 but only needs to pass information for those missing physical volumes that he wishes to attempt to return to the **LVM_PVACTIVE** state.

Parameter

Varyonvg	Pointer to the **varyonvg** structure.

Return Values

Upon successful completion, one or more of the following positive return codes will be returned:

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 which should be checked.

Error Codes

If the **lvm_varyonvg** subroutine does not complete successfully, one of the following negative error codes will be returned:

LVM_NOQUORUM	The volume group could not be varied on because access to a quorum count, or majority, of all volume group descriptor areas could not be 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 will contain the **pv_id**, a null pointer for the **pvname**, and a **pv_status** 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 will be returned only if the **missname_von** flag has a value of **FALSE**; otherwise, the volume group will be varied on if a quorum is obtained.

LVM_MISSINGPV The volume group was not varied on because one of the physical volumes in the volume group has a state of either **LVM_PVMISSING** or **LVM_PVREMOVED**. This error will be returned only if the **misspv_von** flag has a value of **FALSE**; otherwise, the volume group will be varied on if a quorum is obtained.

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, or if the major number given is already in use.

LVM_NOTCHARDEV The device specified is not a character/raw device.

LVM_INV_DEVENT The device entry for a specified device is invalid and cannot be checked to determine if it is raw.

LVM_INVALID_PARAM A field in the **varyongvg** structure is invalid or the pointer structure is invalid.

LVM_MAPFRDWR An error occurred while trying to read or write the mapped file.

LVM_ALLOCERR A memory allocation error has occurred.

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_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.

Implementation Specifics

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

Related Information

The **lvm_varyoffvg** subroutine.

Logical Volume Programming Overview in *General Programming Concepts*.

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*.

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, zero, 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 newline; 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 AIX Base Operating System (BOS) Runtime.

Programs that use the multiple-precision arithmetic functions must link with **libbsd.a**.

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/include/mp.h	include file
/lib/libbsd.a	object code library

Related Information

The **bc** command, **dc** command.

malloc, free, realloc, calloc, mallopt, mallinfo, or alloca Subroutine

Purpose

Provides a memory allocator.

Libraries

Standard C Library (**libc.a**), Berkeley Compatibility Library (**libbsd.a**)

Syntax

#include <malloc.h>

void *malloc (*Size***)**
size_t int *Size*;

char *alloca (*Size***)**
int *Size*;

void free (*Pointer***)**
void **Pointer*;

void *realloc (*Pointer*, *Size***)**
char **Pointer*;
size_t *Size*;

int mallopt (*Command*, *Value***)**
int *Command*
int *Value*;

struct mallinfo mallinfo( )

void *calloc (*NumberOfElements*, *ElementSize***)**
size_t *NumberOfElements*;
size_t *ElementSize*;

Description

The **malloc** subroutine and **free** subroutines provide a simple 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 **malloc** subroutine searches memory for the first contiguous area of free space of at least the number of bytes specified by the *Size* parameter. The search is performed in a circular pattern from the last block of memory allocated or freed. During the search, the subroutine joins adjacent free blocks of memory. If a large enough contiguous area of free space is not found, this subroutine issues an **sbrk** subroutine to get more memory from the system.

The **free** subroutine frees the block of memory pointed to by the *Pointer* parameter for further allocation. The block pointed to by the *Pointer* parameter must have been previously allocated by the **malloc** subroutine. The **free** subroutine does not change the contents of this block of memory. Undefined results occur if the *Pointer* parameter is not a valid pointer.

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 pointer to the block. The contents of the block 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 calls the **malloc** subroutine to enlarge the memory area and moves the data to the new space.

The **realloc** subroutine also works if the *Pointer* parameter points to a block freed since the last call to the **malloc** subroutine, **realloc** subroutine, or **calloc** subroutine.

The **calloc** subroutine allocates space for an array with the number of elements specified by the *NumberOfElements* parameter. Each element is of the size specified by the *ElementSize* parameter. The space is initialized to zeros.

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 **mallopt** subroutine and **mallinfo** subroutine allow tuning the allocation algorithm at execution time. These subroutines are implemented to provide compatibility with System V. Nothing done with **mallopt** affects how memory is allocated by the system. **malloc** performs efficient memory allocation without needing **mallopt**.

The **mallopt** subroutine initiates a mechanism that can be used to allocate small blocks of memory quickly. Using this scheme, a large group (called a holding-block) of these small blocks is allocated at one time. Then, each time a program requests a small amount of memory, a pointer to one of the pre-allocated small blocks is returned. Different holding-blocks are created for different sizes of small blocks and are created when needed. This subroutine allows the programmer to set the following three values to maximize efficient small block allocation for a particular application. The three values are:

grain
The grain of small block sizes. This value determines what range of small block sizes is considered the same size, which influences the number of separate holding-blocks allocated. For example, if the **grain** value is 16 bytes, all small blocks of 16 bytes or less belong to one holding-block and blocks from 17 to 32 bytes belong to another holding-block. Thus, if the **grain** value is too small, space may be wasted because many holding-blocks are created.

number
The number of small blocks in a holding-block. If holding-blocks have many more small blocks than the program is using, space is wasted. If holding-blocks are too small or have too few small blocks in each, performance gain is lost.

size
Below this value, a request to the **malloc** subroutine is filled using the special small block algorithm. Initially this value, which is called MAXFAST, is zero, which means that the small block option is not normally in use by **malloc**.

The values for the *Command* parameter to the **mallopt** subroutine are:

M_GRAIN
Sets the GRAIN value to the *Value* parameter (must be greater than 0). The sizes of all blocks smaller than MAXFAST are considered to be rounded up, to the nearest multiple of GRAIN. The default value for the GRAIN parameter is the smallest number of bytes that allows alignment of any data type. When the GRAIN parameter is set, the *Value* parameter is rounded up to a multiple of the default

M_KEEP Preserves data in a free-block until the next call to the **malloc**, **realloc**, or **calloc** subroutine. This option is provided only for compatibility with the older version of the **malloc** subroutine and is not recommended.

M_MXFAST Sets the MAXFAST value to the value specified by the *Value* parameter. The algorithm allocates all blocks below the size of MAXFAST in large groups and then doles them out very quickly. The default value for MAXFAST is 0.

M_NLBLKS Sets the NUMBLKS value to the *Value* parameter. The aforementioned large groups each contain NUMBLKS blocks. The value for NUMBLKS must be greater than 1. The default value is 100.

The **mallopt** subroutine can be called repeatedly, but parameters cannot be changed after the first small block is allocated from a holding-block. If the **mallopt** subroutine is called again after the first small block is allocated, it returns an error.

The **mallinfo** subroutine can be used during program development to determine the best settings of these parameters for a particular application. It must be called only after some storage is allocated. Information is returned describing space usage. Refer to the **malloc.h** file for details of the **mallinfo** structure.

Parameters

Size Specifies a number of bytes of memory.

Pointer Points to the block of memory that was returned by **malloc** or **calloc**.

Command Specifies a **mallopt** subroutine command.

Value Specifies the value to which M_MXFAST, M_NLBLKS, M_GRAIN, or M_KEEP is to be set.

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** subroutine, **realloc** subroutine, and **calloc** subroutine return a NULL pointer if there is no available memory or if the memory arena has been corrupted by storing outside the bounds of a block. When this happens, the block pointed to by the *Pointer* parameter may be destroyed.

Implementation Specifics

These subroutines are part of AIX 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 function of the **valloc** subroutine is superseded by the **malloc** subroutine, which automatically page aligns large (greater than 1 page) requests. The **valloc** syntax follows:

char *valloc (*Size***)**
unsigned int *Size*;

The **alloca** subroutine obtains storage by increasing the size of the current stack frame. The speed of allocating storage this way and the automatic release of the storage on return of the function, makes the **alloca** subroutine preferable to the **malloc** subroutine in many applications.

Some assistance is typically required from compilers to remove dependence on a fixed-size stack frame and to pass extra information to the **alloca** subroutine. The details vary depending on hardware architecture, stack format, and linkage conventions, but the AIX System/370 **alloca** subroutine support described in the following text is representative.

Space allocated by the **alloca** subroutine resides in its caller's stack frame on a double-word boundary following the outgoing argument list.

The C compiler, through a switchable option, recognizes use of the function name **alloca.** Unlike special-casing of other **libc** functions like **strlen** and **memcpy**, which may be on by default, **alloca** recognition is off by default because support can affect code quality in the function using **alloca.**

When it is recognized that a function contains a call to the **alloca** subroutine:

- Code generated for the function addresses auto-variables, the incoming argument list, and the incoming register save area by using a base register that is relative to the end of the stack frame. The stack pointer, r13, is relative to the start of the stack frame and must be used only to address the outgoing argument list and other values located below any storage allocated by the **alloca** subroutine.
- The external name of **alloca** is left as "alloca" instead of being changed to "_alloca." This ensures that only functions compiled with **alloca** support can call it. The end-of-argument-list offset (rounded up) is passed as a hidden argument to **alloca** in the four bytes following the BALR instruction. (This nonstandard call format is also used by the stack-overflow checker and by the profiling mechanism.)

The **alloca** subroutine itself is written in assembler and does the following:

- Rounds space request up to a multiple of 8 bytes.
- If the request size exceeds red-zone capability, the **alloca** subroutine checks explicitly for stack overflow and returns **NIL** if there is insufficient space.
- Decreases the r13 stack pointer by the request size. Copies the stack back-pointer table into 4(r13).
- Determines the result value: r13 plus the hidden argument.
- Returns.

Related Information

The **_end, _etext, _edata** identifiers.

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** header file, includes:

```
int Type;
char *Name;
double Argument1, Argument2, ReturnValue;
```

Parameters

Type Specifies an integer describing the type of error that has occurred from the flollowing list defined by the **math.h** header 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.

Argument1 Points to the first argument with which the routine was invoked.

Argument2 Points to the second argument with which the routine was invoked.

ReturnValue 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 **errno** 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, **errno** is set to EDOM or ERANGE and the program continues.

Implementation Specifics

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

Related Information

The **bessel: j0, j1, jn, y0, y1, yn** subroutines, **exp, expm1, log, log10, log1p, pow** subroutines, **lgamma, gamma** subroutines, **hypot, cabs** subroutines, **sinh, cosh, tanh** subroutines, **sin, cos, tan, asin, acos, atan, atan2** subroutines.

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***)**
**char ***Mbstring*;
size_t *Number*;

Description

The **mblen** subroutine determines the length in bytes of a multibyte character, similar to the **NLchrlen** subroutine.

Parameters

Mbstring	Pointer to a multibyte character string.
Number	Maximum number of bytes to cosider.

Return Values

The **mblen** subroutine returns 0 if the *Mbstring* parameter points to a null. It returns –1 if a character cannot be formed from the *Number* parameter (or less than *Number*) bytes pointed to by the *Mbstring* parameter. If *Mbstring* is a null pointer, a 0 is returned.

Implementation Specifics

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

Related Information

The **NLchar** subroutines

National Language Support Overview in *General Programming Concepts*.

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 characters (code points) from the *MbString2* parameter to the end of the *MbString1* parameter, appends null to the result, and returns *MbString1*.

The **mbscmp** subroutine compares multibyte characters in the *MbString1* parameter to the *MbString2* parameter and returns an integer greater than zero if *MbString1* is greater than *MbString2*; zero if the strings are equivalent; and an integer less than zero 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 AIX Base Operating System (BOS) Runtime.

Related Information

The **mbsncat**, **mbsncmp**, **mbsncpy** subroutines, **wcscat**, **wcschr**, **wcscmp**, **wcscpy**, **wcscspn** subroutines.

National Language Support Overview in *General Programming Concepts*.

mbschr Subroutine

Purpose

Locates a character (code point) in a multibyte character string.

Library

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

Syntax

#include<mbstr.h>

char *mbschr(*MbString*, *MbCharacter***)**
**char ****MbString*;
int *MbCharacter*;

Description

The **mbschr** subroutine locates the first occurrence of *MbCharacter* in the string pointed to by the *MbString* parameter. The *MbCharacter* parameter is the code point of a multibyte character represented as an integer. The terminating null character is considered to be part of the string.

Parameters

MbString	Pointer to a multibyte character string.
MbCharacter	A code point of a multibyte character represented as an integer.

Return Values

The **mbschr** subroutine returns a pointer to *MbCharacter* within the multibyte character string or a **NULL** pointer if *MbCharacter* does not occur in the string.

Implementation Specifics

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

Related Information

The **wcsrchr** subroutine, **mbsrchr** subroutine.

National Language Support Overview in *General Programming Concepts*.

mbslen Subroutine

Purpose

Determines the number of characters (code points) in a multibyte character string.

Library

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

Syntax

#include <stdlib.h>

size_t mbslen(*MultibyteString***)**
char *mbs;

Description

The **mbslen** subroutine determines the number of characters (code points) in a multibyte character string.

Parameter

MultibyteString	Pointer 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 *MultibyteString* parameter points to a **null** or a character cannot be formed from the string pointed to by the *MultibyteString* parameter.

Implementation Specifics

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

Related Information

The **mblen** subroutine.

National Language Support Overview in *General Programming Concepts.*

mbsncat, mbsncmp, or mbsncpy Subroutine

Purpose

Performs operations on a specified number of null–terminated multibyte characters.

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 value of the *Number* parameter of characters (code points) from the *MbString2* parameter to the end of the *MbString1* parameter, appends null to the result, and returns the *MbString1* parameter.

The **mbsncmp** subroutine compares up to the value of the *Number* parameter of multibyte characters in the *MbString1* parameter to the *MbString2* parameter and returns an integer greater than zero if *MbString1* is greater than *MbString2*; zero if the strings are equivalent; and an integer less than zero if *MbString1* is less than *MbString2*.

The **mbsncpy** subroutine copies up to the value of the *N* parameter of multibyte characters from the *MbString2* parameter to the *MbString1* parameter and returns *MbString1*. If *MbString2* is shorter than *Number* characters (code points), *MbString1* is padded out to *Number* characters with null characters.

Implementation Specifics

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

Related Information

The **mbscat**, **mbscmp**, **mbscpy** subroutines, **wcsncat**, **wcsncmp**, **wcsncpy** subroutines.

National Language Support Overview in *General Programming Concepts*.

mbspbrk Subroutine

Purpose

Locates the first occurrence of multibyte characters (code points) in a string.

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 from the string pointed to by the *MbString2* parameter.

Parameters

MbString1	Pointer to a string being searched.
MbString2	Pointer to a set of characters string.

Return Values

The **mbspbrk** subroutine returns a pointer to the character, or **NULL** 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 AIX Base Operating System (BOS) Runtime.

Related Information

The **wcspbrk** subroutine, **wcswcs** subroutine.

National Language Support Overview in *General Programming Concepts.*

mbsrchr Subroutine

Purpose

Locates a character (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 the code point of a multibyte character represented as an integer. The terminating null character is considered to be part of the string.

Parameters

MbString	Pointer to a multibyte character string.
MbCharacter	A code point of a multibyte character represented as an integer.

Return Values

The **mbsrchr** subroutine returns a pointer to the *MbCharacter* parameter within the multibyte character string or a **NULL** pointer if *MbCharacter* does not occur in the string.

Implementation Specifics

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

Related Information

The **mbschr** subroutine, **wcsrchr** subroutine.

National Language Support Overview in *General Programming Concepts.*

mbstoint Subroutine

Purpose

Extracts a multibyte (single–byte or double–byte) character from a multibyte character string.

Library

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

Syntax

#include <mbstr.h>

int mbstoint(*MultibyteString***)**
**char ***MultibyteString*;

Description

The **mbstoint** subroutine extracts the multibyte character pointed to by the *MultibyteString* parameter from the multibyte character string.

Parameter

MultibyteString	Pointer to a multibyte character string.

Return Values

The **mbstoint** subroutine returns the code point of the multibyte character pointed to by the *MultibyteString* parameter. If an invalid multibyte character is encountered a 0 is returned.

Implementation Specifics

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

Related Information

The **mbsrchr** subroutine, **mbtowc** subroutine, **mbstowcs** subroutine.

National Language Support Overview in *General Programming Concepts.*

mbstowcs Subroutine

Purpose

Converts a multibyte (single–byte or double–byte) 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*;
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 result in the buffer pointed to by the *WcString* parameter. The multibyte characters are converted up to the null character or until the value of the *Number* parameter or (*Number*–1) in wide–characters have been processed.

Parameters

WcString	Pointer to the area where result of the conversion is stored.
String	Pointer to a multibyte character string.
Number	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 –1 is returned.

Implementation Specifics

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

Related Information

The **mbtowc** subroutine, **wcstombs** subroutine, **wctomb** subroutine.

National Language Support Overview in *General Programming Concepts.*

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*;
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, then converts that character to the corresponding wide–character and places it in the location pointed to by the *WideCharacter* paramter. If *WideCharacter* is **NULL**, the multibyte character is not converted. The number of bytes comprising the multibyte character is returned.

The **mbtowc** subroutine is similar to the **NCdecode** subroutine except **NCdecode** does not accept a length argument.

Parameters

WideCharacter	Pointer to location where wide–character is to be placed.
String	Pointer to multibyte character.
Number	Number of bytes of the multibyte character.

Return Values

The **mbtowc** subroutine returns a 0 if the *String* parameter is a **NULL** pointer or if *String* points to a null character (the null is converted to a wide–character null). It returns a –1 if the bytes pointed to by *String* do not form a valid multibyte character within the value of the *Number* parameter or fewer bytes. Otherwise, the number of bytes comprising the multibyte character is returned.

Implementation Specifics

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

Related Information

The **NLchar** subroutines, **mbstowcs** subroutine, **wctomb** subroutine, **wcstombs** subroutine.

National Language Support Overview in *General Programming Concepts*.

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*, ***Source*;
int *C*;
size_t *N*;

void *memchr (*S*, *C*, *N***)**
**void ***S*;
int *C*;
size_t *N*;

int memcmp (*Target*, *Source*, *N***)**
**void ***Target*, ***Source*;
size_t *N*;

void *memcpy (*Target*, *Source*, *N***)**
**void ***Target*, ***Source*;
size_t *N*;

void *memset (*S*, *C*, *N***)**
**void ***S*;
int *C*;
size_t *N*;

void *memmove (*Target*, *Source, N***)**
**void ***Source*, ***Target*;
size_t *N*;

The **memory** subroutines operate on memory areas. A memory area is an array of characters bounded by a count, and not ended by a null character. 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** header 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 is copied, or after *N* characters are been copied, whichever comes first.

The **memcmp** subroutine lexicographically compares the first *N* characters in memory area *Target* to the first *N* characters in memory area *Source*. The **memcmp** subroutine uses native character comparison, which may be signed on some machines.

The **memcpy** subroutine copies *N* characters from memory area *Source* to area *Target* and returns *Target*.

The **memset** subroutine sets the first *N* characters in memory area *S* to the value of character *C* and returns *S*.

Like the **memcpy** subroutine, the **memmove** subroutine copies *N* characters from memory area *Source* to area *Target*. However, if the *Source* and *Target* areas overlap, the move is perfomed non-destructively, 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 for which 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 the character after *C* is copied into *Target*, or a NULL pointer if *C* is not found in the first *N* characters of *Source*.

The **memchr** subroutine returns a pointer to the first occurrence of character *C* in the first *N* characters of memory area *S*, or a NULL pointer if *C* does not occur.

The **memcmp** subroutine returns the following values:

Less than 0	If the *Target* parameter is less than the *Source* parameter
Equal to 0	If *Target* is equal to *Source*
Greater than 0	If *Target* is greater than *Source*.

Implementation Specifics

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

The **memccpy** subroutine is not in the ANSI C library.

Related Information

The **swab** subroutine, **string** subroutine.

mkdir Subroutine

Purpose

Creates a directory.

Library

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

Syntax

#include <sys/mode.h>

int mkdir (*Path*, *Mode*)
char **Path*;
int *Mode*;

Description

The **mkdir** subroutine creates a new directory.

The new directory has the following:

- Owner ID set to the process effective user ID.
- Group ID set to the group ID of its parent directory.
- Permission and attribute bits 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*, *SetFileGroupID*, and *Sticky* (S_ISVTX) attributes are cleared.

Parameters

Path The name of the new directory. If Network File System 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 and write permission in the parent directory of the *Path* parameter.

Mode The mask for the read, write, and execute (RWX) flags for owner, group, and others. The *Mode* parameter specifies the directory permissions and attributes. This parameter is constructed by logically ORing values described in the **sys/mode.h** header file.

Return Values

Upon successful completion, the **mkdir** 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 **mkdir** subroutine fails 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.
EDQUOT	The directory in which the entry for the new link is being placed cannot be extended because the user's quota of disk blocks or i-nodes on the file system containing the directory is exhausted.

The **mkdir** subroutine can also fail if additional errors on page A–1 occur.

If Network File System is installed on the system, the **mkdir** subroutine can also fail if the following is true:

ETIMEDOUT	The connection timed out.

Implementation Specifics

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

Related Information

The **chmod** subroutine, **mknod** subroutine, **rmdir** subroutine, **umask** subroutine.

The **chmod** command, **mkdir** command, **mknod** command.

mknod or mkfifo Subroutine

Purpose

Creates an ordinary file, directory, FIFO, or special file.

Library

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

Syntax

#include <sys/mode.h>

int mknod (*Path*, *Mode*, *Device*)
char **Path*;
int *Mode*;
dev_t *Device*;

int mkfifo(*Path*, *Mode*)
char **Path;*
int *Mode;*

Description

The **mknod** subroutine creates a new regular file, special file or FIFO. Using the **mknod** subroutine to create file types other than FIFO special requires root user authority.

For the **mknod** subroutine to complete successfully, a process must have search permission 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 special file. No special system privileges are required.

The new file has the following characteristics:

- File type as specified by the *Mode* parameter
- Owner ID set to the process effective user ID
- Group ID set to the group ID of the parent directory
- Permission and attribute bits set according to the value of the *Mode* parameter. All bits set in the process file mode creation mask are cleared.

If the new file is a character special file with 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 provides a multiplexed interface to the device driver. The **hft** device driver uses this feature.

Parameters

Path — Names the new file. If Network File System 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** header file.

Device	The *Device* parameter is configuration–dependent and is used only if the *Mode* parameter specifies a block or character special file. The ID of the device is *Device*, and it corresponds to the st_rdev member of the structure returned by the **statx** subroutine. 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 global variable **errno** 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:

EPERM	The *Mode* parameter specifies a file type other than **S_IFIFO** and the calling process does not have root user authority.
EEXIST	The named file exists.
EROFS	The directory in which the file is to be created is located on a read–only file system.
ENOSPC	The directory that would contain the new file cannot be extended or the file system is out of file allocation resources.
EDQUOT	The directory in which the entry for the new link is being placed cannot be extended because the user's quota of disk blocks or inodes on the file system is exhausted.

The **mknod** subroutine can also fail if additional errors on page A–1 occur.

If Network File System 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 AIX Base Operating System (BOS) Runtime.

Related Information

The **chmod** subroutine, **mkdir** subroutine, **open** subroutine, **umask** subroutine, **statx** subroutine.

The **chmod** command, **mkdir** command, **mknod** command.

The **mode.h** header file, **types.h** header file.

mktemp or mkstemp Subroutine

Purpose

Constructs a unique file name.

Library

Standard C Library (**libc.a**), Berkeley Compatibility Library (**libbsd.a**)

Syntax

char *mktemp (*Template*)
char **Template*;

char *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.

Parameter

Template	Points to a string to be replaced with a unique file name. The string in the *Template* parameter must be a file name with six trailing Xs. The **mktemp** subroutine replaces the Xs with a randomly generated character sequence.

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 Xs, or if the **mktemp** subroutine is unable to construct a unique file name, the first character of the *Template* parameter string is replaced with a null character, and a **NULL** pointer 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 AIX 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 AIX Version 3 operating system is more random, this race condition is less likely.

The behavior in the case of a failure on different systems includes:

AIX 2.2 Replaces the first character of the template with a null character.

AT&T System V Returns a **NULL** pointer.

BSD Returns a file name of /.

Former implementations created a unique name by replacing Xs with the process ID and a unique letter.

AIX Version 3 operating system is compatible with the AIX 2.2 and AT&T System V operating systems.

Related Information

The **tmpfile** subroutine, **tmpnam**, **tempnam** subroutine.

The **getpid** subroutine.

mntctl Subroutine

Purpose

Returns information about the mount status of the system.

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 is described by a **vmount** structure; this structure is supplied when the virtual file system is created by the **vmount** subroutine. The **vmount** structure is defined in the **sys/vmount.h** header file.

Parameters

Command Specifies the operation to be performed. Valid commands are defined in the **sys/vmount.h** header 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 will hold 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 virtual file systems, 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** system call otherwise fails, a value of –1 is returned, and the global variable **errno** 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

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

Related Information

The **vmount**, **mount** subroutines,**uvmount**, **umount** subroutines.

moncontrol Subroutine

Purpose

Starts and stops execution profiling, after **monitor** initialization.

Library

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

Syntax

#include <mon.h>

int moncontrol(*Mode***)**

int *Mode*;

Description

The **moncontrol** routine starts and stops profiling, after it 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** function, then when **exit** is called, or when **monitor** is called with a first parameter of 0 then profiling is stopped and an output file is produced regardless of the state of profiling as set by **moncontrol**.

The **moncontrol** subroutine examines global and parameter data in the following order:

1. When the global variable **_mondata.prof_type** is not –1 (**–p** profiling defined), and is not +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 **mcrt0.o** and to +1 in **gcrt0.o** and defaults to 0 when **crt0.o** is used.

2. When *Mode* is 0:

   profiling is stopped, otherwise profiling is started.

   The following global variables are used in a call to the **profil** subroutine:

   _mondata.*ProfBuf* /*buffer address*/

   _mondata.*ProfBufSiz* /*buffer size/multi range flag*/

   _mondata.*ProfLoPC* /*pc offset for hist buffer - lo limit*/

   _mondata.*ProfScale* /*pc scale/compute scale flag*/

   These variables are initialized by the **monitor** subroutine each time it is called to start profiling.

Parameter

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 it returns 0. When the previous state was STARTED it returns 1.

Error Code

When the **moncontrol** subroutine detects an error from the call to **profil**, a –1 is returned.

Implementation Specifics

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

Related Information

The **monstartup** subroutine, **monitor** subroutine, **profil** subroutine.

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;*
caddr_t *HighProgramCounter;*
HISTCOUNTER **Buffer;*
int *BufferSize;*
int *NFunction*;

caddr_t *NotZeroA;*
caddr_t *DoNotCareA;*

Description

The **monitor** subroutine initializes the buffer area and starts profiling, or 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 range(s) specified, and accumulation of function call count data for functions that have been compiled with the **–p** or **–pg** option.

Executable programs created with **cc –p** or **cc –pg** automatically include calls to the **monitor** subroutine (via **monstartup** and **exit**) 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 whether **–p** or **–pg** profiling is to be in effect. **monitor** initializes four global variables that are used as parameters to **profil** by **moncontrol**. The **monitor** subroutine calls the **moncontrol** subroutine to start the profiling data gathering. **moncontrol** calls **profil** to start the system timer driven program address sampling.

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 **monitor** subroutine examines the global data and parameter data in this order:

1. When the global variable **_mondata.prof_type** is not equal to –1 (**–p** profiling defined) and not equal to +1 (**–pg** profiling defined), an error return is made, and the function is considered complete.

   The global variable is set to –1 in **mcrt0.o** and to +1 in **gcrt0.o** and defaults to 0 when **crt0.o** is used.
2. When the first parameter to **monitor** is 0:

   profiling is stopped and the data file is written out.

   If **–p** profiling has been in effect then the file is named **mon.out**. Otherwise **–pg** profiling has been effect and the file is named **gmon.out**. The function is complete.
3. When the first parameter to **monitor** is not 0:

   the **monitor** parameters and the profiling global variable **_mondata.prof_type** are examined to determine how to start profiling.
4. When *BufferSize* 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 *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 *ProfileBuffer* 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 one programming address range to be profiled. Programming ranges must be ordered 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** header file. It contains the fields shown below:

```
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 de-reference the function pointer to get the address of the first instruction in the function. This is required because the function reference in this context produces the address of the function descriptor. The first field of the descriptor is the address of the function code. See the examples for typical expressions to use.

Buffer (**prof** name: **p_buff**)

Defines the beginning address of an array of *BufferSize* HISTCOUNTERs to be used for data collection. This buffer includes the space for the program address sampling counters and the function count data areas. In the case of a multiple range specification, the space for the function count data area is included at the beginning of the first range *BufferSize* specification.

BufferSize (**prof** name: **p_bufsize**)

Defines the size of buffer in number of HISTCOUNTERs. Each counter is of type HISTCOUNTER (defined as short in **mon.h**). 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 the *HighProgramCounter* parameters. See **mon.h**.

NFunction

Defines the size of the space to 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)
```

poutcnt is defined in the **mon.h** header 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. The size required is defined by the following:

```
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);
```

computed and summed for all of the defined ranges. The functions and variables in this expression that are in capital letters, and the structures are defined in **mon.h**.

NotZeroA

Specifies a value of parameter 1, which is any value except zero. 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 outputs 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 shared library libc.a with **–p** profiling. Assume that the range of addresses for the shared libc.a has been determined 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 the second range values*/

pb[1].p_low = liblo;               /*libc.a low address*/

pb[1].p_high = libhi;              /*libc.a high address*/

pb[1].p_buff = pb[0].p_buff + BufferSize1;
                                    /*buffer point for second range*/

pb[1].p_BufferSize = BufferSize2;


/*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 <sys/limits.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;                       /*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    funcspace;                /*bytes needed for function call data*/

int    BufferSize;               /*total buffer size in number of HISTCOUNTERs*/

int    tonum;                    /*num to elements – tmp storage calc*/

int    tosize;                   /*to num bytes  –  tmp storage calc*/

int    fromsize;                 /*from num bytes – tmp storage calc*/


numfunc = 300;                   /*arbitrary number for example*/

range = *(uint *)zot – *(uint *)zit;
                                 /*compute program address range*/

numtics = NUM_HIST_COUNTERS( range );
                                 /*one counter for each 4 byte inst*/


/*compute function space required*/

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 );

funcspace = tosize + fromsize + sizeof(struct gfctl);
```

```
BufferSize = numtics + ( funcspace / 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_PG;
                                   /*define –pg profiling*/

rc = monitor(*(uint *)zit,*(uint *)zot, 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);

}
```

Files

mon.out

gmon.out

mon.h Defines **_mondata.prof_type** in the monglobal data structure, the **prof** structure, and the functions referred to in the examples.

Implementation Specifics

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

Related Information

The **monstartup** subroutine, **moncontrol** subroutine, **profil** subroutine.

The **end, etext** identifiers.

The **prof** command, **gprof** command.

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 **cc –p** or **cc –pg** automatically include a call to **monstartup** to profile the complete user program including system libraries. In this case, you do not need to call **monstartup**.

The **monstartup** subroutine is called by **mcrt0.o** (**–p**) or **gcrt0.o** (**–pg**) to begin profiling. **monstartup** requires a global data variable to define whether **–p** or **–pg** profiling is to be in effect. **monstartup** calls **monitor** to initialize the data areas and to 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 global variable **_mondata.prof_type** is not equal to –1 (**–p** profiling defined) and not equal to +1 (**–pg** profiling defined), an error return is made, and the function is considered complete.

   The global variable is set to –1 in **mcrt0.o** and to +1 in **gcrt0.o** and defaults to 0 when **crt0.o** is used.

2. When *LowProgramCounter* 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 *LowProgramCounter* is –1 and *HighProgramCounter* 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. *HighProgramCounter,* 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 *LowProgramCounter* is –1 and *HighProgramCounter* 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 multi-range.

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 de-reference 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 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 – 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 Value

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 AIX Base Operating System (BOS) Runtime.

Related Information

The **monitor** subroutine, **moncontrol** subroutine, **profil** subroutine.

The **end, etext** identifiers.

The **prof** command, **gprof** command.

msgctl Subroutine

Purpose

Provides message control operations.

Library

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

Syntax

#include <sys/types.h>
#include <sys/ipc.h>
#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.

Parameters

MessageQueueID Specifies the message queue identifier.

Buffer Points to a structure of type **msquid_ds**. The **msqid_ds** structure is defined in the **sys/msg.h** header file, and it contains the following members:

```
struct ipc_perm  *msg_perm;/*Operation permission structure*/
unsigned short   msg_cbytes;/*Current number of bytes on queue*/
unsigned short   msg_qnum;/*Number of messages on queue*/
unsigned short   msg_qbytes;/*Maximum number of bytes on queue*/
pid_t   msg_lspid;/*ID of last process to call msgsnd*/
pid_t   msg_lrpid;/*ID of last process to call msgrcv*/
time_t  msg_stime;/*Time of last msgsnd call*/
time_t  msg_rtime;/*Time of last msgrcv call*/
time_t  msg_ctime;/*Time of the last change to this structure
                 with a msgctl call*/
```

Command The following values for the *Command* parameter are available:

IPC_STAT Stores the current value of the above members 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 members 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 be equal to the value of msg_perm.uid or msg_perm.cuid in the data structure associated with *MessageQueueID* in order to perform this operation. To raise the value of msg_qbytes, 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 msg_perm.uid or msg_perm.cuid in the data structure associated with the *MessageQueueID* parameter in order to perform this operation.

Return Values

Upon successful completion, the **msgctl** 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 **msgctl** subroutine fails if one or more of the following are true:

EINVAL The *Command* or *MessageQueueID* parameter is not valid.

EACCES The *Command* parameter is equal to **IPC_STAT** and read permission is denied to the calling process.

EPERM Either the *Command* parameter is equal to **IPC_RMID** and the effective user ID of the calling process does not have root user authority, or *Command* is equal to **IPC_SET** and the effective user ID of the calling process is not equal to the value of msg_perm.uid or msg_perm.cuid in the data structure associated with the *MessageQueueID* parameter.

EPERM The *Command* parameter is equal to **IPC_SET**, an attempt is being made to increase the value of msg_qbytes, 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 AIX 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/types.h>
#include <sys/ipc.h>
#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 and associated message queue and data structure are created for the value of the *Key* parameter if one of the following is true:

- The *Key* parameter is equal to **IPC_PRIVATE**.
- The *Key* parameter does not already have a message queue identifier associated with it, and **IPC_CREAT** is set.

Upon creation, the data structure associated with the new message queue identifier is initialized as follows:

- msg_perm.cuid, msg_perm.uid, msg_perm.cgid, and msg_perm.gid are set equal to the effective user ID and effective group ID, respectively, of the calling process.
- The low–order 9 bits of msg_perm.mode are set equal to the low–order 9 bits of the *MessageFlag* parameter.
- msg_qnum, msg_lspid, msg_lrpid, msg_stime, and msg_rtime are set equal to 0.
- msg_ctime is set equal to the current time.
- msg_qbytes is set equal to the system limit.

The **msgget** subroutine performs the following actions:

1. The **msgget** subroutine either finds or creates (depending on the value of *MessageFlag*) a queue with the *Key* parameter.
2. The **msgget** subroutine returns the ID of the queue header to its caller.

Parameters

Key	Specifies either the value **IPC_PRIVATE** or an 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 **IPC_CREAT** 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.
	The values that begin with **S_I** are defined in the **sys/mode.h** header 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 global variable **errno** is set to indicate the error.

Error Codes

The **msgget** subroutine fails if one or more of the following are 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 would not be 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 systemwide would be exceeded.
EEXIST	A message queue identifier exists for the *Key* parameter, and both **IPC_CREAT** and **IPC_EXCL** are set.

Implementation Specifics

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

Related Information

The **msgctl** subroutine, **msgrcv** subroutine, **msgsnd** subroutine, **msgxrcv** subroutine, **ftok** subroutine.

msgrcv Subroutine

Purpose

Reads a message from a queue.

Library

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

Syntax

#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/msg.h>

int msgrcv (*MessageQueueID*,*MessagePointer*,
MessageSize,*MessageType*, *MessageFlag*)
int *MessageQueueID*;
void **MessagePointer*;
int *MessageSize*;
long *MessageType*;
int *MessageFlag*;

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.

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** header file, and it contains the following members:

```
long    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 mtext 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 **MSG_NOERROR** 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 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 Is either 0, or is constructed by logically ORing one or more of the following values:

MSG_NOERROR Truncates the message if it is longer than *MessageSize* bytes.

IPC_NOWAIT Specifies the action to take if a message of the desired type is not on the queue:

- If **IPC_NOWAIT** is set, the calling process returns a value of –1 and sets the global variable **errno** to **ENOMSG**.
- If **IPC_NOWAIT** 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, **errno** is set to **EIDRM**, 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, **msgrcv** returns a value equal to the number of bytes actually stored into mtext and the following actions are taken with respect to the data structure associated with the *MessageQueueID* parameter:

- **msg_qnum** is decremented by 1.
- **msg_lrpid** is set equal to the process ID of the calling process.
- **msg_rtime** is set equal to the current time.

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

Error Codes

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

EINVAL	The *MessageQueueID* parameter is not a valid message queue identifier.
EACCES	Operation permission is denied to the calling process.
EINVAL	The *MessageSize* parameter is less than 0.
E2BIG	mtext is greater than *MessageSize* and **MSG_NOERROR** is not set.
ENOMSG	The queue does not contain a message of the desired type and **IPC_NOWAIT** is set.
EFAULT	The *MessagePointer* parameter points outside of the allocated address space of the process.
EINTR	The function **msgrcv** was interrupted by a signal.
EIDRM	The message queue identifier specified by *MessageQueueID* has been removed from the system.

Implementation Specifics

This subroutine is part of AIX 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/types.h>
#include <sys/ipc.h>
#include <sys/msg.h>

int msgsnd (*MessageQueueID,MessagePointer,MessageSize,MessageFlag***)**
int *MessageQueueID*;
void * *MessagePointer*;
size_t *MessageSize*;
int *MessageFlag*;

Description

The **msgsnd** subroutine sends a message to the queue specified by the *MessageQueueID* parameter. The current process must have write permission in order to perform this operation. The *MessagePointer* parameter points to a **msgbuf** structure containing the message. The **msgbuf** structure is defined in the **sys/msg.h** header file, and it contains the following members:

```
long  mtype;          /* Message type */
char  mtext[1];       /* Beginning of message text */
```

The mtype field is a positive integer that is used by the receiving process for message selection. The mtext field is any text of the length in bytes specified by the *MessageSize* parameter. The *MessageSize* parameter can range from 0 to a system–imposed maximum.

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 **msg_qbytes**.
- The total number of messages on the queue is equal to a system–imposed limit.

These actions are as follows:

- If *MessageFlag* is set to **IPC_NOWAIT**, the message is not sent, and **msgsnd** returns a value of –1 and sets the global variable **errno** to **EAGAIN**.
- If *MessageFlag* is 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.
- *MessageQueueID* is removed from the system. (For information on how to remove *MessageQueueID*, see the **msgctl** system call.) When this occurs, **errno** is set equal to **EIDRM**, 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 **sigaction**.

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:

- msg_qnum is incremented by 1.
- msg_lspid is set equal to the process ID of the calling process.
- msg_stime is set equal to the current time.

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

Error Codes

The **msgsnd** subroutine fails and no message is sent if one or more of the following are true:

EINVAL	The *MessageQueueID* parameter is not a valid message queue identifier.
EACCES	Operation permission is denied to the calling process.
EINVAL	**mtype** is less than 1.
EAGAIN	The message cannot be sent for one of the reasons stated previously, and *MessageFlag* is set to **IPC_NOWAIT**.
EINVAL	The *MessageSize* parameter is less than 0 or greater than the system–imposed limit.
EFAULT	The *MessagePointer* parameter points outside of the process' address space.
EINTR	**msgsnd** received a signal.
EIDRM	The message queue identifier specified by *MessageQueueID* has been removed from the system.

Implementation Specifics

This subroutine is part of AIX 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

#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/msg.h>
int msgxrcv(*MessageQueueID,MessagePointer, MessageSize, MessageType,MessageFlag*)
int *MessageQueueID*;
struct msgxbuf * *MessagePointer*;
int *MessageSize*;
long *MessageType*;
int *MessageFlag*;

Description

The **msgxrcv** subroutine reads a message from the queue specified by the *MessageQueueID* parameter and stores it into the extended message receive buffer pointed to by the *MessagePointer* parameter. The current process must have read permission in order to perform this operation. The **msgxbuf** structure is defined in the **sys/msg.h** header file, and it contains the following members:

```
time_t  mtime;   /* Time and date message was sent */
uid_t   muid;    /* Sender's effective user ID */
gid_t   mgid;    /* Sender's effective group ID */
pid_t   mpid;    /* Sender's process ID */
mtyp_t  mtype;   /* Message type */
char    mtext[1]/* Beginning of message text */
```

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 mtext in bytes. The receive message is truncated to the size specified by the *MessageSize* parameter if it is larger than the *MessageSize* parameter and **MSG_NOERROR** 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 **MSG_NOERROR** is not set, the **msgrcv** subroutine fails and sets the global variable **errno** to **E2BIG**.

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 Either 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 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 **IPC_NOWAIT** is set, the calling process returns a value of –1 and sets **errno** to **ENOMSG**.
- If **IPC_NOWAIT** 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, **errno** is set to **EIDRM**, 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 mtext, and the following actions are taken with respect to the data structure associated with the *MessageQueueID* parameter:

- msg_qnum is decremented by 1.
- msg_lrpid is set equal to the process ID of the calling process.
- msg_rtime is set equal to the current time.

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

Error Codes

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

EINVAL The *MessageQueueID* parameter is not a valid message queue identifier.

EACCES Operation permission is denied to the calling process.

EINVAL	*MessageSize* is less than 0.
E2BIG	mtext is greater than the *MessageSize* parameter and **MSG_NOERROR** is not set.
ENOMSG	The queue does not contain a message of the desired type and **IPC_NOWAIT** 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 *MessageQueueID* is removed from the system.

Implementation Specifics

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

Related Information

The **msgctl** subroutine, **msgget** subroutine, **msgrcv** subroutine.

NCctype Subroutines

Purpose

Classifies characters for national language support environments.

Library

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

Syntax

#include <NLctype.h>

int NCisNLchar (*Xcharacter*)
int *Xcharacter;*

int NCisalpha (*Xcharacter*)
int *Xcharacter;*

int NCisupper (*Xcharacter*)
int *Xcharacter;*

int NCislower (*Xcharacter*)
int *Xcharacter;*

int NCisdigit (*Xcharacter*)
int *Xcharacter;*

int NCisxdigit (*Xcharacter*)
int *Xcharacter;*

int NCisalnum (*Xcharacter*)
int *Xcharacter;*

int NCisspace (*Xcharacter*)
int *Xcharacter;*

int NCispunct (*Xcharacter*)
int *Xcharacter;*

int NCisprint (*Xcharacter*)
int *Xcharacter;*

int NCisgraph (*Xcharacter*)
int *Xcharacter;*

int NCiscntrl (*Xcharacter*)
int *Xcharacter;*

Description

Character classification is user–configurable per process through the locale indicated by the **LC_COLLATE** environment variable.

These subroutines classify character–coded integer values using information specified by the current **LC_COLLATE** configuration. The *Xcharacter* parameter is tested as an **NLchar** (an extended character); each subroutine is a predicate form returning 0 for **FALSE**, and a nonzero value for **TRUE**. The value of *Xcharacter* is in the domain of any legal **NLchar**, in a value range from 0 to **NLCHARMAX –1**, inclusive. It can also have a special value of –1. If the value of *Xcharacter* is not in the domain of the routine, the result is undefined.

Japanese Language Support Information

When running AIX with Japanese Language Support, the value range is 0 to **NLCOLMAX**.

Parameter

Xcharacter Character to be classified.

Return Values

The **NCisNLchar** macro is defined on all valid integer values, whereas the other macros are defined only where **NCisNLchar** is true, and on the special value of –1 (end of file).

When a nonzero value is returned for *Xcharacter*:

NCisNLchar *Xcharacter* is a valid **NLchar** with a value between 0 and **NLCHARMAX–1**, inclusive. (The **NCisNLchar** subroutine is not available when running AIX with Japanese Language Support on your system.)

NCisalpha *Xcharacter* is an alphabetic character applicable to **isalpha** and **isjalpha**.

NCisupper *Xcharacter* is an uppercase alphabetic character applicable to **isupper** and **isjupper**.

NCislower *Xcharacter* is a lowercase letter applicable to **islower** and **isjlower**.

NCisdigit *Xcharacter* is a decimal digit (0–9) applicable to **isdigit** and **isjdigit**.

NCisxdigit *Xcharacter* is a hexadecimal digit (0–9, A–F, or a–f) applicable to **isxdigit** and **isjxdigit**.

NCisalnum *Xcharacter* is an alphanumeric character or digit applicable to **isalnum**, **isjdigit**, and **isjalpha**.

NCisspace *Xcharacter* is a space, tab, carriage return, new–line, vertical tab, or form–feed character applicable to **isspace** and **isjspace**.

NCispunct *Xcharacter* is a punctuation character (neither a control character nor an alphanumeric character) applicable to **ispunct** and **isjpunct**.

NCisprint *Xcharacter* is a printing character (including the space character) applicable to **isprint** and **isjprint**.

NCisgraph *Xcharacter* is a printing character (excluding the space character) applicable to **isgraph** and **isjgraph**.

NCiscntrl *Xcharacter* is an ASCII delete character (0177) or an ordinary ASCII control character other than the 4 single–shift characters. This subroutine is applicable only to ASCII characters; it does not apply to kanji characters.

Implementation Specifics

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

Related Information

The **getc, fgetc, getchar, getw** subroutines, **ctype** subroutines **NLchar** subroutines.

National Language Support Overview in *General Programming Concepts*.

NCstring Subroutines

Purpose

Performs operations on strings of type **NLchar**.

Library

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

Syntax

#include <NLchar.h>
NLchar *NCstrcat (*Xstring1*, *Xstring2***)**
NLchar **Xstring1*, **Xstring2*;

NLchar *NCstrncat (*Xstring1*, *Xstring2*, *Number***)**
NLchar **Xstring1*, **Xstring2*;
int *Number;*

int NCstrcmp (*Xstring1*, *Xstring2***)**
NLchar **Xstring1*, **Xstring2*;

int NCstrncmp (*Xstring1*, *Xstring2*, *Number***)**
NLchar **Xstring1*, **Xstring2*;
int *Number;*

NLchar *NCstrcpy (*Xstring1*, *Xstring2***)**
NLchar **Xstring1*, **Xstring2*;

NLchar *NCstrncpy (*Xstring1*, *Xstring2*, *Number***)**
NLchar **Xstring1*, **Xstring2*;
int *Number;*

int NCstrlen (*Xstring***)**
NLchar **Xstring*;

NLchar *NCstrchr (*Xstring, Character***)**
NLchar **Xstring*, *Character*;

NLchar *NCstrrchr (*Xstring*, *Character***)**
NLchar **Xstring*, *Character*;

NLchar *NCstrpbrk (*Xstring1*, *String2***)**
NLchar **Xstring1*;
char **String2*;

int NCstrspn (*Xstring1*, *String2***)**
NLchar **Xstring1*;
char **String2*;

int NCstrcspn (*Xstring1*, *String2***)**
NLchar **Xstring1*;
char **String2*;

NLchar *NCstrtok (*Xstring1*, *String2***)**
NLchar **Xstring1*;

char **String2*;

NLchar *NCstrdup (*Xstring1*)
NLchar **Xstring1*;

Description

The **NCstring** subroutines copy, compare, and append strings in memory, and determine such things as location, size, and existence of strings in memory. For these subroutines, a string is an array of **NLchars**, terminated by a null character. The **NCstring** subroutines parallel the **string** subroutines, but operate on strings of type **NLchar** rather than on type **char**, except as specifically noted below.

These subroutines require their parameters (except the *String2* parameter) to be explicitly converted to type **NLchar**, so they should be used on input that is to be scanned many times for each time it is converted. Where this performance concern does not apply, the **NLstring** subroutines are easier to use.

The *String2* parameter is a string of type **char** containing code point representations of ASCII characters or extended characters for international character support. This supports the use of a double–quoted string for this parameter in calling programs.

The parameters *Xstring1*, *Xstring2* and *Xstring* point to strings of type **NLchar** (arrays of **NLchars** terminated by a null character). The *String2* parameter points to strings of type **char**.

The subroutines **NCstrcat**, **NCstrncat**, **NCstrcpy**, and **NCstrncpy** all alter *Xstring1*. They do not check for overflow of the array pointed to by *Xstring1*. All string movement is performed character by character and starts at the left. Overlapping moves toward the left work as expected, but overlapping moves to the right may give unexpected results. All of these subroutines are declared in the **NLchar.h** header file.

The **NCstrcat** subroutine appends a copy of the string pointed to by the *Xstring2* parameter to the end of the string pointed to by the *Xstring1* parameter. The **NCstrcat** subroutine returns a pointer to the null–terminated result.

The **NCstrncat** subroutine copies at most *Number* **NLchars** of *Xstring2* to the end of the string pointed to by the *Xstring1* parameter. Copying stops before *Number* **NLchars** if a null character is encountered in the *Xstring2* string. The **NCstrncat** subroutine returns a pointer to the null–terminated result.

The **NCstrcmp** subroutine lexicographically compares the string pointed to by the *Xstring1* parameter to the string pointed to by the *Xstring2* parameter. The **NCstrcmp** subroutine returns a value that is:

- Less than 0 if *Xstring1* is less than *Xstring2*
- Equal to 0 if *Xstring1* is equal to *Xstring2*
- Greater than 0 if *Xstring1* is greater than *Xstring2*.

The **NCstrncmp** subroutine makes the same comparison as **NCstrcmp**, but it compares at most *Number* pairs of **NLchars**. Both **NCstrcmp** and **NCstrncmp** use the environment variables **LC_COLLATE**, **LC_CTYPE**, and **LANG** to determine the collating sequence for performing comparisons. Unless a true collating relationship is to be tested, the **strcmp** and **strncmp** subroutines can instead be used for equality comparisons. The bytes will match regardless of the **NLchars** in the string.

The **NCstrcpy** subroutine copies the string pointed to by the *Xstring2* parameter to the character array pointed to by the *Xstring1* parameter. Copying stops when the null character is copied. The **NCstrcpy** subroutine returns the value of the *Xstring1* parameter.

The **NCstrncpy** subroutine copies *Number* **NLchars** from the string pointed to by the *Xstring2* parameter to the character array pointed to by the *Xstring1* parameter. If *Xstring2* is less than *Number* **NLchars** long, then **NCstrncpy** pads *Xstring1* with trailing null characters to fill *Number* **NLchars**. If *Xstring2* is *Number* or more **NLchars** long, then only the first *Number* **NLchars** are copied; the result is not terminated with a null character. The **NCstrncpy** subroutine returns the value of the *Xstring1* parameter.

The **NCstrlen** subroutine returns the number of **NLchars** in the string pointed to by the *Xstring* parameter, not including the terminating null character.

The **NCstrchr** subroutine returns a pointer to the first occurrence of the **NLchar** specified by the *Character* parameter in the string pointed to by the *Xstring* parameter. A **NULL** pointer is returned if the **NLchar** does not occur in the string. The null character that terminates a string is considered to be part of the string.

The **NCstrrchr** subroutine returns a pointer to the last occurrence of the **NLchar** specified by the *Character* parameter in the string pointed to by the *Xstring* parameter. A **NULL** pointer is returned if the **NLchar** does not occur in the string. The null character that terminates a string is considered to be part of the string.

The **NCstrpbrk** subroutine returns a pointer to the first occurrence in the string pointed to by the *Xstring1* parameter of any code point from the string pointed to by the *String2* parameter. A **NULL** pointer is returned if no character matches.

The **NCstrspn** subroutine returns the length of the initial segment of the string pointed to by the *Xstring1* parameter that consists entirely of code points from the string pointed to by the *String2* parameter.

The **NCstrcspn** subroutine returns the length of the initial segment of the string pointed to by the *Xstring1* parameter that consists entirely of code points **not** from the string pointed to by the *String2* parameter.

The **NCstrtok** subroutine returns a pointer to an occurrence of a text token in the string pointed to by the *Xstring1* parameter. The *String2* parameter specifies a set of code points as token delimiters. If the *Xstring1* parameter is anything other than **NULL**, then the **NCstrtok** subroutine reads the string pointed to by the *Xstring1* parameter until it finds one of the delimiter code points specified by the *String2* parameter. It then stores a null character into the string, replacing the delimiter code point, and returns a pointer to the first **NLchar** of the text token. The **NCstrtok** subroutine keeps track of its position in the string so that subsequent calls with a **NULL** *Xstring1* parameter step through the string. The delimiters specified by the *String2* parameter can be changed for subsequent calls to **NCstrtok**. When no tokens remain in the string pointed to by the *Xstring1* parameter, the **NCstrtok** subroutine returns a **NULL** pointer.

The **NCstrdup** subroutine returns a pointer to an **NLchar** string that is a duplicate of the **NLchar** string to which the *Xstring1* parameter points. Space for the new string is allocated using the **malloc** subroutine. When a new string cannot be created, a **NULL** pointer is returned.

Implementation Specifics

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

Related Information

The **NLchar** subroutines, **NLstring** subroutines, **NLstrtime** subroutines, **wstring** subroutines, **string** subroutines.

National Language Support Overview in *General Programming Concepts*.

newpass Subroutine

Purpose

Generates a new password for a user.

Library

Security Library (**libs.a**)

Syntax

#include <usersec.h>
char *newpass(*Password)*
struct userpw **Password;*

Description

The **newpass** subroutine will generate a new password for the user specified by the *Password* parameter. The new password will be checked to insure that it meets the password rules on the system unless the user is exempted from these restrictions, which are defined in the **pw_restrictions** stanza of the **login.cfg** configuration file and are described in the **passwd** command.

Passwords can contain almost any legal value for a character, but may not contain NLS code points. Passwords may not be longer than **MAX_PASS** value of characters.

The **newpass** subroutine will authenticate 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 for 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.

Parameter

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 bitmask containing zero 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 **login** or **su** to this account.

Security

Policy: Authentication

In order to change a password, the invoker must be properly authenticated.

Note: Programs which invoke the **newpass** subroutine should be especially conscious of 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 **errno** is set to indicate the error.

Error Codes

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

EACCES	The password or password restrictions information cannot be read.
EINVAL	The structure passed into the **newpass** subroutine is invalid.
ESAD	Security authentication is denied for the invoker.
EPERM	The user is unable to change a password of a user that has the **PW_ADMCHG** bit set and the real user id of the process is not the root user.

Implementation Specifics

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

Related Information

The **getpass** subroutine, **getuserpw** subroutine.

The **login** command, **passwd** command, **pwdadm** command.

NLcatgets Subroutine

Purpose

Allows initial access to an opened catalog.

Library

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

Syntax

#include <nl_types>

char *NLcatgets(*CatalogDescriptor, SetNumber, MessageNumber, String***)**
nl_catd *CatalogDescriptor*;
int *SetNumber*, *MessageNumber*;
char **String*;

Description

The **NLcatgets** subroutine is used to access a catalog, after first using the **NLcatopen** subroutine to prepare the message catalog for access.

If the **NLcatgets** subroutine finds the specified message, it loads the message into a character string buffer, terminates the message string with a null character, and returns a pointer to the buffer. The pointer is used to reference the buffer and display the message. The message in the buffer is overwritten by the next call to the **NLcatgets** subroutine.

Parameters

CatalogDescriptor	Specifies a catalog description that is returned from the **NLcatopen** subroutine.
SetNumber	Specifies the set ID.
MessageNumber	Specifies the message ID. *SetNumber* and *MessageNumber* specify a particular message in the catalog.
String	Specifies a message string.

Error Codes

If **NLcatgets** fails for any reason, it returns a pointer to the user-supplied default message string pointed to by the *String* parameter.

Implementation Specifics

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

Related Information

The **catgets** subroutine, **catgetmsg** subroutine,**NLgetamsg** subroutine.

NLchar Subroutines

Purpose

Handles data type **NLchar**

Library

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

Syntax

#include <NLchar.h>
typedef unsigned short NLchar;

int NCdecode (*Character*, *Xcharacter***)**
**char ***Character*;
**NLchar ***Xcharacter*;

int NCdecstr (*Character*, *Xcharacter*, *Length***)**
**char ***Character*;
**NLchar ***Xcharacter*;
int *Length*;

int NCchrlen (*Nlcharacter***)**
NLchar *Nlcharacter*;

int NCencode (*Xcharacter*, *Character***)**
**NLchar ***Xcharacter*;
**char ***Character*;

int NCencstr (*Xcharacter*, *Character*, *Length***)**
**NLchar ***Xcharacter*;
**char ***Character*;
int *Length*;

int NLisNLcp (*Character***)**
char *Character*;

int NLchrlen (*Character***)**
**char ***Character*;

char *wstrtos (*Character*, *Xcharacter***)**
**char ***Character*;
**wchar ***Xcharacter*;

wchar *strtows (*Xcharacter*, *Character***)**
**wchar ***Xcharacter*;
**char ***Character*;

Description

Characters for national language support can be either 1 or 2 bytes long, while all ASCII characters are 1 byte long. The **NLchar** data type, which is identical to the **wchar_t** data type, represents both ASCII and extended characters as single units of storage. The **NLchar** subroutines listed here convert between character types **char** and **NLchar** and provide information about a given character of either type.

The **NCdecode** subroutine converts a character starting at *Character* into an **NLchar** at *Xcharacter*, and returns the number of bytes read from *Character*. The **NCencode** subroutine makes the inverse translation from type **NLchar** to type **char** and returns the number of bytes written to *Character*.

The **NCdecstr** subroutine converts a string of characters from type **char** to type **NLchar**, and the **NCencstr** does the reverse translation. Both subroutines require the address of the source and destination strings and the total number of elements available for the destination string. The destination string terminates with a 0 element, which is *not* included in the string length. The destination length should include space for the terminator. If insufficient space is left for the destination string, a portion of it is not converted. The subroutines return the length of the string in elements, *not* including the terminating 0.

Japanese Language Support Information

The **wstrtos** and **strtows** subroutines are similar to the **NCdecstr** and **NCencstr** subroutines. The **wstrtos** subroutines converts a **wchar_t** data type string, terminated by an **wchar** nul character '**\0**' and pointed to by *Xcharacter*, to a character string pointed to by the *Character* parameter. The **strtows** subroutine converts a character string, terminated by a null character and pointed to by the *Character* parameter, to a **wchar** data type string, then stores it in a **wchar_t** string pointed to by *Xcharacter*. Neither function checks for overflow of the converted string.

The **NCdechr** subroutine is like the **NCdecode** subroutine except that **NCdechr** simply returns the value of **NLchar** rather than writing the **NLchar** into memory.

The **NLisNLcp**, **NCchrlen**, and **NLchrlen** subroutines return information about a given character. **NLisNLcp** returns a 0 if the character at the *Character* parameter is not an extended character, but returns the length of the character if it is an extended character. **NCchrlen** returns the length in bytes that an **NLchar** would have if it were converted into an extended or an ASCII character by **NCencode**. **NLchrlen** returns the length in bytes of the extended or ASCII character starting at *Character*.

Parameters

Character	A pointer to a character string.
Xcharacter	A pointer to an **NLchar** or **NLchar** string.
Length	The total number of elements available for the destination string.
Nlcharacter	An extended character.

Implementation Specifics

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

Related Information

The **conv** subroutines, **ctype** subroutines and **NCctype** subroutines.

National Language Support Overview in *General Programming Concepts*.

NLcplen Subroutine

Purpose

Returns the number of code points in a string.

Library

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

Japanese Language Support Syntax

When running AIX with Japanese Language Support on your system, the following subroutine, stored in **libc.a**, is provided:

#include <string.h>
int NLcplen (*String***)**
**char ***S*;

Description

The **NLcplen** subroutine returns the number of code points in the string pointed to by the *String* parameter, not including the terminating null character.

Parameter

String	Pointer to a string.

Implementation Specifics

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

Related Information

NLstring subroutines.

National Language Support Overview in *General Programming Concepts.*

NLescstr, NLunescstr or NLflatstr Subroutine

Purpose

Translates strings of characters.

Library

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

Syntax

#include <ctype.h>

int NLescstr (*Source*, *Destination*, *Length*)
char **Source*, **Destination*;
int *Length*;

int NLflatstr (*Source*, *Destination*, *Length*)
char **Source*, **Destination*;
int *Length*

int NLunescstr(*Source*, *Destination*, *Length*)
char **Source*, **Destination*;
int *Length*;

Description

These subroutines convert an entire string of type **char**, perhaps containing extended characters, into a string of pure ASCII bytes. Each of these subroutines require three parameters: the *Source* parameter contains the address of the source string, the *Destination* parameter contains the address of the string, and the *Length* parameter, gives the total number of bytes available in the destination string. Each writes a result string terminated by a null character and returns its length in bytes. The *Length* parameter should include space for the null character. If the *Destination* parameter is too short to contain the entire output string, not all of the *Source* parameter is translated.

The **NLescstr** subroutine translates each ASCII or extended character in *Source* to pure ASCII. Each extended character encountered is translated to a printable ASCII escape sequence that uniquely identifies the extended character. The display symbol facility contains a list of these escape sequences.

The **NLunescstr** subroutine performs the inverse translation by translating each ASCII byte of the *Source* parameter into the *Destination* parameter, and translate each ASCII escape sequence back into the extended character it represents.

The **NLflatstr** subroutine translates each character, ASCII or extended, in the *Source* parameter to a single ASCII byte in the *Destination* parameter. The *Destination* parameter can have fewer bytes than the *Source* parameter, but the number of logical characters, or the display length, is the same.

Note: The **NLflatstr** subroutine is not available when running AIX with Japanese Language Support.

Parameters

Source A pointer to the source string.

Destination A pointer to the destination string.

Length The number of bytes available in the destination string.

Implementation Specifics

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

Related Information

The **ctype** subroutines, **getc**, **fgetc**, **getchar**, **getw** subroutines, **getwc** , **fgetwc, getwchar** subroutines, **NCctype** subroutine, **NCstring** subroutines, **NLchar** subroutines and **NLstring** subroutines.

The **display symbol** file.

National Language Support Overview in *General Programming Concepts*.

NLgetamsg Subroutine

Purpose

Opens a catalog, retrieves a specified message, and closes the catalog.

Library

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

Syntax

char *NLgetamsg (*CatalogName, SetNumber, MessageNumber, String***)**
char *CatalogName*;
int *SetNumber, MessageNumber*;
char **String*;

Description

The **NLgetamsg** subroutine opens a catalog, retrieves the specified message, and closes the catalog, all in one call. This is particularly useful for infrequent message display. If **NLgetamsg** finds the specified message, it loads the message into a character string buffer, ending the message string with a null character, and returns a pointer to the buffer.

Parameters

CatalogName	Specifies the name of the message catalog file to be opened.
SetNumber	Specifies the set ID.
MessageNumber	Specifies the message ID. *SetNumber* and *MessageNumber* specify a particular message in the catalog.
String	Specifies the string character buffer.

Error Code

If **NLgetamsg** is unsuccessful, it returns a pointer to the user-supplied default message string pointed to by the *String* parameter.

Implementation Specifics

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

Related Information

The **catgets** subroutine, **catgetmsg** subroutine, **NLcatgets** subroutine.

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***)**
char **FileName*;
struct nlist **N1*;

Description

The **nlist** subroutine allows a program to examine thename list in the executable file named by the *FileName* parameter.It selectively extracts a list of values and places them in the arrayof **nlist** structures pointed to by the *N1* parameter.

The name list specified by the *N1* parameter consistsof an array of structures containing names of variables, types, andvalues. The list is terminated with an element that has a null stringin the name structure member. Each variable nameis looked up in the name list of the file. If the name is found, thetype and value or 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 andvalue entries are set to 0.

All entries are set to 0 if the specified file cannot be reador if it does not contain a valid name list.

You can use the **nlist** subroutine to examine the systemname list kept in the /**unix** file. By examining this list,you can ensure that your programs obtain current system addresses.

The **nlist.h** header 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

Parameters

FileName	Specifies the name of the file containing aname list.
N1	Points to the array of **nlist** structures.

Return Values

Upon successful completion, a 0 is returned. In BSD, the numberof unfound namelist entries is returned. If the file cannot be foundor 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, compilewith **libbsd.a**.

Implementation Specifics

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

Related Information

The **a.out** file.

nl_langinfo Subroutine

Purpose

Returns information on 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 corresponding to the *Item* parameter. The active language or cultural area is determined by the default value of the environment variables or the most recent call to the **setlocale** subroutine.

The values for the *Item* parameter are defined in the **langinfo.h** file.

Parameter

Item Information needed from locale.

Return Values

In a locale where **langinfo** data is not defined, the **nl_langinfo** subroutine returns a pointer to the corresponding string in the C locale. In all locales, **nl_langinfo** returns a pointer to an empty string if the *Item* parameter contains an invalid setting.

Implementation Specifics

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

Related Information

The **setlocale** subroutine.

National Language Support Overview in *General Programming Concepts*

NLstring Subroutines

Purpose

Perform operations on strings containing code points.

Library

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

Syntax

#include <NLchar.h>
char *NLstrcat (*String1, String2*)
char **String1*, **String2*;

char *NLstrncat (*String1, String2, Number*)
char **String1*, **String2*;
int *Number*;

int NLstrcmp (*String1, String2*)
char **String1*, **String2*;

int NLstrncmp (*String1, String2, Number*)
char **String1*, **String2*;
int *Number*;

char *NLstrcpy (*String1, String2*)
char **String1*, **String2*;

char *NLstrncpy (*String1, String2, Number*)
char **String1*, **String2*;
int *Number*;

int NLstrlen (*String*)
char **String*;

int NLstrdlen (*String*)
char **String*;

char *NLstrchr (*String, Character*)
char **String*, *Character*;

char *NLstrrchr (*String, Character*)
char **S*, *Character*;

char *NLstrpbrk (*String1, String2*)
char **String1*, **String2*;
int NLstrspn (*String1, String2*)
char **String1*, **String2*;

int NLstrcspn (*String1, String2*)
char **String1*, **String2*;

char *NLstrtok (*String1, String2*)
char **String1*, **String2*;

Description

The **NLstring** subroutines copy, compare, and append strings in memory, and determine such values as location, size, and existence of strings in memory. A string is an array of code points terminated by a null character. The **NLstring** subroutines parallel the **string** subroutines and **NLstrcat, NLstrncat, NLstrcpy, NLstrncpy**, and **NLstrlen** are identical in function to their **string** counterparts.

The subroutines **NLstrcat, NLstrncat, NLstrcpy**, and **NLstrncpy** all alter the *String1* parameter. They do not check for overflow of the array pointed to by *String1*. All string movement is performed character by character and starts at the left. Overlapping moves toward the left work as expected, but overlapping moves to the right can give unexpected results. All of these subroutines are declared in the **NLchar.h** header file.

The **NLstrcat** subroutine appends a copy of the string pointed to by the *String2* parameter to the end of the string pointed to by the *String1* parameter. The string is at most *Number* bytes; this can represent fewer than *Number* code points. The **NLstrcat** subroutine returns a pointer to the null–terminated result.

The **NLstrncat** subroutine copies at most *Number* characters of *String2* to the end of the string pointed to by the *String1* parameter. Copying stops before *Number* characters if a null character is encountered in the *String2* string. The **NLstrncat** subroutine returns a pointer to the null–terminated result.

The **NLstrcmp** subroutine lexicographically compares the string pointed to by the *String1* parameter to the string pointed to by the *String2* parameter. The **NLstrcmp** subroutine returns a value that is:

Less than 0 if *String1* is less than *String2*

Equal to 0 if *String1* is equal to *String2*

Greater than 0 if *String1* is greater than *String2*.

The **NLstrncmp** subroutine makes the same comparison as **NLstrcmp**, but it compares at most *Number* bytes. Characters that have 2–byte representations can cause **NLstrncmp** to return 0 for unequal strings. If *Number* divides a 2–byte character, then the last byte comparison is skipped. If the only difference in the two strings is in that last byte, an incorrect true is returned.

Both the **NLstrcmp** and **NLstrncmp** subroutines use the locale specific data based on the **LC_COLLATE** category. Unless a true collating relationship is to be tested, the **strcmp** and **strncmp** subroutines should be used

The **NLstrcpy** subroutine the string pointed to by the *String2* parameter to the character array pointed to by the *String1* parameter. Copying stops when the null character is copied. The **NLstrcpy** subroutine returns the pointer to the beginning of the *String1* parameter.

The **NLstrncpy** subroutine copies the string pointed to by the *String2* parameter to the character array pointed to by the *String1* parameter, copying at most *Number* bytes. If *String2* is shorter than *Number*, a null character is added to *String1*. If the length in bytes of *String2* is greater than *Number*, the result is not null–terminated. If byte *Number* is the first byte of an extended code, byte *Number* is not copied; *String1* is *Number*–1 in length. The **NLstrncpy** subroutine returns the pointer to the beginning of the *String1* parameter.

The **NLstrlen** subroutine returns the number of bytes in the string pointed to by the *String* parameter, not including the terminating null character.

The **NLstrdlen** subroutine returns the number of **code points** in the string pointed to by *String*, not including the terminating null character.

Japanese Language Support Information: When running AIX with Japanese Language Support the **NLstrdlen** subroutine returns the number of bytes (equal to the number of display columns).

The **NLstrchr** subroutine returns a pointer to the first occurrence of the code point corresponding to the **NLchar** specified by the *Character* parameter in the string pointed to by the *String* parameter. A **NULL** pointer is returned if the code point does not occur in the string. The null character that terminates a string is considered to be part of the string.

The **NLstrrchr** subroutine returns a pointer to the last occurrence of the code point corresponding to the **NLchar** specified by the *Character* parameter in the string pointed to by the *String* parameter. A **NULL** pointer is returned if the code point does not occur in the string. The null character that terminates a string is considered to be part of the string.

The **NLstrpbrk** subroutine returns a pointer to the first occurrence in the string pointed to by the *String1* parameter of any code point from the string pointed to by the *String2* parameter. A **NULL** pointer is returned if no character matches.

The **NLstrspn** subroutine returns the length of the initial segment of the string pointed to by the *String1* parameter that consists entirely of code points from the string pointed to by the *String2* parameter.

The **NLstrcspn** subroutine returns the length of the initial segment of the string pointed to by the *String1* parameter that consists entirely of code points *not* from the string pointed to by the *String2* parameter.

The **NLstrtok** subroutine returns a pointer to an occurrence of text tokens in the string pointed to by the *String1* parameter. The *String2* parameter specifies a set of code points as token delimiters. If the *String1* parameter is anything other than **NULL**, then the **NLstrtok** subroutine reads the string pointed to by the *String1* parameter until it finds one of the delimiter code points specified by the *String2* parameter. It then stores a null character into the string, replacing the delimiter code point, and returns a pointer to the first code point of the text token. The **NLstrtok** subroutine keeps track of its position in the string so that subsequent calls with a **NULL** *String1* parameter step through the string. The delimiters specified by the *String2* parameter can be changed for subsequent calls to **NLstrtok**. When no tokens remain in the string pointed to by the *String1* parameter, the **NLstrtok** subroutine returns a **NULL** pointer.

Implementation Specifics

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

Related Information

The **NCstring** subroutines, **NLchar** subroutines, and **string** subroutines.

National Language Support Overview in *General Programming Concepts*.

NLstrtime or strftime Subroutine

Purpose

Formats time and date.

Library

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

Syntax

include <time.h>

char* NLstrtime (*String*, *Length*,
Format, *Tmdate*)
char **String*, ** Format*;
int *Length*;
struct tm **Tmdate*;

size_t strftime(*String*, *Length*, *Format*, *Tmdate*)
char **String*, **Format*;
size_t *Length*;
struc tm **Tmdate*;

Description

The **NLstrtime** and **strftime** subroutines converts the internal time and date specification of the **tm** structure, which is pointed to by the *Tmdate* parameter into a character string pointed to by the *String* parameter under the direction of the format string pointed to by the *Format* parameter. The **tm** structure values may be assigned by the user or generated by the **localtime** or **gmtime** subroutine. The resulting string is similar to the result of the **printf** *Format* parameter, and is placed in the memory location addressed by the *String* parameter. It has a maximum length of *Length* and terminates with a **NULL** character.

Many conversion specifications are the same as those used by the **date** command. The interpretation of some conversion specifications is affected by the values of environment variables for national language support.

The *Format* parameter is a character string containing two types of objects: plain characters that are simply placed in the output string, and conversion specifications that convert information from *Tmdate* into readable form in the output string. Each conversion specification is a sequence of this form:

%[[–]width][.precision]type

- A % (percent sign) introduces a conversion specification.
- An optional decimal digit string specifies a minimum field **width**. A converted value that has fewer characters than the field width is padded with spaces to the right. If the decimal digit string is preceded by a minus sign, padding with spaces occurs to the left of the converted value.

If no **width** is given, for numeric fields the appropriate default width is used with the field padded on the left with zeros as required. For strings, the output field is made exactly wide enough to contain the string.

- An optional **precision** value gives the maximum number of characters to be printed for the conversion specification. The precision value is a decimal digit string preceded by a period. If the value to be output is longer than the precision, it is truncated on the right.
- The **type** of conversion is specified by one or two conversion characters. The characters and their meanings are:

m	The month of the year is output as a number between 01 and 12.
h or b	The short month is output as a string established by the value of **NLSMONTH** in the NLS database (Jan, for example).
lh or B	The long month is output as a string established by the value of **NLLMONTH** in the NLS database (January, for example).
c	The date and time is output with the locale dependent date and time by the value of the **NLDATIM** environment variable.
d	The day of the month is output as a number between 01 and 31.
j	The Julian day of the year is output as a number between 001 and 366.
w	The day of the week is output as a number between 0 (Sunday) and 6.
a	The short day of the week is output as a string according to the value of **NLDAY** in the NLS database (Mon, for example).
la or A	The long day of the week is output according to the value of **NLLDAY** in the NLS database (Monday, for example).
y	The year is output as a number (without the century) between 00 and 99.
Y	The year is output as a number (with the century) between 0000 and 9999.
D	The format is fixed to return %m/%d/%y. (Example, 20 Jun 1990 will return 06/20/90.)
lD or x	The long date is output in the Format specified by the value of **NLLDATE** in the NLS database (Jul 04, 1986, for example).
sD	The short date is output in the Format specified by the value of (long date) **NLLDATE** in the NLS database, but the year is omitted (July 7, for example).
H	The hour of the day is output as a number between 00 and 23.
sH or I	The hour of the day is output as a number between 01 and 12.
M	The minute is output as a number between 00 and 59.
S	The second is output as a number between 00 and 61.
p	The A.M. or P.M. indicator is output as a string specified by the value of **NLTMISC** in the NLS database (am, for example).

z or Z The (standard or daylight–saving) time zone name is output as a string from the environment variable **TZ** (CDT, for example).

r The time is output as %I:%M:%S (11:07:50 pm, for example).

X The time is output in the format specified by the value of **NLTIME** in the NLS database (19:07:50, for example).

T The time is output as **HH:MM:SS**.

sT The time is output in the format specified by the value of **NLTIME** in the NLS database, but omitting the seconds (19:07, for example).

n Only a new–line character is output.

t Only a tab character is output.

x Date as describted in the NL database.

% The % (percent) character is output.

U The week number of the year (Sunday as the first day of the week). Output format is a decimal number (00, 53)

W The week number of the year (Monday as the first day of the week). Output format is a decimal number (00, 53)

Js The name of the Era as specified by the value of **NLYEAR** in the NLS database (**SHOWA**, for example)

Jy The year relative to the Era (counting from 1).

Parameters

String Pointer to the string to hold the formatted time.

Length Maximum length of string pointed to by the *String* parameter

Format Pointer to the format character string.

Tmdate Pointer to the time structure that is to be converted.

Return Values

The **NLstrtime** subroutine returns a pointer to the *String* parameter. The **strftime** subroutine returns the length of the *String* parameter.

Implementation Specifics

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

Related information

The **gmtime**, **localtime** subroutines, **NLtmtime** subroutine, **printf**, **fprintf**, **sprintf**, **NLprintf**, **NLfprintf** and **NLsprintf** subroutines.

The **date** command.

NLtmtime Subroutine

Purpose

Sets a time structure from string data.

Library

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

Syntax

#include (time.h)
int NLtmtime (*String*, *Format*, *Ptm*)
char **String, *Format*;
struct tm **Ptm*;

Description

The **NLtmtime** subroutine sets the fields in the **time** structure pointed to by the *Ptm* parameter with information in the string pointed to by the *String* parameter that is parsed according to the string pointed to by the *Format* parameter. For each field descriptor in the *Format* parameter, data is read from the *String* parameter and placed into appropriate fields of the **time** structure. The *Format* parameter is described by these rules:

- Each field descriptor begins with a **%** (percent) character.
- A mnemonic string of 1 or 2 characters follows the **%** (percent) character and indicates the type of field or fields being read.
- A blank character (tab, space, or new–line character) anywhere in the *Format* string causes all blank characters at the corresponding location in the *String* parameter to be skipped.
- Any character in the *Format* parameter that appears in a field descriptor, other than the blank character, must be matched exactly by the same character in the *String* parameter. If a mismatch occurs, **NLtmtime** stops processing and any information following the mismatch is ignored. The characters and their meanings are:

  m The month of the year is output as a number between 01 and 12.

  h The short month is output as a string established by the environment variable **NLSMONTH** (Jan, for example).

  lh The long month is output as a string established by the environment variable **NLLMONTH** (January, for example).

  d The day of the month is output as a number between 01 and 31.

  j The Julian day of the year is output as a number between 001 and 366.

  w The day of the week is output as a number between 0 and 6.

  a The short day of the week is output as a string according to the environment variable **NLSDAY** (Mon, for example).

  la The long day of the week is output according to the environment variable **NLLDAY** (Monday, for example).

y	The year is output as a number between 00 and 99.
Y	The year is output as a number between 0000 and 9999.
D	The date is output in the format specified by the environment variable **NLDATE** (05/05/86, for example).
lD	The long date is output in the format specified by the environment variable **NLLDATE** (Jul 04, 1986, for example).
sD	The short date is output in the format specified by the (long date) environment variable **NLLDATE**, but the year is omitted (July 7, for example).
H	The hour of the day is output as a number between 00 and 23.
sH	The hour of the day is output as a number between 01 and 12.
M	The minute is output as a number between 00 and 59.
S	The second is output as a number between 00 and 59.
p	The AM or PM indicator is output as a string specified by environmental variable **NLTMISC** (am, for example).
z	The (standard or daylight–saving) time zone name is output as a string from the environment variable **TZ** (CDT, for example).
r	The time is output in the format specified by the environment variable **NLTIME**, but using a 12–hour clock (7:07:50 pm, for example).
T	The time is output in the format specified by the environment variable **NLTIME** (19:07:50, for example).
sT	The time is output in the format specified by the environment variable **NLTIME**, but omitting the seconds (19:07, for example).

Japanese Language Support

Js	Specifies the particular Era (as specified by the **NLYEAR** environment variable) that is to be used for a following **%Jy**. A **Js** specification is required if there is more than one Era in the **NLYEAR** string, and the **Js** specification must preceed the **Jy**.
Jy	The year is output relative to the Era defined via **Js**.

The field descriptors are the same as those used by the **NLstrtime** subroutine except for those that do not specify information.

Parameters

String	Pointer to a text string to parse.
Format	Pointer to a format string which describes how to parse the string pointed to by the *String* parameter.
Ptm	Pointer to a **time** structure to set with the values obtained by parsing the string pointed to by the *String* parameter.

Implementation Specifics

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

Related Information

The **ctime, localtime, mktime, difftime, asctime, tzset** subroutines, **NLstrtime** subroutine, **scanf, NLscanf, NLfscanf,** and **NLsscanf** subroutines.

The **date** command.

NLvprintf, NLvfprintf, or NLvsprintf Subroutine

Purpose

Prints formatted output.

Library

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

Syntax

#include <stdio.h>
#include <stdarg.h>

int NLvprintf (*Format*, *PrintArgument***)**
**char ***Format*;
va_list *PrintArgument*;

int NLvfprintf (*Stream*, *Format*, *PrintArgument***)**
**FILE ***Stream*;
**char ***Format*;
va_list *PrintArgument*;

int NLvsprintf (*String*, *Format*, *PrintArgument***)**
**char ***String*, ***Format*;
va_list *PrintArgument*;

Description

The **NLvprintf**, **NLvfprintf**, and **NLvsprintf** subroutines format and print parameter lists in the same way that the **vprintf**, **vfprintf**, and **vsprintf** subroutines perform these operations.

The **NLvprintf**, **NLvfprintf**, and **NLvsprintf** subroutines are the same as the **printf**, **fprintf**, and **sprintf** subroutines, respectively, except that the Japanese Language Support subroutines 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

Format — Specifies 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 fetched from the parameter lists.

PrintArgument — Specifies arguments to be printed.

Stream — Specifies the output stream.

String — Specifies the buffer to which output is printed.

Implementation Specifics

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

Related Information

The **vprintf**, **vfprintf**, **vsprintf** subroutines, **printf**, **fprintf**, **sprintf**, **NLprintf**, **NLfprintf**, **NLsprintf** subroutines.

The **varargs** macros.

NLxin or _NLxin Subroutine

Purpose

Perform EBCDIC–to–AIX translation.

Library

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

Syntax

#include <NLxio.h>

int NLxin (*String1*, *String2*, *Number*)
char **String1*, **String2*;
int *Number*;

int _NLxin(*Character*)
int *Character*;

Description

The **NLxin** subroutine and the **_NLxin** macro perform EBCDIC–to–AIX translation based on the translation table named by the environment variable **NLIN**. If the **NLIN** environment variable is not defined or is invalid, the **NLxin** subroutine will use the default universal EBCDIC–to–AIX translation table.

The byte values from the array pointed to by the *String2* parameter are used to index into the translation table to obtain the AIX byte that is placed into the character array pointed to by the *String1* parameter. The translation proceeds on a byte–by–byte basis until a null–byte is encountered in the array pointed to by the *String2* parameter or the number of bytes specified by the *Number* parameter have been placed in the array pointed to by the String1 parameter.

The **_NLxin** macro has a restricted domain. The **_NLxin** macro requires a value specified by the *Character* parameter in the range of 0 to 255 decimal expressed as an integer. Arguments outside of the defined domain cause undefined results.

Parameters

String1	Pointer to an output buffer that is used to store the translated AIX data.
String2	Pointer to the null–terminated EBCDIC character data that is to be translated.
Number	The count of the number of bytes available in the *String1* parameter.
Character	Specifies value of the restricted domain.

Return Values

The **NLxin** subroutine returns the number of bytes placed in the *String1* parameter. The **_NLxin** macro returns the translation table value that corresponds to the *Character* parameter as an integer.

Implementation Specifics

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

Related Information

The **NLxstart** subroutine, **NLxout** subroutine.

National Language Support Overview in *General Programming Concepts.*

NLxout or _NLxout Subroutine

Purpose

Performs AIX to EBCDIC translation.

Library

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

Syntax

#include <NLxio.h>

int NLxout(*String1*, *String2*, *Number***)**
char **String1*, **String2*;
int *Number*;

int _NLxout(*Character***)**
int *Character*;

Description

The **NLxout** subroutine and the **_NLxout** macro perform AIX–to–EBCDIC translation based on the translation table named by the environment variable **NLOUT**. If **NLOUT** is not defined or is invalid, the **NLxout** subroutine will use the default universal AIX–to–EBCDIC translation table.

The **NLxout** subroutine uses the value of the environment variable **NLOUT** as a pathname to the AIX–to–EBCDIC translation table. Subsequent calls to the **NLxout** subroutine from the same process will use the translation table obtained at the first call.

The **_NLxout macro** has a restricted domain. The **_NLxout** macro requires a value of the *Character* parameter in the range of 0 to 255 decimal expressed as an integer. Arguments outside of the defined domain cause undefined results.

Parameters

String1	Pointer to an output buffer that is used to store the translated EBCDIC data.
String2	Pointer to the null–terminated ASCII character data that is to be translated.
Number	The count of the number of bytes available in *String1*.
Character	An integer value in the restricted domain of the **_NLxout** macro.

Return Values

The **NLxout** subroutine returns the number of bytes placed in the *String1* parameter. The **_NLxout** macro returns the translation table value that corresponds to the *Character* parameter as an integer.

Implementation Specifics

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

Related Information

The **NLxstart** subroutine, **NLxin** subroutine.

National Language Support Overview in *General Programming Concepts*.

NLxstart Subroutine

Purpose

Performs translation table initialization.

Library

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

Syntax

#include <NLxio.h>

void NLxstart( )

Description

The **NLxstart** subroutine performs translation table initialization based on the translation tables named by the environment variables **NLOUT** and **NLIN**. If the respective environment variable is not defined or is invalid, the **NLxstart** subroutine will use the respective default universal translation table. The **NLxstart** subroutine must be called prior to the invocation of the **_NLxout** or **_NLxin** macros.

Implementation Specifics

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

Related Information

The **NLxin** subroutine, **NLxout** subroutine.

National Language Support Overview in *General Programming Concepts*.

NLyesno Subroutine

Purpose

Determines forms of affirmative and negative responses.

Library

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

Syntax

int NLyesno (*String*)
char **String*;

Description

The **NLyesno** subroutine determines whether the *String* parameter represents an affirmative response, a negative response, or neither

The **NLyesno** subroutine performs this function by comparing the string in the *String* parameter with currently allowed affirmative and negative responses.

The currently allowed affirmative and negative responses are determined by the setting of the **LANG** and **LC_MESSAGES** variables, and are locale dependent.

Parameter

String	Pointer to a string.

Return Values

The **NLyesno** subroutine returns a value of 1 if the *String* parameter represents an affirmative response. **NLyesno** returns a 0 if the *String* parameter represents a negative response. Otherwise, the **NLyesno** subroutine returns –1, which generally means the calling code should prompt again for a proper response.

Implementation Specifics

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

Related Information

National Language Support Overview in *General Programming Concepts.*

nsleep, usleep or sleep Subroutine

Purpose

Suspends a current process from execution.

Library

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

Syntax

#include <sys/time.h>

int nsleep (*Rqtp*, *Rmtp***)**
**timestruc_t ****Rqtp*, ***Rmtp*;

int usleep (*Useconds***)**
unsigned int *Useconds*;

int sleep (*Seconds***)**
unsigned int *Seconds*;

Description

The **nsleep** subroutine is an extended form of the **sleep** subroutine. The current process shall be suspended from execution until either the time interval specified by the *Rqtp* parameter has elapsed, or a signal is delivered to the calling process and its action is to invoke a signal–catching function or to terminate the process, or the process is notified of an event via an event notification function. The suspension time may be longer than requested due to the scheduling of other activity by the system. Upon return, the location specified by the *Rmtp* parameter shall be updated to contain the amount of time remaining in the interval, or 0 if the full interval has elapsed.

Parameters

Rqtp	Time interval specified for suspension of execution.
Rmtp	Specifies the location of the process.
Seconds	Specifies time interval in seconds.
Useconds	Specifies time interval in microseconds.

Compatibility Interface

The **sleep** and **usleep** subroutines are provided to ensure compatibility with older versions of AIX, AT&T System V and BSD systems. They are implemented simply as front–ends to the **nsleep** subroutine. Programs linking with the **libbsd** library get a BSD compatible version of the **sleep** subroutine. The return value from the BSD compatible **sleep** subroutine has no significance and should be ignored.

Return Values

The **nsleep**, **sleep**, and **usleep** subroutines return a 0 if the requested time has elapsed.

If the **nsleep** subroutine returns a –1 , the notification of a signal or event was received and the *Rmtp* parameter is updated to the unslept amount (the requested time minus the time actually slept), and **errno** is set.

If the **sleep** subroutine returns because of a premature arousal due to delivery of a signal, the return value will be the unslept amount (the requested time minus the time actually slept) in seconds.

Error Codes

If the **nsleep** subroutine fails, –1 is returned and **errno** is set to one of the following error codes.

EINTR A signal was caught by the calling process and control has been returned from the signal–catching routine, or the process has been notified on an event via an event notification function.

EINVAL The *Rqtp* parameter specified a nanosecond value less than zero or greater than or equal to one 1000 million.

Implementation Specifics

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

odm_add_obj Subroutine

Purpose

Adds a new object into an object class.

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 the object class to add to and a pointer to the data structure that contains the object to be added.

The **odm_add_obj** subroutine opens and closes the object class around the add 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 — 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 structure *ClassName*_**CLASS** that was created by the **odmcreate** command.

DataStructure — 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 fails, a value of –1 is returned and the **odmerrno** variable is set to an error code.

Error Codes

Failure of the **odm_add_obj** subroutine sets the **odmerrno** variable to one of the following error codes:

ODMI_CLASS_DNE, ODMI_CLASS_PERMS, ODMI_INVALID_CLXN, ODMI_INVALID_PATH, ODMI_MAGICNO_ERR, ODMI_OPEN_ERR, ODMI_PARAMS, ODMI_READ_ONLY, ODMI_TOOMANYCLASSES

See ODM Error Codes on page B–1 for explanations of the ODM error codes.

Implementation Specifics

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

Related Information

The **odm_create_class** subroutine, **odm_rm_obj** subroutine.

The **odmcreate** command.

ODM Example Code and Output in *General Programming Concepts* for an example of a **.h** file.

Object Data Manager Overview (ODM) in *General Programming Concepts*.

odm_change_obj Subroutine

Purpose

Changes an object in the object class.

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 the object class to change and a pointer to the data structure that contains 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	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 structure *ClassName*_**CLASS** which is created by the **odmcreate** command.
DataStructure	Pointer to an instance of the C language structure corresponding to the object class referenced by the *ClassSymbol* parameter. The structure is declared in the **.h** file created by the **odmcreate** command and has the same name as the object class.

Return Values

Upon successful completion, a value of 0 is returned. If the **odm_change_obj** subroutine fails, a value of –1 is returned and the **odmerrno** variable is set to an error code.

Error Codes

Failure of the **odm_change_obj** subroutine sets the **odmerrno** variable to one of the following error codes:

ODMI_CLASS_DNE, ODMI_CLASS_PERMS, ODMI_INVALID_CLXN, ODMI_INVALID_PATH, ODMI_MAGICNO_ERR, ODMI_NO_OBJECT, ODMI_OPEN_ERR, ODMI_PARAMS, ODMI_READ_ONLY, ODMI_TOOMANYCLASSES

See ODM Error Codes on page B–1 for explanations of the ODM error codes.

Implementation Specifics

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

Related Information

The **odm_get_obj** subroutine.

The **odmcreate** command.

ODM Example Code and Output in *General Programming Concepts* for an example of a **.h** file.

Object Data Manager Overview (ODM) in *General Programming Concepts*.

odm_close_class Subroutine

Purpose

Closes an ODM object class.

Syntax

#include <odmi.h>

int odm_close_class (*ClassSymbol***)**
CLASS_SYMBOL *ClassSymbol*;

Description

The **odm_close_class** subroutine closes the specified object class.

Parameter

ClassSymbol 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 structure *ClassName*_**CLASS** that was created by the **odmcreate** command.

Return Values

Upon successful completion, a value of 0 is returned. If the **odm_close_class** subroutine fails, a value of –1 is returned and the **odmerrno** variable is set to an error code.

Error Codes

Failure of the **odm_close_class** subroutine sets the **odmerrno** variable to one of the following error codes:

ODMI_CLASS_DNE, ODMI_CLASS_PERMS, ODMI_INVALID_CLXN, ODMI_INVALID_PATH, ODMI_MAGICNO_ERR, ODMI_OPEN_ERR, ODMI_TOOMANYCLASSES

See ODM Error Codes on page B–1 for explanations of the ODM error codes.

Implementation Specifics

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

Related Information

The **odm_open_class** subroutine.

Object Data Manager Overview (ODM) in *General Programming Concepts.*

odm_create_class Subroutine

Purpose

Creates an object class.

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.

Parameter

ClassSymbol 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 fails, 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 ODM Error Codes on page B–1 for explanations of the ODM error codes.

Implementation Specifics

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

Related Information

The **odm_mount_class** subroutine.

The **odmcreate** command.

ODM Example Code and Output in *General Programming Concepts* for an example of a **.h** file.

Object Data Manager Overview (ODM) in *General Programming Concepts.*

odm_err_msg Subroutine

Purpose

Returns an error message string.

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* and an address in which to put the string pointer of the message string that corresponds to the input *ODMErrno*. If no corresponding message is found for the input *ODMErrno*, a null string is returned and the subroutine fails.

Parameters

ODMErrno The error code for which the message string is retrieved.

MessageString 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 fails, a value of –1 is returned, and the *MessageString* returned is a null string.

Example

```
#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 AIX Base Operating System (BOS) Runtime.

Related Information

ODM Error Codes on page B–1 for descriptions of error codes.

Object Data Manager Overview (ODM) in *General Programming Concepts.*

odm_free_list Subroutine

Purpose

Frees memory previously allocated for an **odm_get_list** subroutine.

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 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 */
};
```

Return Values

Upon successful completion, a value of 0 is returned. If the **odm_free_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_free_list** subroutine sets the **odmerrno** variable to one of the following error codes:

ODMI_MAGICNO_ERR, ODMI_PARAMS

See ODM Error Codes on page B–1 for explanations of the ODM error codes.

Implementation Specifics

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

Related Information

The **odm_get_list** subroutine.

Object Data Manager Overview (ODM) in *General Programming Concepts*.

odm_get_by_id Subroutine

Purpose

Retrieves an object from an ODM object class by its ID.

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	A class symbol of the form *ClassName*_**CLASS** which is declared in the **.h** file created by the **odmcreate** command.
ObjectID	An identifier retrieved from the corresponding *ClassName* structure of the object class.
ReturnData	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 fails, a value of –1 is returned and the **odmerrno** variable is set to an error code.

Error Codes

Failure of the **odm_get_by_id** subroutine sets the **odmerrno** variable to one of the following error codes:

ODMI_CLASS_DNE, ODMI_CLASS_PERMS, ODMI_INVALID_CLXN, ODMI_INVALID_PATH, ODMI_MAGICNO_ERR, ODMI_MALLOC_ERR, ODMI_NO_OBJECT, ODMI_OPEN_ERR, ODMI_PARAMS, ODMI_TOOMANYCLASSES

See ODM Error Codes on page B–1 for explanations of the ODM error codes.

Implementation Specifics

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

Related Information

The **odm_get_obj**, **odm_get_first**, or **odm_get_next** subroutine.

The **odmcreate** command.

ODM Example Code and Output in *General Programming Concepts* for an example of a **.h** file.

Object Data Manager Overview (ODM) in *General Programming Concepts*.

odm_get_list Subroutine

Purpose

Retrieves all objects in an object class that match the specified criteria.

Syntax

#include <odmi.h>

struct *ClassName* ***odm_get_list(***ClassSymbol,Criteria,ListInfo,MaxReturn,LinkDepth)*
struct *ClassName*_**CLASS** *ClassSymbol;*
char **Criteria;*
struct listinfo **ListInfo;*
int *MaxReturn;*
int *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 get 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
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 structure *ClassName*_**CLASS** that was created by the **odmcreate** command.

Criteria
A string that contains the qualifying criteria for selecting the objects to remove. For information on qualifying criteria, see Understanding ODM Object Searches in *General Programming Concepts.*

ListInfo
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
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
The number of levels to recurse for objects with **ODM_LINK** descriptors. A setting of **1** indicates only the top level is retrieved; **2** indicates **ODM_LINK**s will be followed from the top/first level only; **3** indicates **ODM_LINK**s will be followed at the first and second level, and so on.

Return Values

Upon successful completion, a pointer to an array of C language structures containing the objects is returned. This structure matches that described in the **.h** file that is returned from the **odmcreate** command. If no match is found, **NULL** is returned. If the **odm_get_list** subroutine fails, a value of –1 is returned and the **odmerrno** variable is set to an error code.

Error Codes

Failure of the **odm_get_list** subroutine sets the **odmerrno** variable to one of the following error codes:

ODMI_BAD_CRIT, **ODMI_CLASS_DNE**, **ODMI_CLASS_PERMS**, **ODMI_INTERNAL_ERR**, **ODMI_INVALID_CLXN**, **ODMI_INVALID_PATH**, **ODMI_LINK_NOT_FOUND**, **ODMI_MAGICNO_ERR**, **ODMI_MALLOC_ERR**, **ODMI_OPEN_ERR**, **ODMI_PARAMS**, **ODMI_TOOMANYCLASSES**

See ODM Error Codes on page B–1 for explanations of the ODM error codes.

Implementation Specifics

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

Related Information

The **odm_get_by_id**, **odm_get_obj**, or **odm_free_list** subroutine.

The **odmcreate** command.

ODM Example Code and Output in *General Programming Concepts* for an example of a **.h** file.

Object Data Manager Overview (ODM) in *General Programming Concepts*.

odm_get_obj, odm_get_first, or odm_get_next Subroutine

Purpose

Retrieves objects, one object at a time, from an ODM object class.

Syntax

#include <odmi.h>

struct *ClassName* ***odm_get_obj** (*ClassSymbol*, *Criteria*, *ReturnData*, *FIRST_NEXT*)
struct *ClassName* ***odm_get_first** (*ClassSymbol*, *Criteria*, *ReturnData*)
struct *ClassName* ***odm_get_next** (*ClassSymbol*, *ReturnData*)

CLASS_SYMBOL *ClassSymbol;*
char **Criteria;*
struct *ClassName *ReturnData;*
int *FIRST_NEXT;*

Description

The **odm_get_obj**, **odm_get_first**, and **odm_get_next** subroutines retrieve objects from ODM object classes and return the objects into C language structures defined by the **.h** file produced by the **odmcreate** command.

The **odm_get_obj** , **odm_get_first**, and **odm_get_next** subroutines open and close the specified object class if the object class was not previously opened. If the object class was previously opened, then the subroutines leave the object class open upon return.

Parameters

ClassSymbol — 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 structure *ClassName*_**CLASS** that was created by the **odmcreate** command.

Criteria — The string that contains the qualifying criteria for retrieval of the objects. For more information about qualifying criteria, see Understanding ODM Object Searches in *General Programming Concepts*.

ReturnData — 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.

FIRST_NEXT — Specifies whether to get the first object that matches the criteria, or to get the next object. Valid values are:

- **FIRST** — Retrieve the first object that matches the search criteria.
- **NEXT** — Retrieve the next object that matches the search criteria. The *Criteria* parameter is ignored if the *FIRST_NEXT* parameter is set to **NEXT**.

Return Value

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 fails, a value of –1 is returned and the **odmerrno** variable is set to an error code.

Error Codes

Failure of the **odm_get_obj, odm_get_first** or **odm_get_next** subroutine sets the **odmerrno** variable to one of the following error codes:

ODMI_BAD_CRIT, ODMI_CLASS_DNE, ODMI_CLASS_PERMS, ODMI_INTERNAL_ERR, ODMI_INVALID_CLXN, ODMI_INVALID_PATH, ODMI_MAGICNO_ERR, ODMI_MALLOC_ERR, ODMI_OPEN_ERR, ODMI_TOOMANYCLASSES

See ODM Error Codes on page B–1 for explanations of the ODM error codes.

Implementation Specifics

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

Related Information

The **odm_get_list**, **odm_rm_obj**, or **odm_rm_by_id** subroutine.

The **odmcreate** command.

ODM Example Code and Output in *General Programming Concepts* for an example of a **.h** file.

Object Data Manager Overview (ODM) in *General Programming Concepts*.

odm_initialize Subroutine

Purpose

Prepares ODM for use by an application.

Syntax

#include <odmin.h>
int odm_initialize( )

Description

The **odm_initialize** subroutine starts ODM for use with an application program. This subroutine must be called before any other ODM subroutine calls.

Return Value

Upon successful completion, a value of 0 is returned. If the **odm_initialize** subroutine fails, a value of –1 is returned and the **odmerrno** variable is set to an error code.

Error Codes

Failure of the **odm_initialize** subroutine sets the **odmerrno** variable to one of the following error codes:

ODMI_INVALID_PATH, **ODMI_MALLOC_ERR**

See ODM Error Codes on page B–1 for explanations of the ODM error codes.

Implementation Specifics

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

Related Information

The **odm_terminate** subroutine.

Object Data Manager Overview (ODM) in *General Programming Concepts.*

odm_lock Subroutine

Purpose

Puts an exclusive lock on the requested path name.

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 — A string containing the path name in the file system in which to locate object classes to lock or the path name to an object class to lock.

TimeOut — The amount of time, in seconds, to wait if another application or method holds a lock on the requested object class or classes.

TimeOut = **ODM_NOWAIT**
The **odmlock** subroutine fails if the lock cannot be granted immediately.

TimeOut = *Integer*
The **odmlock** subroutine waits the specified amount of seconds to retry a failed lock request.

TimeOut = **ODM_WAIT**
The **odmlock** subroutine waits until the locked path name is freed from its current lock, then locks it.

Return Values

Upon successful completion a lock identifier is returned. If the **odm_lock** subroutine fails, a value of –1 is returned and the **odmerrno** variable is set to an error code.

Error Codes

Failure of the **odm_lock** subroutine sets the **odmerrno** variable to one of the following error codes:

ODMI_BAD_LOCK, ODMI_BAD_TIMEOUT, ODMI_BAD_TOKEN, ODMI_LOCK_BLOCKED, ODMI_LOCK_ENV, ODMI_MALLOC_ERR, ODMI_UNLOCK

See ODM Error Codes on page B–1 for explanations of the ODM error codes.

Implementation Specifics

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

Related Information

The **odm_unlock** subroutine.

Object Data Manager Overview (ODM) in *General Programming Concepts*.

odm_mount_class Subroutine

Purpose

Retrieves the class symbol structure for the specified object class name.

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 Editor) 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** 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**.

Mounting a class that links to another object class recursively mounts to the linked class. However, if the recursive mount fails, the original **odm_mount_class** subroutine does not fail; the **CLASS_SYMBOL** is set up with a **NULL** link.

Parameter

ClassName 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 fails, a value of –1 is returned and the **odmerrno** variable is set to an error code.

Error Codes

Failure of the **odm_mount_class** subroutine sets the **odmerrno** variable to one of the following error codes:

> **ODMI_BAD_CLASSNAME, ODMI_BAD_CLXNNAME, ODMI_CLASS_DNE, ODMI_CLASS_PERMS, ODMI_CLXNMAGICNO_ERR, ODMI_INVALID_CLASS, ODMI_INVALID_CLXN, ODMI_MAGICNO_ERR, ODMI_MALLOC_ERR, ODMI_OPEN_ERR, ODMI_PARAMS, ODMI_TOOMANYCLASSES**

See ODM Error Codes on page B–1 for explanations of the ODM error codes.

Implementation Specifics

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

Related Information

The **odm_create_class** subroutine.

Object Data Manager Overview (ODM) in *General Programming Concepts.*

odm_open_class Subroutine

Purpose

Opens an ODM object class.

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	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* for the object class is returned. If the **odm_open_class** subroutine fails, 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, ODMI_CLASS_PERMS, ODMI_INVALID_PATH, ODMI_MAGICNO_ERR, ODMI_OPEN_ERR, ODMI_TOOMANYCLASSES

See ODM Error Codes on page B–1 for explanations of the ODM error codes.

Implementation Specifics

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

Related Information

The **odm_close_class** subroutine.

The **odmcreate** command.

ODM Example Code and Output in *General Programming Concepts* for an example of a **.h** file.

Object Data Manager Overview (ODM) in *General Programming Concepts.*

odm_rm_by_id Subroutine

Purpose

Removes objects specified by their IDs from an ODM object class.

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 — 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 structure *ClassName*_**CLASS** that was created by the **odmcreate** command.

ObjectID — Identifier 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 fails, a value of –1 is returned and the **odmerrno** variable is set to an error code.

Error Codes

Failure of the **odm_rm_by_id** subroutine sets the **odmerrno** variable to one of the following error codes:

ODMI_CLASS_DNE, ODMI_CLASS_PERMS, ODMI_FORK, ODMI_INVALID_CLXN, ODMI_INVALID_PATH, ODMI_MAGICNO_ERR, ODMI_MALLOC_ERR, ODMI_NO_OBJECT, ODMI_OPEN_ERR, ODMI_OPEN_PIPE, ODMI_PARAMS, ODMI_READ_ONLY, ODMI_READ_PIPE, ODMI_TOOMANYCLASSES

See ODM Error Codes on page B–1 for explanations of the ODM error codes.

Implementation Specifics

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

Related Information

The **odm_get_obj** subroutine.

Object Data Manager Overview (ODM) in *General Programming Concepts.*

odm_rm_class Subroutine

Purpose

Removes an object class from the file system.

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 — A class symbol identifier returned from the **odm_open_class** subroutine. If the **odm_open_class** subroutine has not been called, then this 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_rm_class** subroutine fails, a value of –1 is returned and the **odmerrno** variable is set to an error code.

Error Codes

Failure of the **odm_rm_class** subroutine sets the **odmerrno** variable to one of the following error codes:

ODMI_CLASS_DNE, ODMI_CLASS_PERMS, ODMI_INVALID_CLXN, ODMI_INVALID_PATH, ODMI_MAGICNO_ERR, ODMI_OPEN_ERR, ODMI_TOOMANYCLASSES, ODMI_UNLINKCLASS_ERR, ODMI_UNLINKCLXN_ERR

See ODM Error Codes on page B–1 for explanations of the ODM error codes.

Implementation Specifics

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

Related Information

The **odm_open_class** subroutine.

The **odmcreate** command.

Object Data Manager Overview (ODM) in *General Programming Concepts.*

odm_rm_obj Subroutine

Purpose

Removes objects from an ODM object class.

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 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 structure *ClassName*_**CLASS** that was created by the **odmcreate** command.

Criteria A string that contains the qualifying criteria for selecting the objects to remove. For information on qualifying criteria, see Understanding ODM Object Searches in *General Programming Concepts*.

Return Values

Upon successful completion, the number of objects deleted is returned. If the **odm_rm_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_rm_obj** subroutine sets the **odmerrno** variable to one of the following error codes:

ODMI_BAD_CRIT, ODMI_CLASS_DNE, ODMI_CLASS_PERMS, ODMI_FORK, ODMI_INTERNAL_ERR, ODMI_INVALID_CLXN, ODMI_INVALID_PATH, ODMI_MAGICNO_ERR, ODMI_MALLOC_ERR, ODMI_OPEN_ERR, ODMI_OPEN_PIPE, ODMI_PARAMS, ODMI_READ_ONLY, ODMI_READ_PIPE, ODMI_TOOMANYCLASSES

See ODM Error Codes on page B–1 for explanations of the ODM error codes.

Implementation Specifics

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

Related Information

The **odm_open_class** subroutine.

The **odmcreate** command.

Object Data Manager Overview (ODM) in *General Programming Concepts*.

odm_run_method Subroutine

Purpose

Runs a specified method.

Syntax

#include <odmi.h>

int odm_run_method(*MethodName,MethodParameters,NewStdOut,NewStdError***)**
char **MethodName*;
char **MethodParameters*;
char ***NewStdOut*;
char ***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 — 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 — A list of parameters for the specified method.

NewStdOut — The address of a pointer to the memory where the standard output of the method will be stored. If the *NewStdOut* parameter points to a **NULL** value (`*NewStdOut == NULL`), standard output is not captured.

NewStdError — 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 Value

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, ODMI_MALLOC_ERR, ODMI_OPEN_PIPE, ODMI_PARAMS, ODMI_READ_PIPE

See ODM Error Codes on page B–1 for explanations of the ODM error codes.

Implementation Specifics

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

Related Information

The **odm_get_obj** subroutine.

Object Data Manager Overview (ODM) in *General Programming Concepts.*

odm_set_path Subroutine

Purpose

Sets the default path for locating object classes.

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.

Parameter

NewPath A string containing 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 fails, 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, ODMI_MALLOC_ERR

See ODM Error Codes on page B–1 for explanations of the ODM error codes.

Implementation Specifics

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

odm_set_perms Subroutine

Purpose

Sets the default permissions for an ODM object class at creation time.

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.

Parameter

NewPermissions	An integer specifying the new default permissions.

Return Values

Upon successful completion, the current default permissions are returned. If the **odm_set_perms** subroutine fails, a value of –1 is returned.

Implementation Specifics

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

odm_terminate Subroutine

Purpose

Terminates an ODM session.

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 fails, a value of –1 is returned and the **odmerrno** variable is set to an error code.

Error Codes

Failure of the **odm_terminate** subroutine sets the **odmerrno** variable to one of the following error codes:

ODMI_CLASS_DNE, ODMI_CLASS_PERMS, ODMI_INVALID_CLXN, ODMI_INVALID_PATH, ODMI_LOCK_ID, ODMI_MAGICNO_ERR, ODMI_OPEN_ERR, ODMI_TOOMANYCLASSES, ODMI_UNLOCK

See ODM Error Codes on page B–1 for explanations of the ODM error codes.

Implementation Specifics

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

Related Information

The **odm_initialize** subroutine.

Object Data Manager Overview (ODM) in *General Programming Concepts.*

odm_unlock Subroutine

Purpose

Releases a lock put on a path name.

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.

Parameter

LockID	The lock identifier returned from the **odm_lock** subroutine.

Return Values

Upon successful completion a value of 0 is returned. If the **odm_unlock** subroutine fails, a value of –1 is returned and the **odmerrno** variable is set to an error code.

Error Codes

Failure of the **odm_unlock** subroutine sets the **odmerrno** variable to one of the following error codes:

ODMI_LOCK_ID, ODMI_UNLOCK

See ODM Error Codes on page B–1 for explanations of the ODM error codes.

Implementation Specifics

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

Related Information

The **odm_lock** subroutine.

Object Data Manager Overview (ODM) in *General Programming Concepts*.

open, openx, or creat Subroutine

Purpose

Opens a file for reading or writing.

Syntax

#include <fcntl.h>
#include <sys/mode.h>

int open (*Path*, *Oflag*[, *Mode*])
**char ***Path*;
int *Oflag*;
int *Mode*;

openx(*Path*,*OFlag*,*Mode*,*Extension*)
**char ***Path*;
int *OFlag*;
int *Mode*;
int *Extension*;

int creat(*Path*[, *Mode*])
**char ***Path*;
int *Mode*;

Description

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 **open**, with the addition of an *Extension* parameter, which is provided for device driver use. The **creat** subroutine is equivalent to an **open** 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.

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. This parameter is constructed by logically ORing together values described in the **sys/mode.h** header file.

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** header file and are described below.

Oflag Parameter Flag Values that 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: Exactly one of the file access values must be specified. Do not use **O_RDONLY**, **O_WRONLY**, or **O_RDWR** together. If none is set, **O_RDONLY** is assumed.

Oflag Parameter Flag Values that Specify Special Open Processing

O_CREAT If the file exists, this flag has no effect, except as noted under **O_EXCL**. 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.
- 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 *SetUserID* attribute (S_ISUID bit) is cleared.
  - The *SetGroupID* attribute (S_ISGID bit) is cleared.
  - The S_ISVTX attribute bit is cleared.

O_EXCL If **O_EXCL** and **O_CREAT** are set, the open fails if the file exists.

O_NSHARE Assures that no process has this file open and precludes subsequent opens. If the file is already open, this open will fail and return immediately unless the *Oflag* parameter also specifies **O_DELAY**.

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 open for writing or open with **O_NSHARE**, this open fails and return immediately unless the *Oflag* parameter also specifies **O_DELAY**.

O_DEFER The file is opened for deferred update. Changes to the file are not reflected on permanent storage until an **fsync()** is performed. If no **fsync()** is performed, the changes are discarded when the file is closed.

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 and is a regular file, and if the file is successfully opened **O_RDWR** or **O_WRONLY**:

- 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.

The **open** subroutine fails 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 already open with **O_RSHARE** or **O_NSHARE**
- The file does not allow write access.
- The file is already opened for deferred update.

Oflag Parameter Flag Values that Specify Type of Update

A program can request some control over when updates should be made permanent for a regular file opened for write access. The following *Oflag* parameter flag values specify the type of update performed:

O_SYNC If set, updates to regular files and writes to block devices that 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 **O_SYNC** is set), the program is assured that all data for the file has been written to the permanent storage, even if the file is also open for deferred update.

Oflag Parameter Flag Values that Define the Initial State of the Open File

O_APPEND The file pointer sets to the end of the file prior to each write operation.

O_DELAY Specifies that if the **open** could not succeed due to an inability to grant the required **O_RSHARE** or **O_NSHARE** access, the process blocks instead of being returned **ETXTBSY**.

O_NDELAY Opens with no delay.

O_NONBLOCK Specifies that the **open** should not block.

The **O_NDELAY** and **O_NONBLOCK** flags 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 Flag Values

The effect of **O_CREAT** is immediate, even if the file is opened with **O_DEFER**.

When opening a file with **O_NSHARE** or **O_RSHARE**, if the file is already open with conflicting access:

- If **O_DELAY** is clear (the default), the **open** fails immediately.
- If **O_DELAY** is set, the **open** blocks until there is no conflicting **open**. There is no deadlock detection for processes using **O_DELAY**.

When opening a file that has already been opened with **O_NSHARE**:

- If **O_DELAY** is clear (the default), the **open** fails immediately.
- If **O_DELAY** is set, the **open** blocks until there is no conflicting open.

When opening a file with **O_RDWR**, **O_WRONLY**, or **O_TRUNC**, and the file is already open with **O_RSHARE**:

- If **O_DELAY** is clear (the default), the **open** fails immediately.
- If **O_DELAY** is set, the **open** blocks until there is no conflicting **open**.

When opening a FIFO with **O_RDONLY**:

- If **O_NDELAY** and **O_NONBLOCK** 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 **O_NDELAY** or **O_NONBLOCK** is set, the **open** succeeds immediately even if no process has the FIFO open for writing.

When opening a FIFO with **O_WRONLY**:

- If **O_NDELAY** and **O_NONBLOCK** 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 **O_NDELAY** or **O_NONBLOCK** 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 non-blocking opens, such as a terminal device:

- If **O_NDELAY** and **O_NONBLOCK** are clear (the default), the **open** blocks until the device is ready or available.
- If **O_NDELAY** or **O_NONBLOCK** 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 **O_NDELAY, O_RSHARE, O_NSHARE** and **O_DELAY** on a specific device is documented in the description of the special file related to the device type.

Return Values

Upon successful completion, the file descriptor, a non-negative integer, is returned. Otherwise, a value of –1 is returned and the global variable **errno** is set to indicate the error.

Error Codes

The **open**, **openx** and **creat** subroutines fail and the named file is not opened if one or more of the following are true:

ENOENT — **O_CREAT** is not set and the named file does not exist.

EACCES — Type of access specified by the *Oflag* parameter is denied for the named file.

EISDIR — Named file is a directory and write access is required.

EMFILE	The system limit for open file descriptors per process has already been reached (**OPEN_MAX**).
ENFILE	The system file table is full.
ENXIO	Named file is a character special or block special file, and the device associated with this special file does not exist.
ENXIO	Named file is a multiplexed special file and either the channel number is outside of the valid range or no more channels are available.
ENXIO	**O_DELAY** or **O_NONBLOCK** is set, the named file is a FIFO, **O_WRONLY** is set, and no process has the file open for reading.
ETXTBSY	File is already open in a manner (**O_RSHARE** or **O_NSHARE**) that precludes this open.
ETXTBSY	**O_NSHARE** or **O_RSHARE** was requested with **O_NDELAY** set, and there is a conflicting **open**.
EEXIST	**O_CREAT** and **O_EXCL** are set and the named file exists.
EAGAIN	**O_TRUNC** is set and the named file contains a record lock owned by another process.
EINTR	A signal was caught during the **open** subroutine.
EROFS	Named file resides on a read-only file system and write access is required.
ENOSPC	Directory that would contain the new file cannot be extended.
EDQUOT	Directory in which the entry for the new link is being placed cannot be extended because the quota of disk blocks or i-nodes defined for the user on the file system containing the directory has been exhausted.

The **open**, **openx** and **creat** subroutines can also fail if additional errors on page A–1 occur.

Implementation Specifics

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

Related Information

The **chmod** subroutine, **close** subroutine, **fcntl** subroutine, **ioctl** subroutine, **lockfx** subroutine, **lseek** subroutine, **read** subroutine, **stat** subroutine, **umask** subroutine, **write** subroutine.

The **fcntl.h** header file, **sys/mode.h** header file.

opendir, readdir, telldir, seekdir, rewinddir, or closedir Subroutine

Purpose

Performs operations on directories.

Library

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

Syntax

#include <sys/types.h>
#include <dirent.h>

DIR *opendir (*DirectoryName***)**
**char ****DirectoryName*;

struct dirent *readdir (*DirectoryPointer***)**
**DIR ****DirectoryPointer*;

long telldir (*DirectoryPointer***)**
**DIR ****DirectoryPointer*;

void seekdir (*DirectoryPointer*, *Location***)**
**DIR ****DirectoryPointer*;
long *Location*;

void rewinddir (*DirectoryPointer***)**
**DIR ****DirectoryPointer*;

void closedir (*DirectoryPointer***)**
**DIR ****DirectoryPointer;*

Description

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.

The **readdir** subroutine returns a pointer to the next directory entry. The **readdir** subroutine returns entries for . and .., 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 **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 to 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.

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 value of **NULL** is returned and the global variable **errno** is set to indicate the error.

On successful completion, the **readdir** subroutine returns a pointer to an object of type **struct dirent**. Otherwise, a value of **NULL** is returned and the global variable **errno** is set to indicate the error. When the end of the directory is encountered, a value of **NULL** is returned and the global variable **errno** 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 global variable **errno** is set to indicate the error.

Error Codes

If the **opendir** subroutine fails, a value of **NULL** is returned and **errno** is set to the one of the following values:

EACCES	Search permission is denied for any component of *DirectoryName* or read permission is denied for *DirectoryName*.
ENAMETOOLONG	The length of the *DirectoryName* argument exceeds **PATH_MAX** or a path name component is longer than **NAME_MAX** while **POSIX_NO_TRUNC** is in effect.
ENOENT	The named directory does not exist.
ENOTDIR	A component of *DirectoryName* is not a directory.
EMFILE	Too many file descriptors are currently open for the process.
ENFILE	Too many file descriptors are currently open in the system.

If the **readdir** subroutine fails, a value of **NULL** is returned and **errno** is set to the following value:

EBADF	The *DirectoryPointer* argument does not refer to an open directory stream.

If the **closedir** subroutine fails, a value of –1 is returned and **errno** is set to the following value:

EBADF	The *DirectoryPointer* argument does not refer to an open directory stream.

Example

1. 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 AIX Base Operating System (BOS) Runtime.

Related Information

The **close** subroutine, **lseek** subroutine, **openx**, **open**, **creat** subroutines, **read**, **readv**, **readx**, **readvx** subroutines.

The **scandir**, **alphasort** subroutines.

pathconf or fpathconf Subroutine

Purpose

Retrieves file implementation characteristics.

Library

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

Syntax

long pathconf(*Path*, *Name***)**
char **Path*;
int *Name*;

long fpathconf(*FileDescriptor*, *Name***)**
int *FileDescriptor*;
int *Name*;

Description

The **pathconf** subroutine allows an application to determine the characteristics of operations supported by the file system underlying 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 *Path* or *FileDescriptor*, **pathconf** returns an error. Symbolic values for the *Name* parameter are defined in the **unistd.h** header file:

Attribute	**Meaning**
_PC_LINK_MAX	The maximum number of links to the file.
_PC_MAX_CANON	The maximum number of bytes in a canonical input line. This is applicable only to terminal devices.
_PC_MAX_INPUT	The maximum number of bytes allowed in an input queue. This is applicable only to terminal devices.
_PC_NAME_MAX	Maximum number of bytes in a file name (not including a terminating NULL). This may be as small as 14, but is never larger than 255. This is applicable only to a directory file.

_PC_PATH_MAX	Maximum number of bytes in a path name (not including a terminating NULL).
_PC_PIPE_BUF	Maximum number of bytes guaranteed to be written atomically. This is applicable only to a FIFO.
_PC_CHOWN_RESTRICTED	Returns 0 if the use of the **chown()** function is restricted to a process with appropriate privileges,.and 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 is applicable only to a directory file.
_PC_V_DISABLE	This is always 0; no disabling character is defined. This is applicable only to a terminal device.

Return Values

If the **pathconf** or **fpathconf** subroutine is successful, the specified parameter is returned. Otherwise, a value of –1 is returned and the global variable **errno** is set to indicate the error.

Error Codes

The **pathconf** and **fpathconf** subroutines fail if the following is true:

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 variable *Name* with the specified file.
ENAMETOOLONG	The length of the *Path* string 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.

Implementation Specifics

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

Related Information

The **limits.h** file, **unistd.h** file.

pause Subroutine

Purpose

Suspends a process until a signal is received.

Library

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

Syntax

int pause ( )

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 a value.

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 global variable **errno** to **EINTR**.

Implementation Specifics

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

Related Information

The **incinterval, alarm, settimer** subroutines, **kill, killpg** subroutines, **sigaction**, **signal, sigvec** subroutines, **wait, waitpid**, **wait3** subroutines.

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 have opened with the **popen** subroutine. The **pclose** subroutine waits for the associated process to end, and then returns the exit status of the command.

Warning: If the original processes and the process started with **popen** concurrently 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.

Some problems with an output filter can be prevented by taking care to flush 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.

Implementation Specifics

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

Related Information

The **fclose, fflush** subroutines, **fopen**, **freopen**, **fdopen** subroutines, **pipe** subroutine, **popen** subroutine, **wait**, **waitvm**, **wait3** subroutines.

perror Subroutine

Purpose

Writes a message explaining a subroutine error.

Library

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

Syntax

#include <errno.h>
void perror (*String***)**
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 blank, 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 global variable **errno**, which is set when an error occurs, but is not cleared when a successful call is made.

To simplify various message formats, an array of message strings is provided in **sys_errlist**.

Use the global variable **errno** as an index into this table 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 **sys_nerr** because new error codes can be added to the system before they are added to the table.

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 AIX Base Operating System (BOS) Runtime.

Related Information

The **printf** subroutine, **strerror** subroutine.

pipe Subroutine

Purpose

Creates an interprocess channel.

Library

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

Syntax

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 on file descriptor *FileDescriptor*[0] accesses the data written to *FileDescriptor*[1] on a first–in, first–out (FIFO) basis.

When writing, at least **PIPE_BUF** bytes of data are buffered by the pipe before the writing process is blocked. In addition, any write to **PIPE_BUF** is guaranteed to be atomic; that is, it is guaranteed that the data will not be interleaved with data written by other processes.

Parameter

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

Error Codes

The **pipe** subroutine fails 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	**OPEN_MAX**–1 or **OPEN_MAX** file descriptors are already open.
ENFILE	The system file table is full, or the device containing pipes has no free inodes.

Implementation Specifics

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

Related Information

The **read** subroutine, **select** subroutine, **write** subroutine.

The **ksh** command, **sh** command.

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 segment (text lock), its process private segment (data lock), or both its text and process private segments (process lock) into memory. This 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()** 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 **plock**, the user application must lower the maximum stack limit value using the **ulimit** subroutine.

Parameter

Operation Specifies one of the following operations:

PROCLOCK	Locks the text and data segments into memory (process lock).
TXTLOCK	Locks the text segment into memory (text lock).
DATLOCK	Locks the data segment 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 global variable **errno** is set to indicate the error.

Error Codes

The **plock** subroutine fails if one or more of the following are 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, a text lock, or a data lock already exists on the calling process.
EINVAL	The *Operation* parameter is equal to **TXTLOCK** and a text lock or a process lock already exists on the calling process.
EINVAL	The *Operation* parameter is equal to **DATLOCK** and data lock or a 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 AIX Base Operating System (BOS) Runtime.

Related Information

The **exec** subroutines, **_exit**, **exit**, **atexit** subroutines, **fork** subroutine.

plot Subroutine Family

Purpose

Performs graphic output.

Libraries

Graphics Libraries **libplot.a**, **libprint.a**, and **lib300.a**

Syntax

void openpl ( )

void erase ( )

void label (*s***)**
char **s*;

void line (*x1*, *y1*, *x2*, *y2***)**
int *x1*, *y1*, *x2*, *y2*;

void circle (*x*, *y*, *r***)**
int *x*, *y*, *r*;

void arc (*x*, *y*, *x0*, *y0*, *x1*, *y1***)**
int *x*, *y*, *x0*, *y0*, *x1*, *y1*;

void move (*x*, *y***)**
int *x*, *y*;

void cont (*x*, *y***)**
int *x*, *y*;

void point (*x*, *y***)**
int *x*, *y*;

void linemod (*s***)**
char **s*;

void space (*x0*, *y0*, *x1*, *y1***)**
int *x0*, *y0*, *x1*, *y1*;

void closepl ( )

Description

The **plot** subroutine family generates graphic output with little dependence on devices. The **space** subroutine must be used before any of these functions to declare the amount of space necessary. The **openpl** subroutine must be used before any of the others to open the device for writing. The **closepl** subroutine flushes the output.

The **circle** subroutine draws a circle of radius *r* with the center at the point (*x*, *y*).

The **arc** subroutine draws an arc of a circle with the center at the point (*x*, *y*) between the points (*x0*, *y0*) and (*x1*, *y1*).

String parameters to the **label** and **linemod** subroutines are terminated by null characters and must not contain new–line characters.

The **plot** subroutines appear in several separate libraries. The routines in the **libplot.a** library generate device–independent output. The **tplot** command interprets this output for a specific device.

The other versions of these routines each generate output for a specific device. You should normally redirect the output of the **libprint.a** library to the printer. The **tplot** commands allow you to save the output of the **libprint.a** library in a regular file and print it later.

On an IBM Graphics Printer, the horizontal distance between points is not the same as the vertical distance between points. This means that arcs and circles are drawn as ellipses. A square or rectangle is drawn with four calls to the **line** subroutine. To adjust for this, call the **space** subroutine with appropriate scaling factors.

Implementation Specifics

The **plot** subroutines are part of AIX Base Operating System (BOS) Runtime.

Files

plot file	Provides the graphics interface.
/usr/lib/libplot.a library	Produces output for **tplot** filters.
/usr/lib/libprint.a library	For an IBM PC Graphics Printer.
/usr/lib/lib300.a library	For DASI 300.
/usr/lib/lib300s.a library	For DASI 300s.
/usr/lib/lib300S.a library	For DASI 300S.
/usr/lib/lib450.a library	For DASI 450.
/usr/lib/lib4014.a library	For Tektronix 4014.

Related Information

The**graph** command and **tplot** command.

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 *struct pollfd ****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 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 **/sys/poll.h>** header file.

Nfdsmsgs The number of file descriptors and the number of message queues to check. The low–order 16 bits give the number of elements present in the array of **pollfd** structures, while the high–order 16 bits give the number of elements present 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 is a value of –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.

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** header file, can be used to separate these two values from the return value. If rc contains the value returned from the **poll** subroutine, then **NFDS**(rc) is the number of files reporting some event or error, and **NMSGS**(rc) is 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).

Upon failure, a value of –1 is returned and the global variable **errno** is set to indicate the error.

Error Codes

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

EAGAIN Allocation of internal data structures failed.

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 specifed 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 AIX Base Operating System (BOS) Runtime.

For compatibility with previous releases of the AIX Operating System and BSD systems, the **select** subroutine is also supported.

Related Information

The **select** subroutine.

popen Subroutine

Purpose

Initiates a pipe to a process.

Library

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

Syntax

#include <stdio.h>

FILE *popen (*Command*, *Type***)**
**char ***Command*, **Type*;

Description

The **popen** subroutine creates a pipe between the calling program and a shell command to be executed.

The **popen** subroutine returns a pointer to a **FILE** structure for the stream.

Warning: 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 taking care to flush 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 Value

The **popen** subroutine returns a **NULL** pointer if files or processes cannot be created, or if the shell cannot be accessed.

Implementation Specifics

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

Related Information

The **fclose**, **fflush** subroutines, **fopen**, **freopen**, **fdopen** subroutines, **pclose** subroutine, **pipe** subroutine, **wait**, **waitpid**, **wait3** subroutines.

printf, fprintf, sprintf, NLprintf, NLfprintf, or NLsprintf Subroutine

Purpose

Prints formatted output.

Library

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

Syntax

#include <stdio.h>

int printf (*Format* [, *Value*, **...**])
char **Format*;

int fprintf (*Stream*, *Format* [, *Value*, . . .])
FILE **Stream*;
char **Format*;

int sprintf (*String*, *Format* [, *Value*, . . .])
char **String*, **Format*;

int NLprintf (*Format* [, *Value*, . . .])
char **Format*;

int NLfprintf (*Stream*, *Format* [, *Value*, . . .])
FILE **Stream*;
char **Format*;

int NLsprintf (*String*, *Format* [, *Value*, . . .])
char **String*, **Format*;

Description

The **printf** subroutine converts, formats, and writes its *Value* parameters, under control of the *Format* parameter, to the standard output stream stdout. This subroutine provides conversion types to handle code points and **NLchars**. The **printf** and **NLprintf** subroutines are identical.

The **fprintf** subroutine converts, formats, and writes its *Value* parameters, under control of the *Format* parameter, to the output stream specified by its *Stream* parameter. This subroutine provides conversion types to handle code points and **NLchars**. The **fprintf** and **NLfprintf** subroutines are identical.

The **sprintf** subroutine converts, formats, and stores its *Value* parameters, under control of the *Format* parameter, into consecutive bytes starting at the address specified by the *String* parameter. The **sprintf** subroutine places a '\0' (null character) at the end. It is your responsibility to ensure that enough storage space is available to contain the formatted string. This subroutine provides conversion types to handle code points and **NLchars**. The **sprintf** and **NLsprintf** subroutines are identical.

All these subroutines work by calling the **_doprnt** subroutine, using variable–length argument facilities of the **varargs** macros.

Parameters

The *Format* parameter is a character string that contains two types of objects:

- Plain characters, which are copied to the output stream.
- Conversion specifications, each of which causes zero or more items to be fetched from the *Value* parameter list.

If there are not enough items for *Format* in the *Value* parameter list, the results are unpredictable. If more *Values* remain after the entire *Format* has been processed, they are ignored.

Each conversion specification in the *Format* parameter has the following syntax:

1. A % (percent) sign.
2. Zero or more *options*, which modify the meaning of the conversion specification. The *option* characters and their meanings are:

–
Left align within the field the result of the conversion.

+
Begin the result of a signed conversion with a sign (+ or –).

blank
Prefix a *blank* to the result if the first character of a signed conversion is not a sign. If both the *blank* and + options appear, the *blank* option is ignored.

#
Convert 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 0x or 0X prefixed to it. 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 zeros are not removed from the result.

B
Give field width and precision in bytes, rather than in code points, for conversions using the **s** or **S** conversion characters.

N
Convert each international character support code point in the converted string converts into a printable ASCII escape sequence that uniquely identifies the code point. This option affects the **s** and **S** conversion characters.

O
Pad to field width using leading zeros (following any indication of sign or base) for **d**, **i**, **o**, **u**, **x**, **X**, **e**, **E**, **f**, **g**, and **G** conversions; no space padding is performed. If the **O** and – flags both appear, the **O** flag will be ignored. For **d**, **i**, **o u**, **x**, and **X** conversions, if a precision is specified, the **O** flag is also ignored. For other conversions, the behavior is undefined.

For Japanese Language Support:

J This option can be used with all conversion characters that take an int, long, double, or float *Value* as an argument. The **J** flag, appearing with any of these numeric conversion, indicates that output such as characters, digits, signs, or padding blanks will be 2–byte codes and two columns wide. The **J** flag can also be used with the **%c**, **%s**, and **%S** conversion characters to indicate that padding should use double–width spaces.

- 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–adjustment 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 given, it is treated as 0. The precision specifies:
  - The minimum number of digits to appear for the **d**, **u**, **o**, **x**, or **X** conversions
  - The number of digits to appear after the decimal point for the **e** and **f** conversions
  - The maximum number of significant digits for the **g** conversion
  - The maximum number of characters to be printed from a string in the **s** conversion
- An optional **l** (the letter l), **h**, or **L** specifying that a following **d**, **u**, **o**, **x**, or **X** conversion character applies to, respectively, a **long** integer *Value*, a **short** integer *Value*, or a **double** integer *Value*.
- A character that indicates the type of conversion to be applied:

% Performs no conversion. Prints %.

d, i Accepts an integer *Value* 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 zeros. The default precision is **1**. The result of converting a zero value with a precision of zero is a null string. Specifying a field width with a zero as a leading character causes the field width value to be padded with leading zeros.

u Accepts an integer *Value* 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 zeros. The default precision is **1**. The result of converting a zero value with a precision of zero is a null string. Specifying a field width with a zero as a leading character causes the field width value to be padded with leading zeros.

o
: Accepts an integer *Value* 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 zeros. The default precision is **1**. The result of converting a zero value with a precision of zero is a null string. Specifying a field width with a zero as a leading character causes the field width value to be padded with leading zeros. An octal value for field width is not implied.

x, X
: Accepts an integer *Value* 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 zeros. The default precision is **1**. The result of converting a zero value with a precision of zero is a null string. Specifying a field width with a zero as a leading character causes the field width value to be padded with leading zeros.

f
: Accepts a float or double *Value* 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 zero, no decimal point appears.

e, E
: Accepts a float or double *Value* and converts it to the exponential form [–]*d.ddd*e+/–*dd.* There is one digit before the decimal point and 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 zero, 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, G
: Accepts a float or double *Value* and converts it in the style of the **e**, **E**, or **f** conversion characters, with the precision specifying the number of significant digits. Trailing zeros 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.

c
: Accepts and prints a **char** *Value.*

C
: Accepts and prints an **NLchar** *Value.*

lc
: Accepts and prints an **NLchar** *Value.*

wc
: Accepts and prints an **NLchar** *Value.*

s
: Accepts a *Value* as a string (character pointer), and characters from the string are printed until a '\0' (null character) is encountered or the number of 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 *Value* has a value of 0 or **NULL**, the results are undefined.

S The corresponding *Value* is taken to be a pointer to a string of the type **NLchar**. Characters from the string are printed until a '\0' (null character) is encountered or the number of 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 *Value* has a value of 0 or **NULL**, the results are undefined.

ls The corresponding *Value* is taken to be a pointer to a string of the type **NLchar**. Characters from the string are printed until a '\0' (null character) is encountered or the number of 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 *Value* has a value of 0 or **NULL**, the results are undefined.

ws The corresponding *Value* is taken to be a pointer to a string of the type **NLchar**. Characters from the string are printed until a '\0' (null character) is encountered or the number of 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 *Value* has a value of 0 or **NULL**, the results are undefined.

p Accepts a pointer to void. The value of the pointer is converted to a sequence of printable characters, the same as unsigned hexadecimal (x).

n Accepts a pointer to an integer into which is written the number of characters written to the output stream so far 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 fetched 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. No truncation occurs. However, a small precision can cause truncation on the right.

The **e**, **E**, **f**, and **g** formats represent the special floating–point values as follows:

Quiet NaN	+NaNQ or –NaNQ
Signalling NaN	+NaNS or –NaNS
+/–INF	+INF or –INF
+/–0	+0 or –0

The representation of the plus sign depends on whether the **+** or **blank** formatting option is specified.

The **printf** and the **NLS** extensions to the **printf** subroutines can handle a format string that enables the system to process elements of the argument list in variable order. In such a case, the normal conversion character % (percent sign) is replaced by "%"digit"$", where digit is a decimal number. Conversions are then applied to arguments in the list with ordinal digits, rather than to the next unused argument.

The following restrictions apply:

- The format passed to the **NLS** extensions can contain either the format of the conversion or the explicit or implicit argument number. These forms cannot be mixed within a single format string.
- The * (asterisk) specification for field width or precision is not permitted with the variable order %digit$ format.

The following interface is provided:

#include <varargs.h>
_doprnt (*Format, Arguments, Stream*)
char **Format*;
va_list **Arguments*;
FILE **Stream*;

Return Values

Upon successful completion, each of these subroutines returns the number of display characters in the output string rather than the number of bytes in the string. The value returned by **sprintf** and **wsprintf** do not include the final '\0' character. (The **NLprintf**, **NLfprintf** and **NLsprintf** subroutines use strings that can contain 2–byte **NLchars**.) The value returned by the **NLsprintf** and **NLprintf** subroutines does not include the final '\0' character. If an output error occurs, a negative value is returned.

Implementation Specifics

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

Related Information

The **conv** subroutine, **ecvt**, **fcvt**, **gcvt** subroutines, **putc**, **putchar**, **fputc**, **putw**, **putwc**, **putwchar**, **fputwc** subroutines, **scanf**, **fscanf**, **sscanf**, **NLscanf**, **NLfscanf**, **NLsscanf** subroutines, **wsprintf** subroutine.

National Language Support Overview in *General Programming Concepts*

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)**

short **ShortBuffer*;
struct prof **ProfBuffer*;
unsigned int *BufferSize*, *Offset*, *Scale*;

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 process' program counter (pc) is examined each clock tick if the process is the currently active process. The value of the *Offset* parameter is subtracted from the pc and 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* parameter / **sizeof(short)**, the corresponding **short** inside the *ShortBuffer* parameter is incremented.
- 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.

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 *BufferSize* 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** header 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 **profil()** 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 **profil()** 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:

(int *)RoutineName*

Scale	Specifies the mapping factor between the program counter and *ShortBuffer.*
ProfBuffer	Points to an array of **prof** structures.

Return Value

The **profil** subroutine always returns a value of 0. Otherwise, the global variable **errno** is set to indicate the error.

Error Codes

The **profil** subroutine fails 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.
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 AIX Base Operating System (BOS) Runtime.

Related Information

The **exec** subroutines, **fork** subroutine, **monitor**, **monstartup**, **moncontrol** subroutines.

The **prof** command.

psdanger Subroutine

Purpose

Defines the amount of free paging space available.

Syntax

#include <signal.h>
int psdanger (*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 *Signal* is SIGKILL then 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 a specific threshold, the return value is negative. If *Signal* is –1, the return value is the number of free paging space blocks available in the system.

Implementation Specifics

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

Related Information

The **swapon** subroutine, **swapqry** subroutine.

The **chps** command, **lsps** command, **mkps** command, **rmps** command, **swapon** command.

psignal Subroutine or sys_siglist Vector

Purpose

Prints system signal messages.

Library

Berkeley Compatibility Library (**libbsd.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 newline.

To simplify variant formatting of signal names, the vector of message strings **sys_siglist** is provided; the signal number can be used as an index in this table to get the signal name without the newline. The define NSIG defined in the **signal.h** 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 the signal. The signal number should be among those found in the **signal.h** header file.

String Specifies a string that is printed. Most usefully, the *String* parameter is the name of the program which incurred the signal.

Implementation Specifics

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

Related Information

The **sigvec** subroutine, **perror** subroutine.

ptrace Subroutine

Purpose

Traces the execution of another process.

Library

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

Syntax

#include <sys/reg.h>
#include <sys/ptrace.h>

int ptrace(*Request,Process,Address,*
*Data,Buffer***)**
int *Request*, *Process*, ***Address*, *Data*, ***Buffer;*

Description

The **ptrace** subroutine allows a process to control the execution of another process. The **ptrace** subroutine is primarily used by utility programs to implement breakpoint debugging. The **dbx** command is such a debugging utility.

The debugged process behaves normally until it encounters a signal, at which time it enters a stopped state and its debugging process is notified with the **wait** subroutine. When the process is in the stopped state, the debugger can examine and modify its memory image using the **ptrace** subroutine. Also, the process can cause the process to either 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.

If a traced process initiates an **exec** subroutine, it stops before executing the first instruction of the new image and shows the signal **SIGTRAP**.

Parameters

Request Determines the action to be taken by the **ptrace** subroutine and is one of the following values:

PT_TRACE_ME
This request must be issued by the debugged process that is to be traced. This request sets the process trace flag that causes the process to be left in a stopped state upon receipt of a signal, rather than the action specified by the **sigaction** subroutine. The *Process, Address,* and *Data* parameters are ignored, and the return value is not defined for this request. Do not issue this request if the parent process does not expect to trace the debugged process.

Note: The remainder of the requests can only be used by the debugger. For each request, the *Process* parameter is the process ID of the child process. The child process must be in a stopped state before these requests are made.

PT_LDINFO This request returns the loader information that allows the debugger to determine what object modules are loaded.

PT_ATTACH

This request allows a debugging process to attach a process that is already running and place it into trace mode for debugging. This request cannot be used if the target process is already being traced.

To debug another process, at least one of the following must be true:

- Either the real or the effective user ID of the debugging process matches the real or effective user ID of the process to be traced.
- The effective user ID of the debugging process has root user authority.

This request fails if the calling process does not meet these permission requirements, returning an error code of **EPERM**.

PT_DETACH

This request allows a debugged process, specified by the *Process* 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.

PT_MULTI This request turns on and off multiprocess debugging mode, 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 causes both the traced process and its newly created process to trap on the next instruction. If a traced process initiated an **exec** subroutine, it stops before executing the first instruction of the new image and shows the signal **SIGTRAP**.

Also, when multiprocess debugging mode is enabled, the following new values will be returned from a **wait** subroutine:

W_SEWTED Process stopped during **exec**.

W_SFWTED Process stopped during **fork**.

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)
```

PT_REGSET

This request writes the contents of all 16 general purpose registers to the area pointed to by the *Address* parameter. This area should be at least 64 bytes. The request fails if the *Address* parameter points to a location outside of the allocated address space of the process, and returns a value of –1, setting the value of **errno** to **EINVAL**.

PT_REATTACH

This request allows a new debugger, with the proper permissions, to trace a process that was already traced by another debugger. This request fails if the calling process does not meet the permission requirements, returning an error code of **EPERM**.

PT_READ_I or **PT_READ_D**

These requests return the **int** in the debugged process address space at the location pointed to by the *Address* parameter. Since on all machines currently supported by the AIX Version 3 Operating System instruction and data request **PT_READ_I** or request **PT_READ_D** can be used with equal results, the data parameter is ignored. These requests fail if the value of the *Address* parameter is not in the address space of the debugged process, in which case a value of –1 is returned, and the debugging process **errno** is set to **EIO**.

PT_READ_U

This request returns the **int** from the debugged process user area of the system's address space that is located at the offset given by the *Address* parameter. (For information about the user area, see the **sys/user.h** header file.) The value of the *Address* parameter must be in the range 0 to sizeof(struct user). The data parameter is ignored. This request fails if the *Address* parameter is outside the user area, in which case a value of –1 is returned to the debugged process and the debugging process **errno** is set to **EIO**.

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 **int** pointed to by the *Address* parameter. Since on all machines currently supported by the AIX Version 3 Operating System instruction and data address spaces are not separated, either request

PT_WRITE_I or request **PT_WRITE_D** 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. These requests fail if the *Address* parameter points to a location in a pure procedure space and a copy cannot be made. They also fail if the *Address* parameter is out of range. Upon failure, a value of –1 is returned to the debugging process and the debugging process **errno** is set to **EIO**.

PT_WRITE_U
This request writes the value of the data parameter into the debugged process user area of the system's address space at the **int** specified by the *Address* parameter. The value of the *Address* parameter is rounded down to the next **int(word)** boundary. The following values for the *Address* parameter are defined in the **sys/reg.h** header file, and they identify the only entries that can be modified. The contents of this file vary for different machine types.

PT_CONTINUE
This request causes 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. Any other pending signals are canceled. 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. This request fails if the data parameter is not 0 or a valid signal number, in which case a value of –1 is returned to the debugging process and the debugging process **errno** is set to **EIO**.

PT_KILL This request causes the process to terminate the same way it would with an **exit** subroutine.

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 which of the registers is to be returned. The *Data* and *Buffer* parameters are ignored. This request fails if the value of the *Address* parameter is not defined in the **sys/reg.h** file for the machine type on which the process is executing. In this case, the **ptrace** subroutine returns the value –1 and sets the debugging process **errno** to **EIO**.

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 which floating-point register, as defined in the **sys/reg.h** file for the machine type the process is running on.

Note: Depending on hardware configuration, there may not be any floating-point registers.

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. The *Buffer* parameter is ignored. Upon successful completion, the value of data is returned to the debugging process. This request fails if the value of the *Address* parameter is not between 0 and 15 inclusive. In this case, the **ptrace** subroutine returns the value –1 and sets the debugging process **errno** to **EIO**.

PT_WRITE_FPR
This request sets the floating-point register specified by the *Data* parameter to the value pointed to by the *Address* parameter.

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 *Buffer* parameter points to the location in the debugging process address space into which the data is to be copied. Upon successful completion, the **ptrace** subroutine returns the value of the data parameter. If an error occurs, the **ptrace** subroutine returns –1 and sets the debugging process **errno** to indicate the error. This request fails when one or more of the following are true:

EINVAL 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.

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 it must no be greater than 1024. The *Buffer* parameter points to the data in the debugging process address space to be copied.

Upon successful completion, the value of data is returned to the debugging process. If an error occurs, the **ptrace** subroutine returns –1 and sets the debugging process **errno** to indicate the error. This request fails when one or more of the following are true:

EINVAL 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.

Process Specifies the process ID.

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.

Error Codes

In general, the **ptrace** subroutine fails if one or more of the following are true:

EIO The *Request* parameter is not one of the values listed.

EIO The *Request* parameter is not valid for the machine type the process is executing on.

ESRCH The *Process* parameter identifies a process that does not exist, has not executed a **ptrace** call with the *Request* parameter **PT_TRACE_ME**, or a process that is not stopped.

Implementation Specifics

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

Related Information

The **exec** subroutines, **load** subroutine, **sigaction** subroutine, **unload** subroutine, **wait**, **waitpid**, **wait3** subroutines.

The **dbx** command.

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***)**
char *Character;*
FILE ***Stream*;**

int putchar(*Character***)**
char *Character;*

int fputc(*Character, Stream***)**
char *Character;*
FILE ***Stream*;**

int putw(*Word*, *Stream***)**
int *Word;*
FILE ***Stream*;**

Description

The **putc** macro writes the character *Character* 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) 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.

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 Pointer to the file structure of an open file.

Character A character to be written.

Word A word to be written (non–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.

Implementation Specifics

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

Related Information

The **fclose, fflush** subroutine, **feof, ferror, cleareer, fileno** subroutines, **fopen, freopen, fdopen** subroutines, **fread, fwrite** subroutines, **getc**, **fgetc**, **getchar**, **getw** subroutines, **getwc**, **fgetwc**, **getwchar** subroutines, **printf, fprintf, sprintf, NLprintf, NLfprintf, NLsprintf, wsprintf** subroutines, **putwc**, **fputwc**, **putwchar** subroutines, **puts, fputs** subroutines, **setbuf** subroutine.

putenv Subroutine

Purpose

Sets an environment variable.

Library

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

Syntax

int putenv (*String*)
char **String*;

Description

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, *EnvironmentPointer*, the third parameter to the main subroutine, is not changed.

The **putenv** subroutine uses the **malloc** subroutine to enlarge the environment.

Warning: 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.

Parameter

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 AIX Base Operating System (BOS) Runtime.

Related Information

The **exec** subroutine, **getenv** or **NLgetenv** 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*)
char **String*;

int fputs (*String*, *Stream*)
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 newline 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.

Parameters

String	Pointer to a string to be written to output.
Stream	Pointer to the **FILE** structure of an open file.

Return Values

Upon successful completion, the **puts** and **fputs** subroutines return the number of characters written. Both subroutines return **EOF** on an error. This happens if the routines try to write on a file that has not been opened for writing.

Implementation Specifics

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

Related Information

The **ferror**, **feof**, **clearerr**, **fileno** macros, **fopen**, **freopen**, **fdopen** subroutines, **fread**, **fwrite** subroutines, **gets**, **fgets** subroutines, **getws**, **fgetws** subroutines, **printf**, **fprintf**, **sprintf**, **NLprintf**, **NLfprintf**, **NLsprintf** subroutines, **putws**, **fputws** subroutines, **putc**, **putchar**, **fputc**, **putw** subroutines, **putwc**, **putwchar**, **fputwc** subroutines.

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>
int putwc(*Character*, *Stream***)**
int *Character*;
FILE **Stream*;

int putwchar(*Character***)**
int *Character*;

int fputwc(*Character*, *Stream***)**
int *Character*;

Description

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's buffering strategy.

The **putwc** subroutine writes the **wchar_t** specified by the *Character* parameter to the output *Stream* as 1 or 2 bytes.

The **putwchar** macro works like the **putwc** subroutine, except that **putwchar** writes the specified **wchar_t** to the standard output.

The **fputwc** subroutine works the same as **putwc**.

Parameters

Character	**wchar_t** to be written.
Stream	Output data.

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* is not open for writing, or if the output file size cannot be increased.

Implementation Specifics

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

Related Information

The **fclose, fflush** subroutines, **fopen, freopen, fdopen** subroutines, **feof, ferror, cleareer, fileno** subroutines, **fread, fwrite** subroutines, **getc**, **fgetc**, **getchar**, **getw** subroutines, **getwc**, **fgetwc**, **getwchar** subroutines, **printf, fprintf, sprintf, NLprintf, NLfprintf, NLsprintf, wsprintf** subroutines, **putc**, **putchar**, **fputc**, **putw** subroutines, **puts, fputs** subroutines, **setbuf** subroutine.

National Language Support Overview in *General Programming Concepts.*

putws or fputws Subroutine

Purpose

Writes a string to a stream.

Library

Standard I/O Library (**libc.a**)

Japanese Language Support Syntax

When running AIX with Japanese Language Support on your system, the following subroutines, stored in **libc.a**, are provided:

#include <stdio.h>
#include <NLchar.h>

int putws (*String*)
NLchar **String*;

int fputws (*String*, *Stream*)
NLchar **String*;
FILE **Stream*;

Description

The **putws** subroutine writes the **NLchar** string pointed to by the *String* parameter to the standard output stream, **stdout**. In this case, each element of the *String* parameter produces either 1 or 2 bytes of output, according to the size required for its encoding. In all other respects, **putws** functions like **puts**.

The **fputws** subroutine writes the **NLchar** string pointed to by the *String* parameter to the output stream. Again, each element of the *String* parameter produces either 1 or 2 bytes of output, according to the size required for its encoding. In all other respects, **fputws** functions like **fputs**.

Parameters

String Pointer to a string to be writen to output.

Stream Pointer to the **FILE** structure of an open file.

Return Values

Upon successful completion, the **putws** and **fputws** subroutines return the number of characters written. Both subroutines return **EOF** on an error. This happens if the routines try to write on a file that has not been opened for writing.

Implementation Specifics

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

Related Information

The **ferror**, **feof**, **clearerr**, **fileno** macros, **fopen**, **freopen**, **fdopen** subroutines, **fread**, **fwrite** subroutines, **gets**, **fgets** subroutines, **getws**, **fgetws** subroutines, **puts**, **fputs** subroutines, **printf**, **fprintf**, **sprintf**, **NLprintf**, **NLfprintf**, **NLsprintf** subroutines, **putc**, **putchar**, **fputc**, **putw** subroutines, **putwc**, **putwchar**, **fputwc** subroutines.

National Language Support Overview in *General Programming Concepts*.

qsort Subroutine

Purpose

Sorts a table of data in place.

Library

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

Syntax

void qsort (*Base*, *NumberOfElements*, *Size*, *ComparisonPointer*)

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

Description

The **qsort** subroutine sorts a table of data in place. It uses the quicker-sort algorithm.

Parameters

Base	Points to the element at the base of the table.
NumberOfElements	Specifies the number of elements in the table.
Size	Specifies the size of each element.
ComparisonPointer	Points to the comparison function, which is passed two parameters that point to the objects being compared.

Return Values

The comparison function must compare its parameters and return 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 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.

Note: If two items are the same when compared, their order in the output of this subroutine is unpredictable.

The pointer to the base of the table should be of type pointer-to-element, and cast to type pointer-to-character.

Implementation Specifics

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

Related Information

The **bsearch** subroutine, **lsearch** subroutine.

raise Subroutine

Purpose

Sends a signal to the executing program.

Library

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

Syntax

#include <sys/signal.h>

int raise(*Signal***)**
int *Signal*;

Description

The **raise** subroutine sends the signal specified by the *Signal* parameter to the executing program. It is equivalent to the following:

```
ProcessID = getpid( );
error = kill(ProcessID, Signal);
```

Parameter

Signal Specifies a signal number.

Return Values

Upon successful completion of the **raise** subroutine, a value of 0 is returned. Otherwise, a nonzero value is returned and the global variable **errno** is set to indicate the error.

Implementation Specifics

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

Related Information

The **_exit** subroutine, **kill** subroutine, **sigaction** subroutine.

rand or srand Subroutine

Purpose

Generates pseudo-random numbers.

Library

Standard C Library (**libc.a**)
Berkeley Compatibility Library (**libbsd.a**)

Syntax

#include <stdlib.h>

int rand ( )

void srand (*Seed***)**
unsigned int *Seed*;

Description

The **rand** subroutine generates a random number using a multiplicative congruential algorithm. The random-number generator has a period of 2**31, and it returns successive pseudo-random numbers in the range from 0 to 2**15 –1.

The **srand** subroutine resets the random-number generator to a random starting point. The generator is initially seeded with a value of 1.

Note: The **rand** subroutine is a simple random-number generator. Its spectral properties, the mathematical measurement of how random the number sequence is, are somewhat limited. See the **drand48** subroutine or the **random** subroutine for more elaborate random-number generators that have better spectral properties.

Parameter

Seed Specifies an initial seed value.

Implementation Specifics

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

The BSD version of the **rand** subroutine returns a number in the range 0 to 2**31 –1, rather than 0 to 2**15 –1, and can be used by compiling with the Berkeley Compatibility Library (**libbsd.a**).

There are better random number generators, as noted above; however, the **rand** and **srand** subroutines are the interfaces defined for the ANSI C library.

The following functions define the semantics of the **rand** and **srand** subroutines, and are included here to facilitate porting applications from different implementations:

```
static unsigned int next = 1;

int rand( )
{
    next = next X 1103515245 + 12345;

    return ((next >>16) & 32767);

}

void srand (Seed)
int Seed;
{

    next = Seed;

}
```

Related Information

The **drand48, erand48, lrand48, nrand48, mrand48, jrand48, srand48, seed48, lcong48** subroutines, **random, srandom, initstate, setstate** subroutines.

random, srandom, initstate, or setstate Subroutine

Purpose

Generates "better" pseudo-random numbers.

Library

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

Syntax

long random ( )

srandom (*Seed*)
int *Seed*;

char *initstate (*Seed*, *State*, *Number*)
unsigned *Seed*;
char **State*;
int *Number*;

char *setstate (*State*)
char **State*;

Description

The **random** subroutine and **srandom** subroutine have almost the same calling sequence and initialization properties as the **rand** subroutine and **srand** subroutine. The difference is that the **rand** subroutine produces a much less random sequence; in fact, the low dozen bits generated by the **rand** subroutine go through a cyclic pattern. All the bits generated by the **random** subroutine are usable. For example, "random( )&01" produces a random binary value.

The **srandom** subroutine, unlike the **srand** subroutine, does not return the old seed because the amount of state information used is more than a single word. The **initstate** subroutine and **setstate** subroutine handle restarting and changing random-number generators. Like the **rand** subroutine, however, the **random** subroutine by default produces a sequence of numbers that can be duplicated by calling the **srandom** subroutine with 1 as the seed.

The **initstate** subroutine allows a state array, passed in as an argument, to be initialized for future use. The size of the state array (in bytes) is used by the **initstate** subtroutine, to decide how sophisticated a random-number generator it should use; the larger the state array, the more random the numbers are. Values for the amount of state information are: 8, 32, 64, 128, and 256 bytes. Amounts less than 8 bytes generate an error, while other amounts are rounded down to the nearest known value. The *Seed* parameter specifies a starting point for the random-number sequence and provides for restarting at the same point. The **initstate** subroutine returns a pointer to the previous state information array.

Once a state has been initialized, the **setstate** subroutine allows rapid switching between states. The array defined by *State* parameter is used for further random-number generation until the **initstate** subroutine is called or the **setstate** subroutine is called again. The **setstate** subroutine returns a pointer to the previous state array.

After initialization, a state array can be restarted at a different point in one of two ways:

- The **initstate** subroutine can be used, with the desired seed, state array, and size of the array, or
- The **setstate** subroutine, with the desired state, can be used, followed by the **srandom** subroutine with the desired seed. The advantage of using both of these subroutines is that the size of the state array does not have to be saved once it is initialized.

Parameters

Seed	Specifies an initial seed value.
State	Points to the array of state information.
Number	Specifies the size of the state information array.

Error Codes

If the **initstate** subroutine is called with less than 8 bytes of state information, or if the **setstate** subroutine detects that the state information has been damaged, error messages are sent to the standard output.

Implementation Specifics

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

The **random** subroutine uses a non-linear additive feedback random number generator employing a default state array size of 31 long integers to return successive pseudo-random number in the range from 0 to 2**31–1. The period of this random number generator is very large, approximately 16 * (2**31–1). The size of the state array determines the period of the random number generator. Increasing the state array size increases the period.

With a full 256 bytes of state information, the period of the random-number generator is greater than 2**69, which should be sufficient for most purposes.

Related Information

The **drand48, erand48, jrand48, lcong48, lrand48**, **mrand48**, **nrand48**, **seed48, srand48** subroutines, **rand**, **srand** subroutines.

re_comp or re_exec Subroutine

Purpose

Regular expression handlers.

Library

Berkeley Compatibility Library (**libbsd.a**)

Syntax

char re_comp(*String***)**
char ***String*;**

re_exec(*String***)**
char ***String*;**

Description

The **re_comp** subroutine compiles a string into an internal form suitable for pattern matching. The **re_exec** subroutine checks the argument *String* against the last string passed to **re_comp**.

The strings passed to both **re_comp** and **re_exec** may have trailing or embedded newline characters; they are terminated by nulls. The regular expressions recognized are described in the manual entry for **ed**, given the above difference.

Parameter

String	Specifies the string to be compiled by **re_comp** and checked by **re_exec**.

Return Values

The **re_comp** subroutine returns 0 if the string pointed to by the *String* parameter was compiled successfully; otherwise a string containing an error message is returned. If **re_comp** is passed 0 or a null string, it returns without changing the currently compiled regular expression.

The **re_exec** subroutine returns 1 if the string pointed to by the *String* parameter matches the last compiled regular expression, 0 if the *String* parameter failed to match the last compiled regular expression, and –1 if the compiled regular expression is not valid (indicating an internal error).

The **re_exec** subroutine returns –1 for an internal error.

If an error occurs, the **re_comp** subroutine returns one of the following strings:

- No previous regular expression
- Regular expression too long
- Unmatched \(
- Too many \(\() pairs
- Missing]
- Unmatched \)

Implementation Specifics

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

Related Information

The **ed** command, **egrep** command, **fgrep** command, **grep** command.

read, readx, readv, or readvx Subroutine

Purpose

Reads from a file.

Syntax

```
int read(FileDescriptor, Buffer, NBytes)
int FileDescriptor;
char *Buffer;
unsigned int NBytes;

int readx(FileDescriptor, Buffer, NBytes, Extension)
int FileDescriptor, Extension;
char *Buffer;
unsigned int NBytes;
int Extension;

#include <sys/types.h>
#include <sys/uio.h>

int readv(FileDescriptor, iov, iovCount)
int FileDescriptor;
struct iovec *iov;
int iovCount;

int readvx(FileDescriptor,iov,iovCount,Extension)
int FileDescriptor;
struct iovec *iov;
int iovCount;
int Extension;
```

Description

The **read** subroutine attempts to read *NBytes* of data from the file associated with the *FileDescriptor* parameter into the buffer pointed to by the the *Buffer* parameter.

The **readv** subroutine performs the same action but scatters the input data into the *iovCount* buffers specified by the array of **iovec** structures pointed to by the *iov* parameter. Each **iovec** entry specifies the base address and length of an area in memory where data should be placed. **readv** always fills an area completely before proceeding to the next.

readx and **readvx** are the same as **read** and **readv**, respectively, with the addition of an *Extension* parameter, which is needed when reading from some device drivers and when reading directories. While directories can be read directly, it is recommended that the **opendir** and **readdir** calls be used instead, as this is a more portable interface.

On regular files and devices capable of seeking, the **read** starts at a position in the file given by the file pointer associated with the *FileDescriptor* parameter. Upon return from the **read** subroutine, the file pointer is incremented by the number of bytes actually read.

Devices that are incapable of seeking always read from the current position. The value of a file pointer associated with such a file is undefined.

On directories, the **readvx** subroutine starts at the position specified by the file pointer associated with the *FileDescriptor* parameter. The value of this file pointer must be either 0 or a value which the file pointer had immediately after a previous call to the **readvx** subroutine on this directory. Upon return from the **readvx** subroutine, the file pointer is incremented by a number that may not correspond to the number of bytes copied into the buffers

When attempting to read from an empty pipe (or FIFO):

- If no process has the pipe open for writing, the **read** returns 0 to indicate end–of–file.
- If some process has the pipe open for writing:
  - If **O_NDELAY** and **O_NONBLOCK** are clear (the default), the **read** will block until some data is written or the pipe is closed by all processes that had opened the pipe for writing.
  - If **O_NDELAY** is set, the **read** subroutine returns a value of 0.
  - If **O_NONBLOCK** is set, the **read** subroutine returns a value of –1 and set the global variable **errno** to **EAGAIN**.

When attempting to read from a character special file that supports non–blocking reads, such as a terminal, and no data is currently available:

- If **O_NDELAY** and **O_NONBLOCK** are clear (the default), the **read** subroutine blocks until data becomes available.
- If **O_NDELAY** is set, the **read** subroutine returns 0.
- If **O_NONBLOCK** is set, the **readvx** subroutine returns –1 and sets the global variable **errno** to **EAGAIN** if no data is available.

When attempting to read a regular file that supports enforcement mode record locks, and all or part of the region to be read is currently locked by another process:

- If **O_NDELAY** and **O_NONBLOCK** are clear, the **read** blocks the calling process until the lock is released.
- If **O_NDELAY** or **O_NONBLOCK** is set, the **read** returns –1 and sets the global variable **errno** to **EAGAIN**.

The behavior of an interrupted **read** subroutine depends on how the handler for the arriving signal was installed.

Note: A read from a regular file is not interruptible. Only reads from objects that may block indefinitely, such as FIFOs, sockets, and some devices, are generally interruptible.

If the handler was installed with an indication that subroutines should not be restarted, the **read** subroutine returns a value of –1 and the global variable **errno** is set to **EINTR** (even if some data was already consumed).

If the handler was installed with an indication that subroutines should be restarted:

- If no data had been read when the interrupt was handled, this **read** will not return a value (it is restarted*)*.
- If data had been read when the interrupt was handled, this **read** subroutine returns the amount of data consumed.

Parameters

FileDescriptor A file descriptor identifying the object to be read.

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 value of the *Extension* parameter is 0.

For directories, the *Extension* parameter determines the format in which directory entries should be returned:

- If the value of the *Extension* parameter is 0, the format in which directory entries are returned depends on the value of the **real directory read** flag (described in **ulimit** subroutine).
- If the calling process does not have the **real directory read** flag set, the buffers are filled with an array of directory entries truncated to fit the format of the System V directory structure. This provides compatibility with programs written for UNIX System V.
- If the calling process has the **real directory read** flag set (see the **ulimit** subroutine), the buffers are filled with an image of the underlying implementation of the directory.
- If the value of the *Extension* parameter is 1, the buffers are filled with consecutive directory entries in the format of a **dirent** structure. This is logically equivalent to the **readdir** subroutine.
- Other values of the *Extension* parameter are reserved.

iov Points to an array of **iovec** structures that identifies the buffers into which the data is to be placed. The **iovec** structure is defined in the **sys/uio.h** header file and contains the following members:

```
caddr_t  iov_base;
int      iov_len;
```

iovCount Specifies the number of **iovec** structures pointed to by the *iov* parameter.

Buffer Points to the buffer.

NBytes Specifies the number of bytes read from the file associated with the *FileDescriptor* parameter.

Return Values

Upon successful completion, the **read**, **readx**, **readv** and **readvx** subroutines return the number of bytes actually read and placed into buffers. The system guarantees to read the number of bytes requested if the descriptor references a normal file that has the same number of bytes left before the end–of–file, but in no other case.

A value of 0 is returned when the end of the file has been reached. (For information about communication files, see the **ioctl** and **termio** files.)

Otherwise, a value of –1 is returned and the global variable **errno** is set to identify the error.

Error Codes

The **read**, **readx**, **readv** and **readvx** subroutines fail if one or more of the following are true:

EBADF	The *FileDescriptor* parameter is not a valid file descriptor open for reading.
EINVAL	The file position pointer associated with the *FileDescriptor* parameter was negative.
EINVAL	The sum of the **iov_len** values in the *iov* array was negative or overflowed a 32–bit integer.
EINVAL	The value of the *iovCount* parameter was not between 1 and 16, inclusive.
EAGAIN	The file was marked for non–blocking I/O, and no data was ready to be read.
EFAULT	The *Buffer* or part of the *iov* points to a location outside of the allocated address space of the process.
EDEADLK	A deadlock would occur if the calling process were to sleep until the region to be read was unlocked.
EINTR	A **read** was interrupted by a signal before any data arrived, and the signal handler was installed with an indication that subroutines are not to be restarted.
EIO	An I/O error occurred while reading from the file system.

If Network File System is installed on the system, the **read** system call can also fail if the following is true:

ETIMEDOUT	The connection timed out.

Implementation Specifics

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

Related Information

The **fcntl** subroutine, **ioctl** subroutine, **lockfx** subroutine, **lseek** subroutine, **open** subroutine, **pipe** subroutine, **poll** subroutine, **socket** subroutine, **socketpair** subroutine.

The **opendir**, **readdir**, **seekdir** subroutines.

readlink Subroutine

Purpose

Reads the contents of a symbolic link.

Library

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

Syntax

int readlink (*Path,Buffer,BufferSize*)
char **Path*;
char **Buffer*;
int *BufferSize*;

Description

The **readlink** subroutine places the contents of the symbolic link named by the *Path* parameter in the buffer *Buffer*, which has size *BufferSize*.

Parameters

Path	The path name of the destination file or directory.
Buffer	A pointer to the user's buffer. The buffer should be at least as large as the *BufferSize* parameter.
BufferSize	The size of the buffer. The contents of the link are not **NULL** terminated. A symbolic link cannot have more than **MAXLINKLEN** bytes including the **NULL**, so **MAXLINKLEN** is an appropriate buffer size.

Return Values

Upon successful completion, the **readlink** subroutine returns a count of the number of characters placed in the buffer. Otherwise, a value of –1 is returned and the global variable **errno** is set to indicate the error.

Error Codes

The **readlink** subroutine fails if one or both of the following are true:

ENOENT	The file named by the *Path* parameter does not exist.
EINVAL	The file named by the *Path* parameter is not a symbolic link.

The **readlink** subroutine can also fail if additional errors on page A–1 occur.

If Network File System is installed on the system, the **readlink** subroutine can also fail if the following is true:

ETIMEDOUT	The connection timed out.

Implementation Specifics

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

Related Information

The **link** subroutine, **statx**, **fstatx** subroutines, **symlink** subroutine, **unlink** subroutine.

The **ln** command.

reboot Subroutine

Purpose

Restarts the system.

Library

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

Syntax

#include <sys/reboot.h>

void reboot (*HowTo, Argument*)
int *HowTo;*
void **Argument*;

Description

The **reboot** subroutine restarts (re–IPLs) the system. The startup is automatic and brings up /**unix** in the normal, nonmaintenance mode.

The calling process must have root user authority in order to run this subroutine successfully.

Warning: Users of the **reboot** subroutine are not portable. The **reboot** subroutine is intended for use only by the **halt**, **reboot**, and **shutdown** commands.

Parameters

HowTo Specifies one of the following values:

RB_SOFTIPL Soft IPL.

RB_HALT Halt operator, power off.

RB_POWIPL Halt operator, power off, wait a specified length of time, then power on.

The programmed power off and programmed power on are supported by the Model 930 system.

Argument Specifies the amount of time to wait between power off and power on.

Return Value

Upon successful completion, the **reboot** subroutine does not return a value. If the **reboot** subroutine fails, a value of –1 is returned and the global variable **errno** is set to indicate the error.

Error Codes

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

EPERM	The calling process does not have root user authority.
EINVAL	The *HowTo* argument is not valid.
EFAULT	The *Argument* argument is not a valid address.

Implementation Specifics

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

Related Information

The **halt** command, **reboot** command, **shutdown** command.

regcmp or regex Subroutine

Purpose

Compiles and matches regular-expression patterns.

Library

Programmers Workbench Library (**libpw.a**)

Syntax

char *regcmp (*String* [, *String*, . . .], **(char *) 0)**
**char ****String*, **String*, . . . ;
char *regex (*Pattern*, *Subject* [, *ret*, . . .])
**char ****Pattern*, **Subject*, **ret*, . . . ;
extern char *_loc1;

Description

The **regcmp** subroutine compiles a regular expression (or *Pattern*) and returns a pointer to the compiled form. If more than one *String* parameter is given, then **regcmp** treats them as if they were concatenated together. It returns a **NULL** pointer if it encounters an incorrect parameter.

You can use the **regcmp** command to compile regular expressions into your C program, frequently eliminating the need to call the **regcmp** subroutine at run time.

The **regex** subroutine compares a compiled *Pattern* to the *Subject* string. Additional parameters are used to receive values. Upon successful completion, the **regex** subroutine returns a pointer to the next unmatched character. If the **regex** subroutine fails, a **NULL** pointer is returned. A global character pointer, **_loc1**, points to where the match began.

The **regcmp** and **regex** subroutines are borrowed from the **ed** command; however, the syntax and semantics have been changed slightly. You can use the following symbols with the **regcmp** and **regex** subroutines:

[ ] * . ^ These symbols have the same meaning as they do in the **ed** command.

– The minus sign (or hyphen) within brackets used with **regex** means "through," according to the current collating sequence. For example, [a-z] can be equivalent to [abcd . . . xyz] or [aBbCc . . . xYyZz] or even [aàáâbc . . . xyz] . You can use the – by itself if the – is the last or first character. For example, the character class expression [ ] –] matches the] (right bracket) and – (minus) characters.

The **regcmp** subroutine does not use the current collating sequence, and the minus character in brackets controls only a direct ASCII sequence. For example, [a-z] always means [abc . . . xyz] and [A-Z] always means [ABC . . . XYZ] . If you need to control the specific characters in a range using **regcmp**, you must list them explicitly rather than using the minus in the character class expression.

$ Matches the end of the string. Use \n to match a new-line character.

+ A regular expression followed by + (plus sign) means one or more times. For example, [0–9] + is equivalent to [0–9] [0–9] *.

[*m*] [*m*,] [*m*, *u*]

Integer values enclosed in [ ] (braces) indicate the number of times to apply the preceding regular expression. *m* is the minimum number and *u* is the maximum number. *u* must be less than 256. If you specify only *m*, it indicates the exact number of times to apply the regular expression. [*m*,] is equivalent to [*m*,*u*.] and matches *m* or more occurrences of the expression. The plus + (plus) and * (asterisk) operations are equivalent to [1,] and [0,] , respectively.

(. . .)$*n* This stores the value matched by the enclosed regular expression in the (*n*+1)th *ret* parameter. Ten enclosed regular expressions are allowed. **regex** makes the assignments unconditionally.

(. . .) Parentheses group subexpressions. An operator, such as *, +, or [ ].] works on a single character or on a regular expression enclosed in parentheses. For example, (a*(cb+)*)$0.

All of the preceding defined symbols are special. You must precede them with a \ (backslash) if you want to match the special symbol itself. For example, \$ matches a dollar sign.

Note: The **regcmp** subroutine uses the **malloc** subroutine to make the space for the vector. Always free the vectors that are not required. If you do not free the unrequired vectors, you can run out of memory if **regcmp** is called repeatedly. Use the following as a replacement for **malloc** to reuse the same vector, thus saving time and space:

```
/*   . . . Your Program . . .   */
malloc(n)
   int n;
[
   static int rebuf[256] ;
   return ((n <= sizeof(rebuf)) ? rebuf : NULL);
]
```

Using the Minus Symbol in Japanese Language Support

The [–] symbol (minus or hyphen within brackets) functions somewhat differently in Japanese Language Support.

The **regcmp** subroutine produces code values that the **regex** subroutine can interpret as the regular expression. For instance, [a-z] indicates a range expression which the **regcmp** subroutine compiles into a string containing the two end points (a and z).

The **regex** subroutine interprets the range statement according to the current collating sequence. The expression [a-z] can be equivalent either to [abcd . . . xyz] , or to [aBbCcDd . . . xXyYzZ] , as long as the character *preceding* the minus sign has a lower collating value than the character *following* the minus sign.

The behavior of a range expression is dependent on the collation sequence. If you want to match a *specific* set of characters, you should list each one. For example, to select letters a, b, or c, use [abc] rather than [a-c] .

Notes:

1. No assumptions are made at compile time about the actual characters contained in the range.
2. You can mix ASCII and SJIS characters in the expression.
3. You can use the] (right bracket) itself within a pair of brackets if it immediately follows the leading [(left bracket) or [^ (a left bracket followed immediately by a circumflex).
4. You can also use the minus sign (or hyphen) if it is the first or last character in the expression. For example, the expression [ ] —0] matches either the right bracket ( ]), or the characters – through 0.

Matching a Character Class in Japanese Language Support

A common use of the range expression is matching a character class. For example, [0-9] represents all digits, and [a-z, A-Z] represents all letters. This form may produce unexpected results when ranges are interpreted according to the current collating sequence.

Instead of the range expression shown above, use a character class expression within brackets to match characters. The system interprets this type of expression according to the current character class definition. However, you cannot use character class expressions in range expressions.

The following exemplifies the syntax of a character class expression:

`[:`*`charclass`*`:]`

a left bracket, followed by a colon, followed by the name of the character class, followed by another colon and a right bracket.

Japanese Language Support supports the following character classes:

[:upper:]	ASCII uppercase letters.
[:lower:]	ASCII lowercase letters.
[:alpha:]	ASCII uppercase and lowercase letters.
[:digit:]	ASCII digits.
[:alnum:]	ASCII uppercase and lowercase letters, and digits.
[:xdigit:]	ASCII hexadecimal digits.
[:punct:]	ASCII punctuation character (neither a control character nor an alphanumeric character).
[:space:]	ASCII space, tab, carriage return, new-line, vertical tab, or form feed character.
[:print:]	ASCII printing characters.
[:jalpha:]	SJIS Roman characters.
[:jdigit:]	SJIS Arabic digits.
[:jxdigit:]	SJIS hexadecimal digits.

[:jparen:] SJIS bracketing characters.

[:jpunct:] SJIS punctuation characters.

[:jspace:] SJIS space, tab, carriage return, new-line, vertical tab, or form feed characters.

[:jprint:] SJIS printing characters.

[:jkanji:] Kanji characters.

[:jhira:] Full-width hiragana characters.

[:jkana:] Half-width and full-width katakana characters.

The brackets are part of the character class definition. To match any uppercase ASCII letter or ASCII digit, use the following regular expression:

```
[ [:upper:]  [:digit:] ]
```

Do not use the expression [A-Z0-9] .

Parameters

Subject Specifies a comparison string.

String Specifies the *Pattern* to be compiled.

Pattern Specifies the expression to be compared.

ret Points to an address at which to store comparison data.

Implementation Specifics

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

Related Information

The **ctype** subroutine, **NCcollate, NCcoluniq, NCeqvmap, _NLXcol** subroutine, **rcompile, step, advance** subroutine, **malloc, free, realloc, calloc, mallopt, mallinfo, alloca** subroutine.

The **ed** command, **regcmp** command.

reltimerid Subroutine

Purpose

Releases a previously allocated interval timer.

Library

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

Syntax

#include <sys/time.h>
#include <sys/events.h>

int reltimerid(*Timerid***)**
timer_t *Timerid***;**

Description

The **reltimerid** subroutine is used to release a previously allocated interval timer, which is returned by the **gettimerid** subroutine. Any pending timer event generated by this interval timer is cancelled when the call returns.

Parameters

Timerid	The id of the interval timer being released.

Return Values

The **reltimerid** subroutine returns a 0 if it is successful. If an error occurs, the value −1 is returned and **errno** is set.

Error Codes

If the **reltimerid** subroutine fails a −1 is returned and **errno** is set with the following error code:

EINVAL	The timer ID specified by the *Timerid* parameter is not a valid timer ID.

Implementation Specifics

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

Related Information

The **gettimerid** subroutine.

remove Subroutine

Purpose

Removes a file.

Library

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

Syntax

#include <stdio.h>

int remove(*FileName***)**
**char ****FileName*;

Description

The **remove** subroutine causes a file whose name is the string pointed to by *FileName* to be no longer accessible by that name. A subsequent attempt to open that file using that name will fail, unless it is created anew. If the file is open, the operation will fail.

If the file operated upon by the **remove** subroutine has multiple links, the link count in the other files is decremented.

Parameter

FileName	Specifies the file name.

Return Values

Upon successful completion, the **remove** subroutine returns a value of 0; otherwise it returns a non–0 value.

Implementation Specifics

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

Related Information

The **link** subroutine, **rename** subroutine.

The **link, unlink** commands.

rename Subroutine

Purpose

Renames a directory or a file within a file system.

Library

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

Syntax

int rename (*FromPath*, *ToPath*)
char **FromPath*, **ToPath*;

Description

The **rename** subroutine renames a directory or a file within a file system.

For **rename** to complete successfully, the calling process must have write and search permission to the parent directories of both *FromPath* and *ToPath*. If *FromPath* is a directory and the parent directories of *FromPath* and *ToPath* are different, then the calling process must have write and search permission to *FromPath* as well.

If *FromPath* and *ToPath* both refer to the same existing file, the **rename** subroutine returns successfully and perform no other action.

Both *FromPath* and *ToPath* must be of the same type (that is, both directories or both non-directories) and must reside on the same file system. If *ToPath* already exists, it is first removed. In this case it is guaranteed that a link named *ToPath* will exist throughout the operation. This link refers to the file named by either *ToPath* or *FromPath* before the operation began.

If the final component of *FromPath* is a symbolic link, the symbolic link (not the file or directory to which it points) is renamed.

If the parent directory of the *FromPath* parameter has the *Sticky* attribute (described in the **sys/mode.h** header file), the calling process must have an effective user ID equal to the owner ID of the *FromPath* parameter, or to the owner ID of the parent directory of *FromPath*.

For a user who is not the owner of the file or directory to perform the **rename**, the user must have root user authority.

If the *FromPath* and *ToPath* parameters name directories, the following must be true:

- *FromPath* is not an ancestor of *ToPath*. For example, the *FromPath* pathname must not contain a path prefix that names *ToPath*.
- *FromPath* is well-formed; for example, the . entry in *FromPath*, if it exists, refers to the same directory as *FromPath*, exactly one directory has a link to *FromPath* (excluding the self-referential .), and the .. entry in *FromPath*, if it exists, refers to the directory that contains an entry for *FromPath*.
- *ToPath*, if it exists, must be well-formed (as defined previously).

Parameters

FromPath Identifies the file or directory to be renamed.

ToPath Identifies the new pathname of the file or directory to be renamed. If *T* is an existing file or empty directory, it is replaced by *FromPath*. If *ToPath* is not an empty directory, the **rename** subroutine exits with an error.

Return Values

Upon successful completion, the **rename** 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 **rename** subroutine fails and the file or directory name remains unchanged if one or more of the following are true:

ENOTDIR *FromPath* names a directory and *ToPath* names a non-directory.

EISDIR The *ToPath* parameter names a directory and the *FromPath* parameter names a nondirectory.

ENOENT A component of either path does not exist or the file named by *FromPath* does not exist.

EACCES Creating the requested link requires writing in a directory with a mode that denies write permission.

EXDEV The link named by *ToPath* and the file named by *FromPath* are on different file systems.

EBUSY The directory named by the *FromPath* or *ToPath* parameter is currently in use by the system.

EINVAL Either *FromPath* or the *ToPath* is not a well-formed directory.

EINVAL An attempt is made to rename . or .. .

EINVAL *FromPath* is an ancestor of *ToPath*.

EROFS The requested operation requires writing in a directory on a read-only file system.

EEXIST The *ToPath* parameter is an existing non-empty directory.

ENOSPC The directory that would contain *ToPath* cannot be extended because the file system is out of space.

EDQUOT The directory that would contain *ToPath* cannot be extended because the user's quota of disk blocks on the file system containing the directory is exhausted.

The **rename** subroutine can also fail if additional errors on page A–1 occur.

If Network File System is installed on the system, the **rename** system call can also fail if the following is true:

ETIMEDOUT The connection timed out.

Implementation Specifics

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

Related Information

The **chmod** subroutine, **link** subroutine, **mkdir** subroutine, **rmdir** subroutine, **unlink** subroutine.

The **chmod** command, **mkdir** command, **mv** command, **mvdir** command.

revoke Subroutine

Purpose

Revokes access to a file.

Library

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

Syntax

int revoke(*Path***)**
char ***Path*;**

Description

The **revoke** subroutine revokes access to a file by all processes.

All accesses to the file are revoked. Subsequent attempts to access the file using a file descriptor established before the **revoke** subroutine fail and cause the process to be killed.

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 calling process is privileged.

Note: The **revoke** subroutine has no affect on subsequent attempts to open the file. To assure exclusive access to the file, the caller should change the mode of the file before issuing the **revoke** subroutine. Currently the **revoke** subroutine works only on terminal devices.

Parameter

Path Path name of the file for which access is to be revoked.

Return Values

Upon successful completion, the **revoke** subroutine returns a value of 0.

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

Error Codes

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

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

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

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

ENOENT The path name is null.

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

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

EFAULT The *Path* parameter points outside of the process's address space.

ELOOP Too many symbolic links were encountered in translating the path name.

ENAMETOOLONG
A component of a path name exceeds five characters, or an entire path name exceeds 1023 characters.

EIO An I/O error occurred during the operation.

EPERM The effective user ID of the calling process is not the same as the file's owner ID.

EINVAL Access rights revocation is not implemented for this file.

Implementation Specifics

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

Related Information

The **frevoke** subroutine.

rmdir Subroutine

Purpose

Removes a directory file.

Library

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

Syntax

int rmdir (*Path***)**
char ****Path*;**

Description

The **rmdir** subroutine removes the directory specified by the *Path* parameter. If Network File System is installed on your system, this path can cross into another node.

For the **rmdir** subroutine to execute successfully, the calling process must have write access to the parent directory of the *Path* parameter.

In addition, if the parent directory of *Path* has the *Sticky* attribute (described in the **sys/mode.h** header file), the calling process must have an effective user ID equal to the directory to be removed, or have an effective user ID equal to the owner ID of the parent directory of *Path*, or have root user authority.

Parameter

Path Specifies the directory path name. The directory you specify must be:

Empty	The directory contains no entries other than . and .. .
Well-formed	If the . entry in the *Path* parameter exists, it must refer to the same directory as *Path*. Exactly one directory has a link to the *Path* parameter (excluding the self-referential .). If the .. entry in *Path* exists, it must refer to the directory that contains an entry for *Path*.

Return Values

Upon successful completion, the **rmdir** 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 **rmdir** subroutine fails and the directory is not deleted if the following errors occur:

EBUSY	The directory is in use as a mount point.
EEXIST	The directory named by the *Path* parameter is not empty.
ENOENT	The directory named by the *Path* parameter does not exist.
EINVAL	The directory named by the *Path* parameter is not well formed.
EROFS	The directory named by the *Path* parameter resides on a read-only file system.

The **rmdir** subroutine can also fail if additional errors on page A–1 occur.

If Network File System is installed on the system, the **rmdir** subroutine can also fail if the following is true:

ETIMEDOUT	The connection timed out.

Implementation Specifics

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

Related Information

The **chmod**, **fchmod** subroutines, **mkdir** subroutine, **rename** subroutine, **umask** subroutine.

The **rm** command, **rmdir** command.

scandir or alphasort Subroutine

Purpose

Scans or sorts directory contents.

Library

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

Syntax

#include <sys/types.h>
#include <sys/dir.h>

int scandir (*DirectoryName*, *NameList*, *Select*, *Compare***)**
char ****DirectoryName*;**
struct dirent * (*[*NameList* **]);**
int (**Select***) ( );**
int (**Compare***) ( );**

int alphasort (*Directory1*, *Directory2***)**
struct dirent *****Directory1*, *****Directory2***;**

Description

The **scandir** subroutine reads the directory pointed to by the *DirectoryName* parameter, and then uses the **malloc** subroutine to create an array of pointers to directory entries. The **scandir** subroutine returns the number of entries in the array and, through the *NameList* parameter, a pointer to the array.

The *Select* parameter points to a user–supplied subroutine that is called by the **scandir** subroutine to select which entries to include in the array. The selection routine is passed a pointer to a directory entry and should return a nonzero value for a directory entry that is included in the array. If the *Select* parameter is a **NULL** value, all directory entries are included.

The *Compare* parameter points to a user–supplied subroutine. This routine is passed to the **qsort** subroutine to sort the completed array. If the *Compare* parameter is a **NULL** value, the array is not sorted. The **alphasort** subroutine provides comparison functions for sorting alphabetically.

The memory allocated to the array can be de–allocated by freeing each pointer in the array, and the array itself, with the **free** subroutine.

The **alphasort** subroutine alphabetically compares the two **dirent** structures pointed to the the *Directory1* and *Directory2* parameters. This subroutine can be passed as the *Compare* parameter to either the **scandir** subroutine or the **qsort** subroutine, or a user–supplied subroutine can be used.

Parameters

DirectoryName	Points to the directory name.
NameList	Points to the array of pointers to directory entries.
Select	Points to a user–supplied subroutine that is called by the **scandir** subroutine to select which entries to include in the array.

Compare	Points to a user-supplied subroutine that sorts the completed array.
Directory1, Directory2	Point to **dirent** structures.

Return Values

The **scandir** subroutine returns the value –1 if the directory cannot be opened for reading or if the **malloc** subroutine cannot allocate enough memory to hold all the data structures. If successful, the **scandir** subroutine returns the number of entries found.

The **alphasort** subroutine returns the following values:

Less than 0	The **dirent** structure pointed to by the *Directory1* parameter is lexically less than the **dirent** structure pointed to by the *Directory2* parameter.
0	The **dirent** structures pointed to by the *Directory1* parameter and the *Directory2* parameter are equal.
Greater than 0	The **dirent** structure pointed to by the *Directory1* parameter is lexically greater than the **dirent** structure pointed to by the *Directory2* parameter.

Implementation Specifics

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

Related Information

The **opendir**, **readdir**, **telldir**, **seekdir**, **rewinddir**, **closedir** subroutines, **malloc** subroutine, **free** subroutine, **qsort** subroutine.

scanf, fscanf, sscanf, NLscanf, NLfscanf, or NLsscanf Subroutine

Purpose

Converts formatted input.

Library

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

Syntax

#include <stdio.h>

int scanf (*Format* [, *Pointer*, ...])
char **Format*;

int fscanf (*Stream*,*Format* [, *Pointer*, ...])
FILE **Stream*;
char **Format*;

int sscanf (*String*, *Format* [, *Pointer*, ...])
char **String*, **Format*;

int NLscanf (*Format* [, *Pointer*, ...])
char **Format*;

int NLfscanf (*Stream*, *Format* [, *Pointer*, ...])
FILE **Stream*;
char **Format*;

int NLsscanf (*String*, *Format* [, *Pointer*, ...])
char **String*, **Format*;

Description

The **scanf** subroutine, **fscanf** subroutine, and **sscanf** subroutine read character data, interpret it according to a format, and store the converted results into specified memory locations. If there are insufficient arguments for the format, the behavior is undefined. If the format is exhausted while arguments remain, the excess arguments are evaluated but otherwise ignored.

These subroutines read their input from the following sources:

scanf, NLscanf	Read from standard input (stdin).
fscanf, NLfscanf	Read from the *Stream* parameter.
sscanf, NLsscanf	Read from the character string specified by the *String* parameter.

Parameters

Stream	Specifies the input stream
String	Specifies input to be read.
Pointer	Specifies where to store the interpreted data.

Format Contains conversion specifications used to interpret the input. If there are insufficient arguments for the *Format*, the behavior is undefined. If the *Format* is exhausted while arguments remain, the excess arguments are evaluated as always but are otherwise ignored.

The *Format* parameter can contain the following:

- White space characters (blanks, tabs, new–line, or form feed) that, except in the following two cases, read the input up to the next nonwhite space character. Unless there is a match in the control string, trailing white space (including a new–line character) is not read.
- Any character except % (percent), which must match the next character of the input stream.
- A conversion specification that directs the conversion of the next input field. It consists of the following:
  1. The character % (percent)
  2. The optional assignment suppression character * (asterisk)
  3. An optional numeric maximum field width
  4. An optional character that sets the size of the receiving variable as for some flags, as follows:

     l Signed long integer rather than an int when preceding the **d**, **u**, **o**, or **x** conversion codes.

     L A double rather than a float, when preceding the **e**, **f**, or **g** conversion codes.

     h Signed short integer (half int) rather than an int when preceding the **d**, **u**, **o**, or **x** conversion codes.
  5. A conversion code.

     The conversion specification takes the form:

     ```
     %[*][width][size]convcode
     ```

     The results from the conversion are placed in **Pointer* unless you specify assignment suppression with *. Assignment suppression provides a way to describe an input field that is to be skipped. The input field is a string of nonwhite space characters. It extends to the next inappropriate character or until the field width, if specified, is exhausted.

     The conversion code indicates how to interpret the input field. The corresponding *Pointer* must usually be of a restricted type. You should not specify the *Pointer* parameter for a suppressed field. You can use the following conversion codes:

     % Accepts a single % input at this point; no assignment is done.

     d, i Accepts a decimal integer; the *Pointer* pointer should be an integer pointer.

     u Accepts an unsigned decimal integer; the *Pointer* parameter should be an unsigned integer pointer.

     o Accepts an octal integer; the *Pointer* parameter should be an integer pointer.

x Accepts a hexadecimal integer; the *Pointer* parameter should be an integer pointer.

e, f, g Accepts a floating–point number. The next field is converted accordingly and stored through the corresponding parameter, which should be a pointer to a float. The input format for floating–point numbers is a string of digits, with some optional characteristics:

- It can be a signed value.
- It can be an exponential value, containing a decimal point followed by an exponent field, which consists of an **E** or an **e** followed by an (optionally signed) integer.
- It can be one of the special values **INF**, **NaNQ**, or **NaNS**. This value is translated into the ANSI/IEEE value for infinity, quiet NaN, or signaling NaN, respectively.

For Japanese Language Support, the conversion codes recognize double–width versions of digits as equivalent to the single–width versions of those digits.

p Matches an unsigned hexadecimal integer, the same as the **&p** conversion of the **printf** subroutine. The corresponding argument is a pointer to a pointer to **void**.

n No input is consumed. The corresponding argument is a pointer to an integer into which is written the number of characters read from the input stream so far by this function. The assignment count returned at the completion of this function is not incremented.

s Accepts a string of **char**s. The *Pointer* parameter should be a character pointer that points to an array of characters large enough to accept the string and ending with \0. The \0 is added automatically. The input field ends with a white space character. A string of **char** values is output.

S Accepts an **NLchar** string. The *Pointer* parameter points to an array of characters large enough to accept the string and ending with \0. The \0 is added automatically. The input field ends with a white space character. A string of **NLchar** values is output.

ls Accepts an **NLchar** string. The *Pointer* parameter points to an array of characters large enough to accept the string and ending with \0. The \0 is added automatically. The input field ends with a white space character. A string of **NLchar** values is output.

ws Accepts an **NLchar** string. The *Pointer* parameter points to an array of characters large enough to accept the string and ending with \0. The \0 is added automatically. The input field ends with a white space character. A string of **NLchar** values is output.

N Accepts an ASCII string, possibly containing extended character information in the form of escape sequences used by the **NLescstr** and **NLunescstr** subroutines. The output is in the form of **NLchars**.

B Returns the length of the string in bytes, rather than the display length of the string.

c A **char** value is expected. The *Pointer* parameter should be a **char** pointer. The normal skip over white space is suppressed. Use **%s** to read the next nonwhite space character. If a field width is given, *Pointer* refers to a character array, and the indicated number of **char** values is read.

C Accepts and prints an **NLchar** *Value.*

lc Accepts and prints an **NLchar** *Value.*

wc Accepts and prints an **NLchar** *Value.*

[scanset] Accepts as input the characters included in the *scanset.* The *scanset* explicitly defines the characters that are accepted in the string data as those enclosed within square brackets. The normal skip over leading white space is suppressed. In*scanset* in the form of [^*scanset*] of an *exclusive scanset,* the ^ (circumflex) serves as a complement operator and the following characters in the *scanset* are not accepted as input. Conventions used in the construction of the *scanset* follow:

- You can represent a range of characters by the construct *First–Last.* Thus, you can express [0123456789] as [0–9]. The *First* parameter must be lexically less than or equal to *Last,* or else the – (hyphen) stands for itself. The – also stands for itself whenever it is the first or the last character in the *scanset.*
- You can include the] (right bracket) as an element of the *scanset* if it is the first character of the *scanset.* In this case it is not interpreted as the bracket that closes the *scanset.* If the *scanset* is an exclusive *scanset,* the] is preceded by the ^ (circumflex) to make the] an element of the *scanset.* The corresponding *Pointer* parameter must point to a character array large enough to hold the data field and that ends with '\0'. The '\0' is added automatically.

A **scanf** or **NLscanf** conversion ends at the end of the file, the end of the control string, or when an input character conflicts with the control string. If it ends with an input character conflict, the conflicting character is not read from the input stream.

Unless there is a match in the control string, trailing white space (including a new–line character) is not read.

The success of literal matches and suppressed assignments is not directly determinable.

The **NLS** extensions to the **scanf** subroutines can handle a format string that enables the system to process elements of the argument list in variable order. The normal conversion character % is replaced by *%digit$,* where *digit* is a decimal number. Conversions are then applied to arguments in the list with ordinal digits, rather than to the next unused argument.

The following restrictions apply:

- The format passed to the **NLS** extensions can contain one of the following forms, but not both:
  - The format of the conversion
  - The explicit or implicit argument number.

These forms cannot be mixed within a single format string.

- The * (asterisk) specification for field width or precision is not permitted with the variable order *%digit$* format.

Return Values

Each of these subroutines returns the *display length* of the string it outputs, which is the number of the display characters in the string, rather than the number of bytes. These subroutines return **EOF** on the end of input and on a short count for missing or invalid data items.

The **scanf** and **NLscanf** subroutines return the number of successfully matched and assigned input items. This number can be 0 is there was an early conflict between an input character and the control string. If the input ends before the first conflict or conversion, only **EOF** is returned.

Implementation Specifics

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

Related Information

The **atof**, **atoff**, **strtod**, **strtof** subroutines, **getc**, **getchar**, **getw**, **getwc**, **fgetwc**, **getwchar** subroutines, **printf**, **fprintf**, **sprintf**, **NLprintf**, **NLfprintf**, **NLsprintf** subroutines, **strtol**, **strtoul**, **atol**, **atoi** subroutines, **wsscanf** subroutine.

National Language Support Overview in *General Programming Concepts*

select Subroutine

Purpose

Checks the I/O status of multiple file descriptors and message queues.

Library

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

Syntax

#include <sys/select.h>
#include <sys/types.h>

int select (*Nfdsmsgs, ReadList, WriteList, ExceptList, TimeOut***)**
int *Nfdsmsgs*;
struct sellist **ReadList*, **WriteList*, **ExceptList*;
struct timeval **TimeOut*;

Description

The **select** subroutine checks the specified file descriptors and message queues to see if they are ready for reading (receiving) or writing (sending), or if they have an exceptional condition pending.

Parameters

Nfdsmsgs Specifies the number of file descriptors and the number of message queues to check. The low–order 16 bits give the length of a bit mask that specifies which file descriptors to check; the high–order 16 bits give the size of an array that contains message queue identifiers. If either half of the *Nfdsmsgs* parameter is equal to a value of 0, the corresponding bit mask or array is assumed not to be present.

ReadList, WriteList, ExceptList
Specify what to check for reading, writing, and exceptions, respectively. Together, they specify the selection criteria. Each of these parameters points to a **sellist** structure, which can specify both file descriptors and message queues. Your program must define the **sellist** structure in the following form:

```
struct sellist
{
int fdsmask[F]; /* file descriptor bit mask  */
int msgids[M];  /* message queue identifiers */
};
```

The **fdsmask** array is treated as a bit string in which each bit corresponds to a file descriptor. File descriptor *n* is represented by the bit **(1 <<** *n***)** in the array element **fdsmask[***n* **/ BITS(int)]**. (The **BITS** macro is defined in the **values.h** header file.) Each bit that is set to 1 indicates that the status of the corresponding file descriptor is to be checked. Note that the low–order 16 bits of the *Nfdsmsgs* parameter specify the number of *bits* (not elements) in the **fdsmask** array that make up the file descriptor mask. If only part of the last int is included in the mask, the appropriate number of low–order bits are used, and the remaining high–order bits are ignored.

If you set the low–order 16 bits of the *Nfdsmsgs* parameter to 0, you must *not* define an **fdsmask** array in the **sellist** structure.

Each int of the **msgids** array specifies a message queue identifier whose status is to be checked. Elements with a value of –1 are ignored. The high–order 16 bits of the *Nfdsmsgs* parameter specify the number of elements in the **msgids** array. If you set the high–order 16 bits of the *Nfdsmsgs* parameter to 0, you must *not* define a **msgids** array in the **sellist** structure.

Note: The arrays specified by the *ReadList*, *WriteList*, and *ExceptList* parameters are the same size because each of these parameters points to the same **sellist** structure type. However, you need not specify the same number of file descriptors or message queues in each. Set the file descriptor bits that are not of interest to 0, and set the extra elements of the **msgids** array to –1.

You can use the **SELLIST** macro defined in the **sys/select.h** header file to define the **sellist** structure. The format of this macro is:

SELLIST(*f*, *m***)** *declarator* . . . ;

where *f* specifies the size of the **fdsmask** array, *m* specifies the size of the **msgids** array, and each *declarator* is the name of a variable to be declared as having this type.

TimeOut Specifies either a **NULL** pointer or a pointer to a **timeval** structure that specifies the maximum length of time to wait for at least one of the selection criteria to be met. The **timeval** structure is defined in the **/sys/time.h** header file and it contains the following members:

```
struct timeval {
   int tv_sec;  /* seconds       */
   int tv_usec; /* microseconds  */
    };
```

The number of microseconds specified in *TimeOut*.tv_usec, a value from 0 to 999999, is set to one millisecond by the AIX Version 3 Operating System if the process does not have root user authority and the value is less than one millisecond.

If the *TimeOut* parameter is a **NULL** pointer, the **select** subroutine waits indefinitely, until at least one of the selection criteria is met. If the *TimeOut* parameter points to a **timeval** structure that contains zeros, the file and message queue status is polled, and the **select** subroutine returns immediately.

Return Values

Upon successful completion, the **select** subroutine returns a value that indicates the total number of file descriptors and message queues that satisfy the selection criteria. The **fdsmask** bit masks are modified so that bits set to 1 indicate file descriptors that meet the criteria. The **msgids** arrays are altered so that message queue identifiers that do not meet the criteria are replaced with a value of –1.

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. These values indicate the sum total that meet each of the read, write, and exception criteria. Therefore, the same file descriptor or message queue can be counted up to three times. You can use the **NFDS** and **NMSGS** macros found in the **/sys/select.h** header file to separate out these two values from the return value. For example, if *rc* contains the value returned from the **select** subroutine, **NFDS**(*rc*) is the number of files selected, and **NMSGS**(*rc*) is the number of message queues selected.

If the time limit specified by the *TimeOut* parameter expires, the **select** subroutine returns a value of 0.

If the **select** subroutine fails, it returns a value of –1 and sets the global variable **errno** to indicate the error. In this case, the contents of the structures pointed to by the *ReadList*, *WriteList*, and *ExceptList* parameters are unpredictable.

Error Codes

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

EBADF	An invalid file descriptor or message queue identifier was specified.
EAGAIN	Allocation of internal data structures failed.
EINTR	A signal was caught during the **select** subroutine and the signal handler was installed with an indication that subroutines are not to be restarted.
EINVAL	One of the parameters to the **select** subroutine contained an invalid value.
EFAULT	The *ReadList*, *WriteList*, *ExceptList*, or *TimeOut* parameter points to a location outside of the address space of the process.

Implementation Specifics

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

The **select** subroutine is supported for compatibility with previous releases of the AIX Operating System and BSD systems.

Related Information

The **poll** subroutine.

semctl Subroutine

Purpose

Controls semaphore operations.

Library

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

Syntax

#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/sem.h>

int semctl (*SemaphoreID,SemaphoreNumber,Command, Value*)
or
int semctl (*SemaphoreID,SemaphoreNumber,Command,Buffer*)
or
int semctl (*SemaphoreID,SemaphoreNumber,Command,Array*)

int *SemaphoreID*;
int *SemaphoreNumber*;
int *Command*;
int *Value*;
struct semid_ds * *Buffer*;
unsigned short *Array*[ ];

Description

The **semctl** subroutine performs a variety of semaphore control operations as specified by the *Command* parameter. The data type of the last parameter depends on the value of the *Command* parameter. It is referred to as the *Value*, *Buffer*, or *Array* parameter to indicate one of the definitions given in the preceding syntax section.

Parameters

SemaphoreID	Specifies the semaphore identifier.
SemaphoreNumber	Specifies the semaphore number.
Value	Specifies the data type of the *Command* parameter.
Buffer	Specifies the data type of the *Command* parameter.
Array	Specifies the data type of the *Command* parameter.

Command

Specifies semaphore control operations.The first seven values of the *Command* parameter get and set the values of a **sem** structure, which is defined in the **sys/sem.h** header file.

The following *Command* parameter values are executed with respect to the semaphore specified by the *SemaphoreID* and *SemaphoreNumber* parameters.

GETVAL
Returns the value of semval, if the current process has read permission.

SETVAL
Sets the value of semval to the value specified by the *Value* parameter, if the current process has write permission. When this *Command* parameter is successfully executed, the semadj value corresponding to the specified semaphore is cleared in all processes.

GETPID
Returns the value of sempid, if the current process has read permission.

GETNCNT
Returns the value of semncnt, if the current process has read permission.

GETZCNT
Returns the value of semzcnt, if the current process has read permission.

The following *Command* parameter values return and set every semval in the set of semaphores.

GETALL
Stores semvals into the array pointed to by the *Array* parameter, if the current process has read permission.

SETALL
Sets semvals according to the array pointed to by the *Array* parameter, if the current process has write permission. When this *Command* is successfully executed, the semadj value corresponding to each specified semaphore is cleared in all processes.

The following *Commands* get and set the values of a **semid_ds** structure, defined in the **sys/sem.h** header file.

IPC_STAT
This command obtains status information about the semaphore identified by the *SemaphoreID* parameter. This information is stored in the area pointed to by the *Buffer* parameter.

IPC_SET
These two commands set the owning user and group IDs, and the access permissions for the set of semaphores associated with the *SemaphoreID* parameter. The **IPC_SET** command uses as input the values found in the *Buffer* parameter structure.

IPC_SET sets the following fields:

sem_perm.uid	Owning user ID
sem_perm.gid	Owning group ID
sem_perm.mode	Permission bits only

IPC_SET can only be executed by a process that has root user authority or an effective user ID equal to the value of the sem_perm.uid field in the data structure associated with the *SemaphoreID* parameter.

IPC_RMID Removes the semaphore identifier specified by the *SemaphoreID* parameter from the system and destroys the set of semaphores and data structures associated with it. This *Command* can only be executed by a process that has root user authority or an effective user ID equal to the value of sem_perm.uid in the data structure associated with the *SemaphoreID* parameter.

Return Values

Upon successful completion, the value returned depends on the *Command* parameter as follows:

Command	**Return Value**
GETVAL	Returns the value of semval.
GETPID	Returns the value of sempid.
GETNCNT	Returns the value of semncnt.
GETZCNT	Returns the value of semzcnt.
all others	Return a value of 0.

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

Error Codes

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

EINVAL	The *SemaphoreID* parameter is not a valid semaphore identifier.
EINVAL	The *SemaphoreNumber* parameter is less than 0 or greater than sem_nsems.
EINVAL	The *Command* parameter is not a valid command.
EACCES	Operation permission is denied to the calling process.
ERANGE	The *Command* parameter is **SETVAL** or **SETALL** and the value to which semval is to be set is greater than the system–imposed maximum.

EPERM The *Command* parameter is equal to **IPC_RMID** or **IPC_SET** and the calling process does not have root user authority or an effective user ID equal to the value of the sem_perm.uid field in the data structure associated with the *SemaphoreID* parameter.

EFAULT The *Buffer* or *Array* parameter points outside of the allocated address space of the process.

ENOMEM The system does not have enough memory to complete the subroutine.

Implementation Specifics

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

Related Information

The **semget** subroutine, **semop** subroutine.

semget Subroutine

Purpose

Gets a set of semaphores.

Library

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

Syntax

#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/sem.h>

int semget (*Key*, *NumberOfSemaphores*, *SemaphoreFlag*)
key_t *Key*;
int *NumberOfSemaphores*, *SemaphoreFlag*;

Description

The **semget** subroutine returns the semaphore identifier associated with the specified *Key*.

The **semget** subroutine creates a data structure for the semaphore ID and an array containing the *NumberOfSemaphores* parameter semaphores if one of the following is true:

- The *Key* parameter is equal to **IPC_PRIVATE**.
- The *Key* parameter does not already have a semaphore identifier associated with it, and **IPC_CREAT** is set.

Upon creation, the data structure associated with the new semaphore identifier is initialized as follows:

- sem_perm.cuid and sem_perm.uid are set equal to the effective user ID of the calling process.
- sem_perm.cgid and sem_perm.gid are set equal to the effective group ID of the calling process.
- The low–order 9 bits of sem_perm.mode are set equal to the low–order 9 bits of the *SemaphoreFlag* parameter.
- sem_nsems is set equal to the value of the *NumberOfSemaphores* parameter.
- sem_otime is set equal to 0 and sem_ctime is set equal to the current time.

If the *Key* parameter is not **IPC_PRIVATE**, **IPC_EXCL** is not set, and a semaphore identifier already exists for the specified *Key* parameter, the value of the *NumberOfSemaphores* parameter specifies the number of semaphores that the current process needs. If the *NumberOfSemaphores* parameter is a value of 0, any number of semaphores is acceptable. If the *NumberOfSemaphores* parameter is not a value of 0, the **semget** subroutine fails if the set contains fewer than *NumberOfSemaphores* semaphores.

Parameters

Key Specifies either the value **IPC_PRIVATE** or an IPC key constructed by the **ftok** subroutine (or by a similar algorithm).

NumberOfSemaphores
Specifies the number of semaphores in the set.

SemaphoreFlag 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 **semget** subroutine to fail if **IPC_CREAT** 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.

The values that begin with the **S_I** prefix are defined in the **sys/mode.h** header file and are a subset of the access permissions that apply to files.

Return Values

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

Error Codes

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

EINVAL The *NumberOfSemaphores* parameter is less than a value of 0, equal to a value of 0, or greater than the system–imposed limit.

EACCES A semaphore identifier exists for the *Key* parameter but operation permission, as specified by the low–order 9 bits of the *SemaphoreFlag* parameter, is not granted.

EINVAL A semaphore identifier exists for the *Key* parameter, but the number of semaphores in the set associated with it is less than the value of the *NumberOfSemaphores* parameter and the *NumberOfSemaphores* parameter is not equal to 0.

ENOENT A semaphore identifier does not exist for the *Key* parameter and **IPC_CREAT** is not set.

ENOSPC A semaphore identifier is to be created, but doing so would exceed the maximum number of identifiers allowed systemwide.

EEXIST A semaphore identifier exists for the *Key* parameter, and both **IPC_CREAT** and **IPC_EXCL** are set.

ENOMEM There is not enough memory to complete the operation.

Implementation Specifics

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

Related Information

The **semctl** subroutine, **semop** subroutine.

The **ftok** subroutine.

semop Subroutine

Purpose

Performs semaphore operations.

Library

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

Syntax

#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/sem.h>

int semop (*SemaphoreID, SemaphoreOperations, NumberOfSemaphoreOperations*)
int *SemaphoreID*;
struct sembuf **SemaphoreOperations*;
unsigned *NumberOfSemaphoreOperations*;

Description

The **semop** subroutine performs operations on the set of semaphores associated with the semaphore identifier specified by the *SemaphoreID* parameter. The **sembuf** structure is defined in the **sys/sem.h** header file.

Parameters

SemaphoreID	Specifies the semaphore identifier.
SemaphoreOperations	Points to an array of structures, each of which specifies a semaphore operation.
NumberOfSemaphoreOperations	Specifies the number of structures in the array.

Return Values

Upon successful completion, the **semop** subroutine returns a value of 0. Also, the *SemahoreID* parameter value for each semaphore that is operated upon is set to the process ID of the calling process.

If the **semop** subroutine fails, a value of –1 is returned and the global variable **errno** is set to indicate the error. If **SEM_ORDER** was set in the sem_flg for the first semaphore operation in the *SemaphoreOperations* array, the **SEM_ERR** value is set in the sem_flg for the failing operation.

Error Codes

The **semop** subroutine fails if one or more of the following are true for any of the semaphore operations specified by the *SemaphoreOperations* parameter. If the operations were performed individually, then see the preceding discussion of **SEM_ORDER** for more information about error situations.

EINVAL The *SemaphoreID* parameter is not a valid semaphore identifier.

EFBIG sem_num is less than 0 or it is greater than or equal to the number of semaphores in the set associated with the *SemaphoreID* parameter.

E2BIG The *NumberOfSemaphoreOperations* parameter is greater than the system–imposed maximum.

EACCES Operation permission is denied to the calling process.

EAGAIN The operation would result in suspension of the calling process, but **IPC_NOWAIT** is set in sem_flg.

ENOSPC The limit on the number of individual processes requesting a **SEM_UNDO** would be exceeded.

EINVAL The number of individual semaphores for which the calling process requests a **SEM_UNDO** would exceed the limit.

ERANGE An operation would cause a **semval** to overflow the system–imposed limit.

ERANGE An operation would cause a **semadj** value to overflow the system–imposed limit.

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

EINTR The **semop** subroutine received a signal.

EIDRM The semaphore identifier *SemaphoreID* parameter has been removed from the system.

Implementation Specifics

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

Related Information

The **exec** subroutine, **exit** subroutine, **fork** subroutine, **semctl** subroutine, **semget** subroutine.

setbuf, setvbuf, setbuffer, or setlinebuf Subroutine

Purpose

Assigns buffering to a stream.

Library

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

Syntax

#include <stdio.h>

void setbuf (*Stream*, *Buffer*)
FILE **Stream*;
char **Buffer*;

int setvbuf (*Stream*, *Buffer*, *Mode*, *Size*)
FILE **Stream*;
char **Buffer*;
int *Mode*;
size_t *Size*;

void setbuffer (*Stream*, *Buffer*, *Size*)
FILE **Stream*;
char **Buffer*;
size_t *Size*;

void setlinebuf (*Stream*)
FILE **Stream*;

Description

The **setbuf** subroutine causes the character array pointed to by the *Buffer* parameter to be used instead of an automatically allocated buffer. Use the **setbuf** subroutine after a stream has been opened, but before it is read or written.

If the *Buffer* parameter is a null character pointer, input/output is completely unbuffered.

A constant, **BUFSIZ**, defined in the **stdio.h** header file, tells how large an array is needed:

```
char buf[BUFSIZ];
```

For the **setvbuf** subroutine, the *Mode* parameter determines how the *Stream* parameter is buffered:

_IOFBF Causes input/output to be fully buffered.

_IOLBF Causes output to be line–buffered. The buffer is flushed when a new line is written, the buffer is full, or input is requested.

_IONBF Causes input/output to be completely unbuffered.

If the *Buffer* parameter is not a null character pointer, the array it points to is used for buffering instead of an automatically allocated buffer. The *Size* parameter specifies the size of the buffer to be used. The constant **BUFSIZ** in the **stdio.h** header file is one buffer size. If input/output is unbuffered, the *Buffer* and *Size* parameters are ignored. The **setbuffer** subroutine, an alternate form of the **setbuf** subroutine, is used after *Stream* has been opened, but before it is read or written. The character array *Buffer*, whose size is determined by the *Size* parameter, is used instead of an automatically allocated buffer. If the *Buffer* parameter is a null character pointer, input/output is completely unbuffered.

The **setbuffer** subroutine is not needed under normal circumstances since the default file I/O buffer size is optimal.

The **setlinebuf** subroutine is used to change STDOUT or STDERR from block buffered or unbuffered to line–buffered. Unlike the **setbuf** and **setbuffer** subroutines, the **setlinebuf** subroutine can be used any time the file descriptor is active.

A buffer is normally obtained from the **malloc** subroutine at the time of the first **getc** subroutine or **putc** subroutine on the file, except that the standard error stream, STDERR, is normally not buffered.

Output streams directed to terminals are always either line–buffered or unbuffered.

Note: A common source of error is allocating buffer space as an automatic variable in a code block, and then failing to close the stream in the same block.

Parameters

Stream	Specifies the input/output stream.
Buffer	Points to a character array.
Mode	Determines how the *Stream* parameter is buffered.
Size	Specifies the size of the buffer to be used.

Implementation Specifics

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

The **setbuffer** and **setlinebuf** subroutines are included for compatibility with BSD.

Related Information

The **fopen**, **freopen**, **fdopen** subroutines, **fread** subroutine, **getc**, **fgetc**, **getchar**, **getw**, **getwc**, **fgetwc**, **getwchar** subroutines, **malloc**, **free**, **realloc**, **calloc**, **malinfo**, **mallopt**, **alloca** subroutines, **putc**, **putchar**, **fputc**, **putw** subroutines, **putwc**, **putwchar**, **fputwc** subroutines.

setgid, setrgid, setegid or setregid Subroutine

Purpose

Sets the process group IDs.

Library

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

Syntax

#include <sys/types.h>

int setgid (*GID*)
gid_t *GID*;

int setrgid (*RGID*)
gid_t *RGID*;

int setegid (*EGID*)
gid_t *EGID*;

int setregid (*RGID*, **EGID**)
gid_t *RGID*;
gid_t *EGID*

Description

These subroutines set the group IDs of the calling process. The following semantics are supported:

setgid
If the invoker is the root user, the process's real, effective, and saved group IDs are set to the value of the *GID* parameter. Otherwise, the process's effective group ID is reset if *GID* is equal to either the current real or saved group IDs.

setegid
The process's effective group ID is reset if *EGID* is equal to either the current real or saved group IDs.

setrgid
EPERM is always returned.

setregid
There are two cases:

RGID != *EGID*
If *EGID* is equal to either the process's real or saved group IDs, the process's effective group ID is set to *EGID*;
else **EPERM** is returned.

RGID= = *EGID*
If the invoker is the root user, the process's real, effective, and saved group IDs are set to *EGID*; else if *EGID* is equal to the process's real or saved group IDs, the process's effective group ID is set to *EGID*;
else **EPERM** is returned.

These functions are provided as compatibility interfaces to **setgidx**; this subroutine should be called directly by all new programs. The current semantics of these functions is only supported insofar as they do not conflict with the ID setting policy of **setgidx**.

Parameters

GID	Specifies the value of the group ID to be set.
RGID	Specifies the value of the real group ID to be set.
EGID	Specifies the value of the effective group ID to be set.

Return Values

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

Error Codes

The **setgid** and **setegid** subroutines fail if:

EPERM	The *GID* or *EGID* parameter is not equal to either the real or saved group IDs of the process.

Implementation Specifics

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

Related Information

The **getgroups** subroutine, **getuidx** subroutine, **setgidx** subroutine, **setgroups** subroutine, **setuidx** subroutine.

The **setgroups** command.

setgidx Subroutine

Purpose

Sets the process group IDs.

Library

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

Syntax

#include <sys/id.h>

uid_t setgidx (*Which*, *GID*)
int *Which*;
gid_t *GID*

Description

The **setgidx** subroutine sets the specified group IDs of the current process.

Parameters

Which — Specifies which group ID to return. This parameter is a bitmask and may include one or more of the following, which are defined in **sys/id.h**:

- **ID_EFFECTIVE** — Sets the effective group ID of the process.
- **ID_REAL** — Sets the real group ID of the process.
- **ID_SAVED** — Sets the saved group ID of the process.

GID — Specifies the value of the group ID to be set.

Return Values

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

Error Codes

The **setgidx** subroutine fails if:

EINVAL — The *Which* parameter does not contain a valid combination of IDs to be changed.

Security

DAC Policy — The following policies are enforced:

The real and saved IDs can only be changed by the root user.

If the real ID is changed, the effective ID must be changed to the same value (i.e. *Which* = **ID_EFFECTIVE/ID_REAL**) (i.e. *Which* = **ID_EFFECTIVE/ID_REAL/ID_SAVED**). If the saved ID is changed, the real and effective ID must be changed to the same values.

The effective ID may be changed only to the current real or saved IDs.

Implementation Specifics

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

Related Information

The **getgroups** subroutine, **getgidx** subroutine, **setgroups** subroutine, **setuidx** subroutine.

The **setgroups** command.

setgroups Subroutine

Purpose

Sets the concurrent group set of the current process.

Library

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

Syntax

#include <grp.h>

int setgroups (*NumberGroups*, *GroupIDSet***)**
int *NumberGroups*;
gid_t **GroupIDSet*;

Description

The **setgroups** subroutine sets the concurrent group set of the process. The **setgroups** subroutine cannot set more than **NGROUPS_MAX** groups in the group set. (**NGROUPS_MAX** is a constant defined in the **limits.h** header file)

Parameters

GroupIDSet	Pointer to the array of group IDs to be established.
NumberGroups	Indicates the number of entries in the *GroupIDSet* parameter.

Return Values

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

Error Codes

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

EFAULT	The *NumberGroups* and *GroupIDSet* parameters specify an array that is partially or completely outside of the process's allocated address space.
EINVAL	The *NumberGroups* parameter is greater than the **NGROUPS_MAX** value.
EPERM	A group ID in the *GroupIDSet* parameter is not presently in the concurrent group set and the invoker does not have root user authority.

Security

Auditing Events:

Event	**Information**
PROC_SetGroups	*NumberGroups*, *GroupIDSet*

Implementation Specifics

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

Related Information

The **getgid** subroutine, **getgidx** subroutine, **getgroups** subroutine, **initgroups** subroutine, **setgid** subroutine, **setgidx** subroutine.

setjmp or longjmp Subroutine

Purpose

Saves and restores the current execution context.

Library

Standard C Library

Syntax

#include <setjmp.h>

int setjmp (*Context***)**
jmp_buf *Context*;

void longjmp (*Context*, *Value***)**
jmp_buf *Context*;
int *Value*;

int _setjmp (*Context***)**
jmp_buf *Context*;

void _longjmp (*Context*, *Value***)**
jmp_buf *Context*;
int *Value*;

Description

The **setjmp** subroutine and the **longjmp** subroutine are useful when handling errors and interrupts encountered in low-level subroutines of a program.

The **setjmp** subroutine saves the current stack context and signal mask in the buffer specified by the *Context* parameter.

The **longjmp** subroutine restores the stack context and signal mask that were saved by the **setjmp** subroutine in the corresponding *Context* buffer. After the **longjmp** subroutine runs, program execution continues as if the corresponding call to the **setjmp** subroutine had just returned the value of the *Value* parameter. The subroutine that called the **setjmp** subroutine must not have returned before the completion of the **longjmp** subroutine. The **setjmp** and **longjmp** subroutines save and restore the signal mask **sigmask (2),** while **_setjmp** and **_longjmp** manipulate only the stack context.

Parameters

Context	An address for a **jmp_buf** structure.
Value	Any integer value.

Return Values

The **setjmp** subroutine returns a value of 0, unless the return is from a call to the **longjmp** function, in which case **setjmp** returns a non-zero value.

The **longjmp** subroutine cannot return 0 to the previous context. The value 0 is reserved to indicate the actual return from the **setjmp** subroutine when first called by the program.The **longjmp** subroutine does not return from where it was called, but rather, program execution continues as if the corresponding call to **setjmp** was returned with a returned value of *Value*.

If the **longjmp** subroutine is passed a *Value* parameter of 0, then execution continues as if the corresponding call to the **setjmp** subroutine had returned a value of 1. All accessible data have values as of the time the **longjmp** subroutine is called.

Warning: If the **longjmp** subroutine is called with a *Context* parameter that was not previously set by the **setjmp** subroutine, or if the subroutine that made the corresponding call to the **setjmp** subroutine has already returned, then the results of the **longjmp** subroutine are undefined. If the **longjmp** subroutine detects such a condition, it calls the **longjmperror** routine. If **longjmperror** returns, the program is aborted. The default version of **longjmperror** prints the message "longjmp or siglongjmp used outside of saved context" to standard error and returns. Users wishing to exit in another manner can write their own version of the **longjmperror** program.

Implementation Specifics

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

If a process is using the AT & T System V **sigset** interface, then the **setjmp** and **longjmp** subroutines do not save and restore the signal mask. In such a case, their actions are identical to those of the **_setjmp** and **_longjmp** subroutines.

Related Information

The **_setjmp** library routine and **_longjmp** library routine.

The **sigsetjmp, siglongjmp** subroutine.

setlocale Subroutine

Purpose

Changes or queries the program's entire current locale or portions thereof.

Library

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

Syntax

#include <locale.h>

char *setlocale (*Category*, *Locale***)**
int *Category*;
char **Locale*;

Description

The **setlocale** subroutine selects the appropriate portion of the program's locale as specified by the *Category* and *Locale* parameters. The **setlocale** subroutine can be used to change or query the program's entire current locale or portions thereof. The **LC_ALL** value for the *Category* parameter names the entire locale (all the categories); the other values name only a portion of the program locale. **LC_COLLATE** affects the behavior of the **strcoll** and **strxfrm** subroutines. **LC_CTYPE** affects the behavior of the character handling functions and the multibyte functions. **LC_MONETARY** affects the monetary formatting information returned by the **localeconv** subroutine. **LC_NUMERIC** affects the decimal–point character for the formatted input/output subroutines and the string conversion subroutines, as well as the non–monetary formatting information returned by the **localeconv** subroutine. **LC_TIME** affects the behavior of the **strftime** subroutine.

A value of "**C**" for the *Locale* parameter specifies locale; a value of " " specifies the implementation–defined environment. Other implementation–defined strings may be passed in *Locale*.

At program startup, the equivalent of

```
setlocale (LC_ALL, "C");
```

is executed.

Parameters

Category A value from the **locale.h** header file that names the program's entire locale or a portion thereof.

Locale A string defining the locale.

Return Values

If a pointer to a string is given for the *Locale* parameter and the selection can be honored, the **setlocale** subroutine returns the string associated with the specified *Category* parameter for the new locale. If the selection cannot be honored, a **NULL** pointer is returned and the program locale is unchanged.

The string returned by the **setlocale** subroutine is such that a subsequent call with that string and its associated category restores that part of the program locale. The string returned is not modified by the program, but can be overwritten by a subsequent call to the **setlocale** subroutine.

Implementation Specifics

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

Related Information

The **localeconv** subroutine, **ctype** subroutines, **Japanese ctype** subroutines, **string** subroutines.

National Language Support Overview in *General Programming Concepts*.

setpcred Subroutine

Purpose

Sets the current process credentials

Library

Security Library (**libs.a**)

Syntax

#include <usersec.h>

int setpcred (*User, Credentials)*
char ***Credentials*
char **User ;*

Description

The **setpcred** subroutine will set a process's credentials according to the *Credentials* parameter. If the *User* parameter is specified, the credentials defined for the user in the user database are used. If the *Credentials* parameter is specified, the credentials in this string are used. If both the *User* and *Credentials* parameters are specified, both the user's and the supplied credentials will be used, but the supplied credentials of the *Credentials* parameter will override those of the user. At least one parameter must be specified.

Parameters

User Specifies the user for whom credentials are being established.

Credentials Specifies specific credentials to be established. This parameter points to an array of NULL terminated character strings which may contain the following. The last character string must be a NULL.

LOGIN_USER = %s	The login user name.
REAL_USER = %s	The real user name.
REAL_GROUP = %s	The real group name.
GROUPS = %s	The concurrent group set.
AUDIT_CLASSES = %s	The audit classes.
RLIMIT_CPU = %d	The process CPU limit.
RLIMIT_FSIZE = %d	The process maximum file size.
RLIMIT_DATA = %d	The process maximum data segment size.
RLIMIT_STACK = %d	The process maximum stack segment size.
RLIMIT_CORE = %d	The process maximum core file size.
RLIMIT_RSS = %d	The process maximum resident set size.
UMASK = %o	The process **umask**.

Return Values

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

Error Codes

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

EINVAL The *Credentials* parameter contains invalid credentials specifications.

EINVAL The *User* parameter is NULL and the *Credentials* parameter is either NULL or points to an empty string.

EACCES The *User* parameter is specified and the calling process is unable to access the user credentials.

Other errors may be set by any subroutines invoked by the **setpcred** subroutine.

Implementation Specifics

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

Related Information

The **ckuseracct** subroutine, **ckuserID** subroutine, **getpcred** subroutine, **getpenv** subroutine, **setpenv** subroutine.

setpenv Subroutine

Purpose

Sets the current process environment

Library

Security Library (**libs.a**)

Syntax

#include <usersec.h>
int setpenv*(User, Mode, Environment, Command)*
char **User;*
int *Mode;*
char ***Environment;*
char **Command;*

Description

The **setpenv** subroutine will set the environment of the current process according to its parameter values, set the working directory, and run the specified command. The environment consists of both user–state and system–state environment variables.

The **setpenv** subroutine will perform the following steps:

setting the process environment

This step involves changing the user and system state environment. Since this is dependent on the values of the *Mode* and *Environment* parameters, this step is described below in the parameter description for the *Mode* parameter.

setting the process current working directory

After the user and system state environment is reset as required, the working directory of the process is changed. If the **PWD** environment value is defined, the current working directory is set to this value. If **PWD** is not defined, the **HOME** environment value is used instead (and **PWD** is initialized to **HOME**). In either case, this subroutine will fail if the change of the working directory fails. If **HOME** is not defined then "/" the root directory is used.

executing the initial program

After the working directory is reset, the initial program (usually the shell interpreter) is executed. If the *Command* parameter is **NULL**, the shell from the user data base is used, if not defined then the shell from the user state environment is used, if it is not defined the *Command* parameter defaults to the /**bin**/**sh** file. if the Command parameter is not **NULL**, this string is used as the command to be executed. if the *Mode* parameter contains the **PENV_ARGV** value, the *Command* parameter is assumed is be in **argv** format and is simply passed to the **execve** subroutine. The first string is used as the *Path* parameter of **execve**. If *Mode* does not contain **PENV_ARGV**, the *Command* parameter is parsed into an **argv** structure and executed. If the *Command* parameter contains the string **$SHELL**, substitution is done prior to execution.

Note: This step will fail if the *Command* parameter contains the **$SHELL** value and the user state environment does not contain the **SHELL** value.

Parameters

Command
: Specifies the command to be executed. If the *Mode* parameter contains **PENV_ARGV**, then *Command* is assumed to be a valid argument vector for the **execv** subroutine.

Environment
: Specifies the value of user state and system state environment variables in the same format returned by the **getpenv** subroutine. The user state variables are prefaced by the string "**USRENVIRON:**" and the system state variables are prefaced by the string "**SYSENVIRON:**". Each variable is defined by a string of the form **var=***value*, which is an array of character pointers NULL terminated.

Mode
: Specifies how the **setpenv** subroutine is to set the environment and run the command. This parameter is a bit mask and must contain only one of the following values, which are defined in the **usersec.h** file:

PENV_INIT The user–state environment is initialized as follows:

TERM — **TERM** is retained from the current environment. If **TERM** is not present, it is defaulted to an **IBM6155**.

SHELL — **SHELL** is set from the initial program defined for the real user ID of the current process. If no program is defined, then **/bin/sh** is used as the default.

HOME — **HOME** is set from the home directory defined for the real user ID of the current process. If no home directory is defined, the default is **/usr/guest**.

LOGNAME — **LOGNAME** is set to the environment variable **LOGNAME**.

PATH — **PATH** is set initially to the value for **PATH** in the **/etc/environment** file. If it is not set, it is destructively replaced by the default value of **PATH=/usr/bin:$HOME:.** (the period at the end meaning the current working directory). The **PATH** variable is, again, destructively replaced by the **usrenv** attribute for this user in the **/etc/security/environ** file if a value exists.

The following files are read for additional environment variables.

/etc/environment The variables defined in **/etc/environment** are added to the environment.

/etc/security/environ
The environment variables defined for the user are added to the user state environment.

The user state variables in the *Environment* parameter are added to the user–state environment. These are preceded by the **USRENVIRON:** keyword.

The system–state environment is initialized as follows:

LOGIN LOGIN is set to the current LOGIN value in the protected user environment, if not found it is not set. The LOGIN (tsm) command will pass this value to the **setpenv** subroutine to ensure correctness.

LOGNAME LOGNAME is set to the current LOGIN value in the protected user environment, if not found it is not set. The LOGIN (tsm) command will pass this value to the **setpenv** subroutine to ensure correctness.

NAME NAME is set to the login name corresponding to the real user ID.

TTY TTY is set to the tty name corresponding to standard input.

The following file is read for additional environment variables.

/etc/security/environ
The system state environment variables defined for the user are added. The system–state variables in the Environment parameter are added. These are preceded by the **SYSENVIRON** keyword.

PENV_DELTA The existing user and system state environment variables are preserved and the variables defined in the *Environment* parameter are added.

PENV_RESET The existing environment is cleared and totally replaced by the content of the *Environment* parameter.

PENV_KLEEN All the open file descriptors will be closed before executing the command. This value must be ORed with **PENV_DELTA**, **PENV_RESET**, or **PENV_INIT**. It cannot be used alone.

For both system state and user state environment, variable substitution is performed.

The *Mode* parameter may also contain:

PENV_ARGV Specifies that the *Command* parameter is already in **argv** format and need not be parsed. This value must be ORed with **PENV_DELTA**, **PENV_RESET**, or **PENV_INIT**. It cannot be used alone.

Return Values

If the environment was successfully established, this function does not return. If the **setpenv** subroutine fails, a value of –1 is returned and **errno** is set to indicate the error.

Error Codes

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

EINVAL The *Mode* parameter contains values other than **PENV_INIT**, **PENV_DELTA**, **PENV_RESET**, or **PENV_ARGV**.

EINVAL The *Mode* parameter contains more than one of **PENV_INIT**, **PENV_DELTA**, or **PENV_RESET**.

EINVAL The *Environment* parameter in not NULL or empty and does not contain a valid environment string.

Implementation Specifics

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

Related Information

The **execv** subroutine, **getpenv** subroutine.

The **su** command, **login** command.

setpgid or setpgrp Subroutine

Purpose

Sets the process group ID.

Libraries

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

setpgrp: Standard C Library (**libc.a**); Berkeley Compatibility Library (**libbsd.a**)

Syntax

#include <sys/types.h>

int setpgid (*ProcessID,ProcessGroupID***)**
pid_t *ProcessID, ProcessGroupID***;**

int setpgrp ( )

Description

The **setpgid** subroutine is used either to join an existing process group or to create a new process group within the session of the calling process. The process group ID of a session leader will not change. Upon return, the process group ID of the process with a process ID that matches *ProcessID* is set to *ProcessGroupID*. As a special case, if *ProcessID* is 0, the process ID of the calling process is used. If *ProcessGroupID* is 0, the process ID of the indicated process is used.

This function is implemented to support job control.

The **setpgrp** subroutine in **libc.a** supports a subset of the function of the **setpgid** subroutine. It has no parameters. It sets the process group ID of the calling process to be the same as its process ID and returns the new value.

Parameters

ProcessID	Specifies the process whose process group ID is to be changed.
ProcessGroupID	Specifies the new value of calling process group ID.

Return Values

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

Error Codes

The **setpgid** subroutine fails If one or more of the following are true:

EACCES	The value of the *ProcessID* parameter matches the process ID of a child process of the calling process and the child process has successfully executed one of the **exec** subroutines.
EINVAL	The value of the *ProcessGroupID* parameter is less than 0, or is not a valid value.
ENOSYS	The **setpgid** subroutine is not supported by this implementation.

EPERM The process indicated by the value of the *ProcessID* parameter is a session leader.

EPERM The value of *ProcessID* parameter matches the process ID of a child process of the calling process and the child process is not in the same session as the calling process.

EPERM The value of *ProcessGroupID* parameter is valid but does not match the process ID of the process indicated by the *ProcessID* parameter and there is no process with a process group ID that matches the value of the *ProcessGroupID* parameter in the same session as the calling process.

ESRCH The value of *ProcessID* parameter does not match the process ID of the calling process of a child process of the calling process.

Implementation Specifics

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

In BSD systems, the **setpgrp** subroutine is defined with two parameters, as follows:

int setpgrp (*ProcessID, ProcessGroup***)**
int *ProcessID, ProcessGroup*;

BSD systems set the process group to the value specified by the *ProcessGroup* parameter. If the *ProcessID* parameter has a value of 0, the call applies to the current process. In the AIX Version 3 Operating System, this version of the **setpgrp** subroutine can be used by compiling with the Berkeley Compatibility Library (**libbsd.a**) and is implemented as a call to the **setpgid** subroutine. The restrictions that apply to the **setpgid** subroutine also apply to the **setpgrp** subroutine.

Related Information

The **getpid** subroutine.

setpri Subroutine

Purpose

Sets a process scheduling priority to a constant value.

Library

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

Syntax

#include <sys/sched.h>

int setpri (*ProcessID*, *Priority***)**
pid_t pid;
int pri;

Description

The **setpri** subroutine sets the scheduling priority of a process to be a constant. A process *nice* value and cpu usage will no longer be used to determine the process scheduling priority. Only processes that have root user authority may set a process scheduling priority to a constant.

Parameters

ProcessID — Specifies the process ID. If this value is zero then the current process scheduling priority is set to a constant.

Priority — Specifies the scheduling priority for the process. *Priority* must be in the range **PRIORITY_MIN** < pri < **PRIORITY _MAX**. (See the **sys/sched.h** header file.)

Return Values

Upon successful completion, the **setpri** subroutine returns the former scheduling priority of the process just changed. Otherwise, a value of –1 is returned and the global variable **errno** is set to indicate the error.

Error Codes

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

EINVAL — The priority specified by the *Priority* parameter is outside the range of acceptable priorities.

EPERM — The process executing the **setpri** system call does 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 AIX Base Operating System (BOS) Runtime.

Related Information

The **getpri** subroutine.

setpwdb or endpwdb Subroutine

Purpose

Opens and closes the authentication database.

Library

Security Library (**libs.a**)

Syntax

#include <usersec.h>

int setpwdb(*Mode***)**
int *Mode***;**

int endpwdb( )

Description

These functions may be used to open and close access to the authentication database. Programs which call either the **getuserpw** or **setuserpw** subroutine should call **setpwdb** to open the database and **endpwdb** to close the database.

The **setpwdb** subroutine will open the authentication database in the specified mode, if it is not already open. The open count is incremented by one.

The **endpwdb** subroutine will decrement the open count by one and close the authentication database when this count goes to zero. Any uncommitted changed data is lost.

Parameter

Mode Specifies the mode of the open. This parameter may contain one or more of the following values, defined in the **usersec.h** file:

S_READ Specifies read access

S_WRITE Specifies update access. If the process has previously opened the database for read access and then attempts to open the database for write access, the open may fail.

Return Values

The **setpwdb** and **endpwdb** subroutines return 0 to indicate success. Otherwise, –1 is returned and the global variable **errno** is set to indicate the error.

Error Codes

The **setpwdb** and **endpwdb** subroutines fail if the following is true:

EACCESS Access permission is denied for the data request.

Both of these functions will return errors from other subroutines.

Security

file access The calling process must have access to the authentication data. This includes:

modes	**file**
rw	**/etc/security/passwd**
rw	**/etc/passwd**

Implementation Specifics

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

Related Information

The **getgroupattr** subroutine, **getuserattr** subroutine, **getuserpw** subroutine, **setuserdb** subroutine.

setsid Subroutine

Purpose

Creates a session and sets the process group ID.

Library

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

Syntax

#include <sys/types.h>

int setsid( )

Description

The **setsid** subroutine creates a new session, if the calling process is not a process group leader. Upon return, the calling process is the session leader of this new session, the process group leader of a new process group, and has no controlling terminal. The process group ID of the calling process is set equal to its process ID. The calling process will be the only process in the new process group and the only process in the new session.

Return Values

Upon successful completion, the value of the new process group ID is returned. Otherwise, (**pid_t**) –1 is returned and the global variable **errno** is set to indicate the error.

Error Code

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

EPERM The calling process is already a process group leader, or the process group ID of a process other than the calling process matches the process ID of the calling process.

Implementation Specifics

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

Related Information

The **getpid**, **getpgrp**, **getppid** subroutines, **fork** subroutine, **setpgid** subroutine, **setpgrp** subroutine.

setuid, setruid, seteuid, or setreuid Subroutine

Purpose

Sets the process user IDs.

Library

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

Syntax

#include <sys/id.h>
#include <sys/types.h>

int setuid(*UID***)**
uid_t *UID***;**

int setruid(*RUID***)**
uid_t *RUID***;**

int seteuid(*EUID***)**
uid_t *EUID***;**

int setreuid(*RUID*, *EUID***)**
uid_t *RUID***;**
uid_t *EUID***;**

Description

These subroutines reset the process's user IDs as follows:

setuid If the invoker is the root user, the process's real, effective, and saved user IDs are set to the value of the *UID* parameter. Otherwise, the process's effective user ID is reset if *UID* is equal to either the current real or saved user IDs.

seteuid The process's effective user ID is reset if *UID* is equal to either the current real or saved user IDs.

setruid **EPERM** is always returned. Processes cannot reset only their real user IDs.

setreuid There are two cases:

RUID != *EUID*
If *EUID* is equal to either the process's real or saved user IDs, the process's effective user ID is set to *EUID*; else **EPERM** is returned.

RUID= = *EUID*
If the invoker is the root user, the process's real, effective, and saved user IDs are set to *EUID*; else **EPERM** is returned.

Parameters

UID Specifies the user ID to set.

EUID Specifies the effective user ID to set.

RUID Specifies the real user ID to set.

Return Values

Upon successful completion, the **setuid**, **seteuid**, and **setreuid** subroutines return 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 **setuid**, **seteuid**, and **setreuid** subroutines fail if:

EPERM The *UID* or *EUID* parameters are not equal to either the real or saved user IDs of the process.

Implementation Specifics

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

Related Information

The **getuidx** subroutine, **setuidx** subroutine.

The **getuid** subroutine.

setuidx Subroutine

Purpose

Sets the process user IDs.

Library

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

Syntax

#include <sys/id.h>
#include <sys/types.h>

uid_t setuidx (*Which*, *UID*)
int *Which*;
uid_t *UID*

Description

The **setuidx** subroutine sets the specified user IDs of the current process.

Parameters

Which — Specifies which user ID to return. This parameter is a bitmask and may include one or more of the following, which are defined in **sys/id.h**:

ID_EFFECTIVE	Sets the effective user ID of the process.
ID_REAL	Sets the real user ID of the process.
ID_SAVED	Sets the saved user ID of the process.
ID_LOGIN	Sets the login user ID of the process.

UID — Specifies the value of the user ID to be set.

Return Values

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

Error Codes

The **setuidx** subroutine fails if:

EPERM — *Which* = ID_EFFECTIVE and the *UID* parameter are not equal to either the real or saved user IDs of the process, or the invoker is not the root user and ID_REAL and/or ID_SAVED were specified.

EINVAL — The *Which* parameter does not contain a valid combination of IDs to be changed.

Security

DAC Policy

The following policies are enforced

Only the root user can change the real or saved user IDs

If the real ID is changed, the effective ID must be changed to the same value. If the saved ID is changed, the real and effective ID must be changed to the same values.

The effective ID may be changed only to the current real or saved IDs.

Implementation Specifics

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

Related Information

The **getuidx** subroutine, **setuid** subroutine.

The **getuid** subroutine.

setuserdb or enduserdb Subroutine

Purpose

Opens and closes the user database.

Library

Security Library (**libs.a**)

Syntax

#include <usersec.h>

int setuserdb(*Mode***)**
int *Mode***;**

int enduserdb( )

Description

These functions may be used to open and close access to the user database. Programs which call either the **getuserattr** or **getgroupattr** subroutine should use these functions.

The **setuserdb** subroutine will open the user database in the specified mode, if it is not already open. The open count is incremented by one.

The **endpwdb** subroutine will decrement the open count by one and close the user database when this count goes to zero. Any uncommitted changed data is lost.

Parameter

Mode Specifies the mode of the open. This parameter may contain one or more of the following values, defined in the **usersec.h** file:

S_READ Specifies read access

S_WRITE Specifies update access. If the process has previously opened the database for read access and then attempts to open the database for write access, the open may fail.

Return Values

The **setuserdb** and **enduserdb** subroutines return 0 to indicate success. Otherwise, –1 is returned and the global variable **errno** is set to indicate the error.

Error Codes

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

EACCESS Access permission is denied for the data request.

Both of these functions will return errors from other subroutines.

Security

file access The calling process must have access to the user data. Depending on the actual attributes accessed, this may include:

modes	file
rw	/etc/passwd
rw	/etc/group
rw	/etc/security/user
rw	/etc/security/limits
rw	/etc/security/audit/audit.config
rw	/etc/security/group
rw	/etc/security/environ

Implementation Specifics

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

Related Information

The **getgroupattr** subroutine, **getuserattr** subroutine, **getuserpw** subroutine, **setpwdb** subroutine.

sgetl or sputl Subroutine

Purpose

Accesses long numeric data in a machine-independent fashion.

Library

Object File Access Routine Library (**libld.a**)

Syntax

long sgetl (*Buffer***)**
**char ***Buffer*;

void sputl (*Value*, *Buffer***)**
long *Value*;
**char ***Buffer*;

Description

The **sgetl** subroutine retrieves four bytes from memory starting at the location pointed to by the *Buffer* parameter. It then returns the bytes as a long *Value* with the byte ordering of the host machine.

The **sputl** subroutine stores the four bytes of the *Value* parameter into memory starting at the location pointed to by the *Buffer* parameter. The order of the bytes is the same across all machines.

Using the **sputl** and **sgetl** subroutines together provides a machine-independent way of storing long numeric data in an ASCII file. For example, the numeric data stored in the portable archive file format is accessed with the **sputl** and **sgetl** subroutines.

Parameters

Value Specifies a 4-byte value to store into memory.

Buffer Points to a location in memory.

Implementation Specifics

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

Related Information

The **ar** command, **dump** command.

The **ar** file format, **a.out** file format.

shmat Subroutine

Purpose

Attaches a shared memory segment or a mapped file to the current process.

Syntax

#include <sys/shm.h>
#include <sys/types.h>
#include <sys/ipc.h>

char *shmat (*SharedMemoryID*, *SharedMemoryAddress*, *SharedMemoryFlag***)**
int *SharedMemoryID*;
**char ****SharedMemoryAddress*;
int *SharedMemoryFlag*;

Description

The **shmat** subroutine attaches the shared memory segment or mapped file associated with the *SharedMemoryIdentifier* (returned by the **shmget** subroutine), or *FileDescriptor* (returned by the **openx** subroutine) specified by the *SharedMemoryID* parameter to the address space of the calling process.

Parameters

SharedMemoryID
: Specifies an identifier for the shared memory segment.

SharedMemoryAddress
: Identifies the segment or file attached at the address specified by the *SharedMemoryAddress* parameter as follows:

- If the *SharedMemoryAddress* parameter is equal to 0, the segment or file is attached at the first available address as selected by the system.
- If the *SharedMemoryAddress* parameter is *not* equal to 0, and the SHM_RND value is set in the *SharedMemoryFlag* parameter, the segment or file is attached at the next lower segment boundary. This address is given by (*SharedMemoryAddress* –(*SharedMemoryAddress* modulo SHMLBA)).
- If the *SharedMemoryAddress* parameter is not equal to 0 and the SHM_RND value is not set in the *SharedMemoryFlag* parameter, the segment or file is attached at the address given by the *SharedMemoryAddress* parameter. If this address does not point to a segment boundary, the **shmat** subroutine returns the value –1 and sets the global variable **errno** to EINVAL.

SharedMemoryFlag	The *SharedMemoryFlag* parameter specifies several options. Its value is either 0, or is constructed by logically ORing one or more of the following values:

SHM_COPY	Changes an open file to deferred update (see the **openx** system call). Included only for compatibility with previous AIX versions.
SHM_MAP	Maps a file onto the address space instead of a shared memory segment. The *SharedMemoryID* must specify an open file descriptor in this case.
SHM_RDONLY	Specifies read-only mode instead of the default read-write mode.
SHM_RND	Rounds the address given by the *SharedMemoryAddress* parameter to the next lower segment boundary, if necessary.

The **shmat** program makes a shared memory segment addressable by the current process. The segment is attached for reading if SHM_RDONLY is set with *SharedMemoryFlag* and if the current process has read permission. If SHM_RDONLY is not set and the current process has both read and write permission, it is attached for reading and writing.

If SHM_MAP is set in *SharedMemoryFlag*, file mapping takes place. In this case, the **shmat** subroutine maps the file open on the file descriptor specified by the *SharedMemoryID* onto a segment. The file must be a regular file. The segment is then mapped into the address space of the process. Any size file can be mapped if there is enough space in the user address space.

When file mapping is requested, the *SharedMemoryFlag* parameter specifies how the file is to be mapped. If SHM_RDONLY is set, the file is mapped read-only. To map read-write, the file must have been opened for writing.

All processes that map the same file read-only or read-write map to the same segment. This segment remains mapped until the last process mapping the file closes it.

If the file that was mapped is opened with the O_DEFER update, the mapped file also has deferred update. Changes to the shared segment do not affect the contents of the file resident in the file system until an **fsync** subroutine is issued to the file descriptor for which the mapping was requested. Setting the SHM_COPY flag causes the file to be changed to the deferred state. It remains in this state until all processes close the file. The SHM_COPY flag is only for AIX Version 2 compatibility. New programs should use the open flag 0_DEFER.

A file descriptor can be used to map the corresponding file only once. A file can be mapped several times by using multiple file descriptors.

When a file is mapped onto a segment, the file is referenced by accessing the segment. The memory paging system automatically takes care of the physical I/O. References beyond the end of the file cause the file to be extended in page-sized increments.

Return Values

Upon successful completion, the segment start address of the attached shared memory segment or mapped file is returned.

Error Codes

If the **shmat** subroutine is unsuccessful, a value of –1 is returned and the global variable **errno** is set to indicate the error. The **shmat** subroutine is unsuccessful and the shared memory segment or mapped file is not attached if one or more of the following are true:

EACCES	Operation permission is denied to the calling process.
EACCES	The file to be mapped has enforced locking enabled and the file is currently locked.
EACCES	The SHM_RDONLY and SHM_COPY flags are both set.
EBADF	A file descriptor to map does not refer to an open regular file.
EEXIST	The file to be mapped has already been mapped.
EINVAL	The *SharedMemoryID* parameter is not a valid shared memory identifier.
EINVAL	The *SharedMemoryAddress* parameter is not equal to 0, and the value of (*SharedMemoryAddress* – (*SharedMemoryAddress* modulo SHMLBA)) points outside the address space of the process.
EINVAL	The *SharedMemoryAddress* parameter is not equal to 0, SHM_RND is not set in *SharedMemoryFlag*, and the *SharedMemoryAddress* parameter points to a location outside of the address space of the process.
EINVAL	The *SharedMemoryAddress* parameter is not equal to 0, SHM_RND is not set in *SharedMemoryFlag*, and the *SharedMemoryAddress* parameter does not point to a segment boundary.
EMFILE	The number of shared memory segments attached to the calling process exceeds the system-imposed limit.
ENOMEM	The available data space in memory is not large enough to hold the shared memory segment.
ENOMEM	The available data space in memory is not large enough to hold the mapped file data structure.

Implementation Specifics

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

Related Information

The **exec** subroutine, **exit** subroutine, **fclear** subroutine,**fork** subroutine, **fsync** subroutine, **truncate** subroutine, **readvx** subroutine, **shmctl** subroutine, **shmdt** subroutine, **shmget** subroutine, **writevx** subroutine.

shmctl Subroutine

Purpose

Controls shared memory operations.

Syntax

#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/shm.h>

int shmctl (*SharedMemoryID*, *Command*, *Buffer*)
int *SharedMemoryID, Command*;
struct shmid_ds **Buffer;*

Description

The **shmctl** subroutine performs a variety of shared memory control operations as specified by the *Command* parameter.

Parameters

SharedMemoryID — Specifies an identifier returned by the **shmget** subroutine.

Buffer — Pointer to **shmid_ds** struct.

Command — The following commands are available:

IPC_STAT — This command obtains status information about the shared memory segment identified by the *SharedMemoryID* parameter. This information is stored in the area pointed to by the *Buffer* parameter. The calling process must have read permission to run this command. The **shmid_ds** structure is defined in the **sys/shm.h** header file, and it contains the following members:

```
struct ipc_perm shm_perm;
/* Operation permissions struct*/
int shm_segsz;
/* Segment size*/
pid_t shm_lpid;
/* ID of last process to call shmop*/
pid_t shm_cpid;
/* ID of process that created this
      SharedMemoryID*/
unsigned short shm_nattch;
/* Current number of processes
        attached */
/* No. of in–memory processes
        attached */
time_t shm_atime;
/* Time of last shmat call */
time_t shm_dtime;
/* Time of last shmdt call */
time_t shm_ctime;
/* Time of the last change to this */
```

IPC_ SET

Set the owning user and group IDs, and the access permissions for the shared memory segment associated with the *SharedMemoryID* parameter. Sets the following fields:

```
shm_perm.uid    /* owning user ID*/
shm_perm.gid    /* owning group ID*/
shm_perm.mode   /* permission bits only
*/
```

You must have an effective user ID equal to root or to the value of **shm_perm.cuid** or **shm_perm.uid**.

IPC_RMID

Removes the shared memory identifier specified by the *SharedMemoryID* parameter from the system and erases the shared memory segment and data structure associated with it. This command can only be run to the value of **shm_perm.uid** or **shm_perm.cuid** in the data structure associated with the *SharedMemoryID* parameter, or by a process that has an effective root user ID.

SHM_SIZE

Sets the size of the shared memory segment to the value specified by *Buffer*->**shm_segsz**. This value can be larger or smaller than the current size; the limit is the hardware segment size. This command can only be run to the value of **shm_perm.uid** or **shm_perm.cuid** in the data structure associated with the *SharedMemoryID* parameter, or by a process that has an effective root user ID.

Return Value

Upon successful completion, a value of 0 is returned.

Error Codes

If the **shmctl** subroutine is unsuccessful, a value of –1 is returned and the global variable **errno** is set to indicate the error. The **shmctl** subroutine is unsuccessful if one or more of the following are true:

EACCES	The *Command* parameter is equal to the IPC_STAT value and read permission is denied to the calling process.
EFAULT	The *Buffer* parameter points to a location outside the allocated address space of the process.
EINVAL	The *SharedMemoryID* parameter is not a valid shared memory identifier.
EINVAL	The *Command* parameter is not a valid command.
EINVAL	The *Command* parameter is equal to the SHM_SIZE value and *Buffer*–>**shm_segsz** is greater than the value of the hardware segment size limit.
ENOMEM	The *Command* parameter is equal to SHM_SIZE and the attempt to change the segment size is unsuccessful because the system does not have enough memory.
EPERM	The *Command* parameter is equal to the IPC_RMID or SHM_SIZE value and the effective user ID of the calling process is not equal to the value of **shm_perm.uid** or **shm_perm.cuid** in the data structure associated with the *SharedMemoryID* parameter. The effective user ID of the calling process is not the root user ID.

Implementation Specifics

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

Related Information

The **disclaim** subroutine, **shmat** subroutine, **shmdt** subroutine, **shmget** subroutine.

shmdt Subroutine

Purpose

Detaches a shared memory segment.

Syntax

#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/shm.h>

int shmdt (*SharedMemoryAddress*)
char **SharedMemoryAddress*;

Description

The **shmdt** subroutine detaches, from the data segment of the calling process, the shared memory segment located at the address specified by the *SharedMemoryAddress* parameter.

Mapped file segments are automatically detached when the mapped file is closed. However, you can use the **shmdt** subroutine to explicitly release the segment register used to map a file. Shared memory segments must be explicitly detached with the **shmdt** subroutine.

If the file was mapped for writing, the **shmdt** system call updates the **mtime** and **ctime** time stamps.

Parameter

SharedMemoryAddress	Specifies the data segment start address of a shared memory segment.

Return Value

Upon successful completion, a value of 0 is returned.

Error Code

If the **shmdt** subroutine is unsuccessful, a value of −1 is returned and the global variable **errno** is set to indicate the error. The **shmdt** subroutine is unsuccessful and the shared memory segment is not detached if the following is true:

EINVAL	The *SharedMemoryAddress* parameter is not the data segment start address of a shared memory segment.

Implementation Specifics

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

Related Information

The **exec** subroutine, **fork** subroutine, **fsync** subroutine, **shmat** subroutine, **shmctl** subroutine, **shmget** subroutine, **exit** subroutine.

shmget Subroutine

Purpose

Gets shared memory segments.

Syntax

#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/shm.h>

int shmget (*Key*, *Size*, *SharedMemoryFlag***)**
key_t *Key*;
int *Size*, *SharedMemoryFlag*;

Description

The **shmget** subroutine returns the shared memory identifier associated with the specified *Key* parameter.

Parameters

Key Either the value IPC_PRIVATE or an IPC key constructed by the **ftok** subroutine (or by a similar algorithm).

Size The number of bytes of shared memory required.

SharedMemoryFlag

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 **shmget** subroutine to be unsuccessful if IPC_CREAT 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.

The values that begin with the **S_I–** prefix are defined in the **sys/mode.h** header file and are a subset of the access permissions that apply to files.

A shared memory identifier, its associated data structure, and a shared memory segment equal in number of bytes to the value of the *Size* parameter are created for the *Key* parameter if one of the following is true:

- The *Key* parameter is equal to the IPC_PRIVATE value.
- The *Key* parameter does not already have a shared memory identifier associated with it, and the IPC_CREAT value is set.

Upon creation, the data structure associated with the new shared memory identifier is initialized as follows:

- **shm.perm.cuid** and **shm.perm.uid** are set equal to the effective user ID of the calling process.
- **shm.perm.cgid** and **shm.perm.gid** are set equal to the effective group ID of the calling process.
- The low-order 9 bits of **shm.perm.mode** are set equal to the low-order 9 bits of the *SharedMemoryFlag* parameter.
- **shm.segsz** is set equal to the value of the *Size* parameter.
- **shm.lpid**, **shm.nattch**, **shm.atime**, and **shm.dtime** are set equal to 0.
- **shm.ctime** is set equal to the current time.

Return Values

Upon successful completion, a shared memory identifier is returned.

Error Codes

If the **shmget** subroutine is unsuccessful, a value of –1 is returned and the global variable **errno** is set to indicate the error. The **shmget** subroutine is unsuccessful if one or more of the following are true:

EACCES	A shared memory identifier exists for the *Key* parameter but operation permission as specified by the low-order 9 bits of the *SharedMemoryFlag* parameter is not granted.
EEXIST	A shared memory identifier exists for the *Key* parameter, and both IPC_CREAT and IPC_EXCL are set.
EINVAL	The *Size* parameter is less than the system-imposed minimum or greater than the system-imposed maximum.
EINVAL	A shared memory identifier exists for the *Key* parameter, but the size of the segment associated with it is less than the *Size* parameter and the *Size* parameter is not equal to 0.
ENOENT	A shared memory identifier does not exist for the *Key* parameter and IPC_CREAT not set.

ENOMEM A shared memory identifier and associated shared memory segment are to be created but the amount of available physical memory is not sufficient to fill the request.

ENOSPC A shared memory identifier is to be created but the system-imposed limit on the maximum number of allowed, shared memory identifiers system-wide will be exceeded.

Implementation Specifics

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

Related Information

The **shmat** subroutine, **shmctl** subroutine, **shmdt** subroutine, **ftok** subroutine.

sigaction, sigvec or signal Subroutine

Libraries

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

signal, sigvec: Standard C Library (**libc.a**); Berkeley Compatibility Library (**libbsd.a**)

Purpose

Specifies the action to take upon delivery of a signal.

Syntax

#include <signal.h>

int sigaction (*Signal*, *Action*, *OAction***)**
int *Signal*;
struct sigaction **Action*, **OAction*;

int sigvec(*Signal*, *Invec*, *Outvec***)**
int *Signal*;
struct sigvec **Invec*, **Outvec*;

void (*signal(*Signal*, *Action***))( )**
int *Signal*;
void (**Action***)( );**

Description

The **sigaction** subroutine allows the calling process to examine and/or change the action to be taken when a specific signal is delivered to the process issuing this subroutine.

The *Signal* parameter specifies the signal. If the *Action* parameter is not **NULL**, it points to a **sigaction** structure that describes the action to be taken on receipt of the *Signal* parameter signal. If the *OAction* parameter is not **NULL**, it points to a **sigaction** structure in which the signal action data in effect at the time of the **sigaction** call is returned. If the *Action* parameter is **NULL**, signal handling is unchanged; thus, the call can be used to inquire about the current handling of a given signal.

The **sigaction** structure has the following members:

```
void           (*sa_handler)();
sigset_t       sa_mask;
int            sa_flags;
```

The sa_handler field can have the **SIG_DFL** or **SIG_IGN** value, or it can be a pointer to a function. A **SIG_DFL** value requests default action to be taken when the signal is delivered. A value of **SIG_IGN** requests that the signal have no effect on the receiving process. A pointer to a function requests that the signal be caught; that is, the signal should cause the function to be called. These actions are more fully described following the list of supported signal values.

The sa_mask field can be used to specify that individual signals, in addition to those in the process signal mask, be blocked from being delivered while the signal handler function specified in the sa_handler is executing. The sa_flags field can have the **SA_ONSTACK**, **SA_OLDSTYLE**, or **SA_NOCLDSTOP** bits set to specify further control over the actions taken on delivery of a signal.

If the **SA_ONSTACK** bit is set, the system runs the signal–catching function on the signal stack specified by the **sigstack** subroutine. If this bit is not set, the function runs on the stack of the process to which the signal is delivered.

If the **SA_OLDSTYLE** bit is set, the signal action is set to **SIG_DFL** prior to calling the signal–catching function. This is supported for compatibility with old applications, and is not recommended since the same signal can recur before the signal–catching routine is able to reset the signal action and the default action (normally termination) is taken in that case.

If a signal for which a signal–catching function exists is sent to a process while that process is executing certain subroutines, the call can be restarted if the **SA_RESTART** bit is set for each signal. The only affected subroutines are the following:

- **read**, **readx**, **readv**, **readvx**
- **write**, **writex**, **writev**, **writevx**
- **ioctl**, **ioctlx**
- **fcntl**, **lockf**, **flock**
- **wait**, **wait3**, **waitpid.**

Other subroutines do not restart and return **EINTR**, independent of the setting of **SA_RESTART**.

The *Signal* parameter can be any one of the following signal values except **SIGKILL**. Each of the names shown in the following list is defined in the **signal.h** header file with the value of the corresponding signal number.

SIGHUP	1	Hangup.
SIGINT	2	Interrupt.
SIGQUIT	3*	Quit.
SIGILL	4*	Invalid instruction (not reset when caught).
SIGTRAP	5*	Trace trap (not reset when caught).
SIGIOT	6*	End process (see the **abort** subroutine).
SIGEMT	7*	EMT instruction.
SIGFPE	8*	Arithmetic exception, integer divide by 0, or floating–point exception.
SIGKILL	9	Kill (cannot be caught or ignored).
SIGBUS	10*	Specification exception.
SIGSEGV	11*	Segmentation violation.
SIGSYS	12*	Invalid parameter to subroutine.
SIGPIPE	13	Write on a pipe when there is no process to read it.
SIGALRM	14	Alarm clock.
SIGTERM	15	Software termination signal.

SIGURG	16+	Urgent condition on I/O channel.
SIGSTOP	17@	Stop (cannot be caught or ignored).
SIGTSTP	18@	Interactive stop.
SIGCONT	19!	Continue if stopped.
SIGCHLD	20+	To parent on child stop or exit.
SIGTTIN	21@	Background read attempted from control terminal.
SIGTTOU	22@	Background write attempted from control terminal.
SIGIO	23+	Input/Output possible or completed.
SIGXCPU	24	CPU time limit exceeded (see **setrlimit**).
SIGXFSZ	25	File size limit exceeded (see **setrlimit**).
reserved	26	
SIGMSG	27#	Input data has been stored into the HFT monitor mode ring buffer.
SIGWINCH	28+	Window size change.
SIGPWR	29+	Power–fail restart.
SIGUSR1	30	User–defined signal 1.
SIGUSR2	31	User–defined signal 2.
SIGPROF	32	Profiling time alarm (see **setitimer**).
SIGDANGER	33+	System crash imminent.
SIGVTALRM	34	Virtual time alarm (see **setitimer**).
SIGMIGRATE	35	Migrate process.
SIGPRE	36	Programming exception (user defined).
reserved	37–58	
SIGGRANT	60#	HFT monitor access wanted.
SIGRETRACTION	61#	HFT monitor access should be relinquished.
SIGSOUND	62#	An HFT sound control has completed execution.
SIGSAK	63	Secure attention key.

The symbols in the preceding table have the following meaning:

*	Default action includes creating a core dump file.
@	Default action is to stop the process receiving these signals.
!	Default action is to restart or continue the process receiving these signals.

+	Default action is to ignore these signals.
%	The most likely cause for this signal is a shortage of paging space.
#	For more information on the use of these signals, see *Terminal Programming*.

The three types of actions that can be associated with a signal: **SIG_DFL**, **SIG_IGN**, or a pointer to a function are described as follows:

SIG_DFL Default action: signal–specific default action.

Except for those signal numbers marked with a +, @, or !, the default action for a signal is to end the receiving process with all of the consequences described in the **_exit** subroutine. In addition, a memory image file is created in the current directory of the receiving process if the *Signal* parameter is one for which an asterisk appears in the preceding list and the following conditions are met:

- The effective user ID and the real user ID of the receiving process are equal
- An ordinary file named **core** exists in the current directory and is writable, or it can be created. If the file must be created, it will have the following properties:
  - The access permission code 0666 (0x1B6), modified by the file creation mask (see the **umask** subroutine)
  - A file owner ID that is the same as the effective user ID of the receiving process
  - A file group ID that is the same as the effective group ID of the receiving process.

For signal numbers marked with a !, the default action is to restart the receiving process if it is stopped, or to continue execution of the receiving process.

For signal numbers marked with a @, the default action is to stop the execution of the receiving process temporarily. When a process stops, a **SIGCHLD** signal is sent to its parent process, unless the parent process has set the **SA_NOCLDSTOP** bit. While a process is stopped, any additional signals that are sent to the process are not delivered until the process is continued. An exception to this is **SIGKILL**, which always terminates the receiving process. Another exception is **SIGCONT**, which always causes the receiving process to restart or continue running. A process whose parent has ended shall be sent a **SIGKILL** signal if the **SIGTSTP**, **SIGTTIN**, or **SIGTTOU** signals are generated for that process.

For signal numbers marked with a +, the default action is to ignore the signal. In this case, delivery of the signal has no effect on the receiving process.

If a signal action is set to **SIG_DFL** while the signal is pending, the signal remains pending.

SIG_IGN Ignore signal.

Delivery of the signal has no effect on the receiving process. If a signal action is set to **SIG_IGN** while the signal is pending, the pending signal is discarded.

An exception to this is the **SIGCHLD** signal whose **SIG_DFL** action is to ignore the signal. If **SIGCHLD** is set to **SIG_IGN**, it means that the process does not want to receive the **SIGCHLD** signal when one of its child processes dies, and does not want to have its **wait** calls return because a child process is dead (just when no more child processes exist, and on stopped child processes).

Note: The **SIGKILL** and **SIGSTOP** signals cannot be ignored.

pointer to a function Catch signal.

Upon delivery of the signal, the receiving process is to run the signal–catching function specified by the pointer to function. The signal–handler subroutine can be declared as follows:

```
handler(Signal, Code, SCP)
int Signal, Code;
struct sigcontext *SCP;
```

The *Signal* parameter is the signal number. The *Code* parameter is provided only for compatibility with other UNIX compatible systems, and its value is always 0. The *SCP* parameter points to the **sigcontext** structure that is later used to restore the previous execution context of the process. The **sigcontext** structure is defined in the **signal.h** header file.

A new signal mask is calculated and installed for the duration of the signal–catching function (or until **sigprocmask** or **sigsuspend** subroutines are made). This mask is formed by taking the union of the process signal mask, the mask associated with the action for the signal being delivered, and a mask corresponding to the signal being delivered. The mask associated with the signal–catching function is not allowed to block those signals that cannot be ignored. This is enforced by the kernel without causing an error to be indicated. If and when the signal–catching function returns, the original signal mask is restored (modified by any **sigprocmask** calls that were made since the signal–catching function was called) and the receiving process resumes execution at the point it was interrupted.

The signal–catching function can cause the process to resume in a different context by calling the **longjmp** subroutine. When the **longjmp** subroutine is called, the process leaves the signal stack, if it is currently on it, and restores the process signal mask to the state when the corresponding **setjmp** call was made.

Once an action is installed for a specific signal, it remains installed until another action is explicitly requested (by another call to the **sigaction** subroutine), or until one of the **exec** subroutines is called. An exception to this is when the **SA_OLDSTYLE** is set in which case the action of a caught signal gets set to **SIG_DFL** prior to calling the signal–catching function for that signal.

If a signal action is set to a pointer to a function while the signal is pending, the signal remains pending.

When signal–catching functions are invoked asynchronously with process execution, the behavior of some of the functions defined by this standard is unspecified if they are called from a signal–catching function. The following set of functions are reentrant with respect to signals (that is, applications can invoke them, without restriction, from signal–catching functions):

_exit(), access(), alarm(), chdir(), chmod(), chown(), close(), creat(), dup2(), dup(), exec(), fcntl(), fork(), fstat(), getegid(), geteuid(), getgid(), getgroups(), getpgrp(), getpid(), getppid(), getuid(), kill(), link(), lseek(), mkdir(), mkfifo(), open(), pause(), pipe(), readx(), rename(), rmdir(), setgid(), setpgrp(), setuid(), sigaction(), sigaddset(), sigdelset(), sigfillset(), sigismember(), signal(), sigpending(), sigprocmask(), sigsuspend(), sleep(), statx(), tcdrain(), tcflow(), tcflush(), tcgetattr(), tcgetprgp(), tcsendbreak(), tcsetattr(), tcsetpgrp(), time(), times(), umask(), uname(), unlink(), ustat(), utime(), wait2(), wait(), write().

All other subroutines should not be called from signal–catching functions since their behavior is undefined.

Parameters

Signal Defines the signal.

Action Points to a **sigaction** structure that describes the action to be taken upon receipt of the *Signal* parameter signal.

OAction Points to a **sigaction** structure in which the signal action data in effect at the time of the **sigaction** call is returned.

Invec Points to a **sigvec** structure that describes the action to be taken upon receipt of the *Signal* parameter signal.

Outvec Points to a **sigvec** structure in which the signal action data in effect at the time of the **sigvec** call is returned.

Action Specifies the action associated with a signal.

Return Values

Upon successful completion, the **sigaction** 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 **sigaction** subroutine fails and no new signal handler is installed if one of the following occurs:

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

EINVAL The *Signal* parameter is not a valid signal number.

EINVAL An attempt was made to ignore or supply a handler for the **SIGKILL**, **SIGSTOP**, and **SIGCONT** signals.

Implementation Specifics

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

The **sigvec** and **signal** subroutines are provided for compatibility to older UNIX and AIX systems; their function is a subset of that available with **sigaction**.

sigvec uses the **sigvec** structure is used instead of the **sigaction** structure. The **sigvec** structure specifies a mask as an *int* instead of a *sigset_t*. The mask for **sigvec** is constructed by setting the *i–th* bit in the mask if signal *i* is to be blocked. Therefore, **sigvec** only allows signals of value 1–31 to be blocked when a signal–handling function is called. The other signals will not be blocked by the signal–handler mask.

The **sigvec** structure has the following members:

```
int *sv_handler()();      /* signal handler */
int  sv_mask;                     /* signal mask */
int  sv_flags;            /* flags */
```

sigvec() in **libbsd.a** interprets the **SV_INTERRUPT** flag and inverts it to the **SA_RESTART** flag of the **sigaction()**. **sigvec()** in **libc.a** always sets the **SV_INTERRUPT** flag regardless of what was passed in the **sigvec** structure.

The **signal()** in **libc.a** allows an action to be associated with a signal. The *Action* parameter can have the same values that are described for the sv_handler field in the **sigaction** structure of the **sigaction** subroutine. However, no signal handler mask or flags can be specified; the **signal** subroutine implicitly sets the signal handler mask to additional signals, and the flags to be **SA_OLDSTYLE**.

Upon successful completion of a **signal** call, the value of the previous signal action is returned. If the call fails, a value of –1 is returned and the global variable **errno** is set to indicate the error as in the **sigaction** call.

The **signal()** in **libc.a** does not set **SA_RESTART**. It sets the signal mask to the signal whose action is being specified, and sets flags to **SA_OLDSTYLE**. The BSD version of **signal()** sets **SA_RESTART** and preserves the current settings of the signal mask and flags. The BSD version can be used by compiling with the Berkeley Compatibility Library (**libbsd.a**).

Related Information

The **acct** subroutine, **_exit**, **exit**, **atexit** subroutines, **kill** subroutine, **longjmp** subroutine, **pause** subroutine, **ptrace** subroutine, **setjmp** subroutine, **sigpause**, **sigsuspend** subroutines, **sigstack** subroutine, **sigprocmask**, **sigsetmask**, **sigblock** subroutines, **umask** subroutine, **wait**, **waitpid**, **wait3** subroutines.

The **kill** command.

The **core** file.

sigemptyset, sigfillset, sigaddset, sigdelset or sigismember Subroutine

Purpose

Creates and manipulates signal masks.

Library

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

Syntax

#include <signal.h>

int sigemptyset (*Set***)**
sigset_t **Set;*

int sigfillset (*Set***)**
sigset_t **Set;*

int sigaddset (*Set, SignalNumber***)**
sigset_t **Set;*
int *SignalNumber;*

int sigdelset (*Set, SignalNumber***)**
sigset_t **Set;*
int *SignalNumber;*

int sigismember (*Set, SignalNumber***)**
sigset_t **Set;*
int *SignalNumber;*

Description

The **sigemptyset, sigfillset, sigaddset, sigdelset,** and **sigismember** subroutines manipulate sets of signals. These functions operate on data objects addressable by the application, not on any set of signals known to the system, such as the set blocked from delivery to a process or the set pending for a process.

The **sigemptyset** subroutine initializes the signal set pointed to by the parameter *Set* such that all signals are excluded. The **sigfillset** subroutine initializes the signal set pointed to by the *Set* parameter such that all signals are included. A call to either the **sigfillset** or **sigemptyset** subroutine must be made at least once for each object of the type **sigset_t** prior to any other use of that object.

The **sigaddset** and **sigdelset** subroutines respectively add and delete the individual signal specified by the *SignalNumber* parameter from the signal set specified by the *Set* parameter. The **sigismember** subroutine tests whether the *SignalNumber* parameter is a member of the signal set pointed to by the *Set* parameter.

Parameters

Set	Specifies the signal set.
SignalNumber	Specifies the individual signal.

Example

To generate and use a signal mask that blocks only the **SIGINT** signal from delivery, enter:

```
#include <signal.h>

int return_value;
sigset_t newset;
sigset_t *newset_p;
 . . .
newset_p = &newset;
sigemptyset(newset);
sigaddset(newset, SIGINT);
return_value = sigprocmask (SIG_SETMASK, newset_p, NULL);
```

Return Values

Upon successful completion, the **sigismember** subroutine returns a value of one if the specified signal is a member of the specified set, or the value of 0 if not. Upon successful completion, the other subroutines return a value of 0. For all the preceding subroutines, if an error is detected, a value of –1 is returned and the global variable **errno** is set to indicate the error.

Error Code

The **sigfillset**, **sigdelset**, **sigismember**, and **sigaddset** subroutines fail if the following is true:

EINVAL The value of the *SignalNumber* parameter is not a valid signal number.

Implementation Specifics

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

Related Information

The **sigprocmask** subroutine, **sigsuspend** subroutine. **sigaction**, **signal**, **sigvec** subroutines.

siginterrupt Subroutine

Purpose

Sets restart behavior with respect to signals and subroutines.

Library

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

Syntax

siginterrupt(*Signal*, *Flag***);**
int *Signal, Flag*;

Description

The **siginterrupt** subroutine is used to change the subroutine restart behavior when a subroutine is interrupted by the specified signal. If the flag is true (1), subroutines are restarted if they are interrupted by the specified signal and no data has been transferred yet.

If the flag is false (0), the restarting of subroutines is disabled. If a subroutine is interrupted by the specified signal and no data has been transferred, the subroutine will return a value of –1 with the global variable **errno** set to **EINTR**. Interrupted subroutines that have started transferring data will return the amount of data actually transferred. Subroutine interrupt is the signal behavior found on 4.1 BSD and AT&T System V UNIX systems.

Note that the new 4.2 BSD signal handling semantics are not altered in any other way. Most notably, signal handlers always remain installed until explicitly changed by a subsequent **sigaction** or **sigvec** call, and the signal mask operates as documented in the **sigaction** subroutine. Programs can switch between restartable and interruptible subroutine operation as often as desired in the execution of a program.

Issuing a **siginterrupt** call during the execution of a signal handler causes the new action to take place on the next signal to be caught.

Restart will not occur unless it is explicitly specified with the **sigaction** subroutine or the **sigvec** subroutine in **libc.a**.

Parameters

Signal	Indicates the signal.
Flag	Indicates true or false.

Return Values

A value of 0 indicates that the call succeeded. A value of –1 indicates that the supplied signal number is not valid.

Implementation Specifics

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

This subroutine uses an extension of the **sigvec** subroutine that is not available in the 4.2 BSD; hence it should not be used if backward compatibility is needed.

Related Information

The **sigaction**, **sigvec** subroutines, **sigpause** subroutine, **sigsetmask, sigblock** subroutines.

sigpending Subroutine

Purpose

Returns the set of signals that are blocked from delivery.

Library

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

Syntax

#include <signal.h>

int sigpending (*Set***)**
**sigset_t ***Set*;

Description

The **sigpending** subroutine stores the set of signals that are blocked from delivery and pending for the calling process, in the space pointed to by the argument *Set*.

Parameter

Set	Specifies the set of signals.

Return Values

Upon successful completion, the **sigpending** 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 Code

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

EINVAL	The input parameter is outside the user's address space.

Implementation Specifics

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

Related Information

The **sigprocmask** subroutine.

sigprocmask, sigsetmask, or sigblock Subroutine

Purpose

Sets the current signal mask.

Library

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

Syntax

int sigprocmask(*How*, *Set*, *OSet***)**
int *How*;
sigset_t **Set*, **OSet*;

int sigsetmask (*SignalMask*)
int *SignalMask*;

int sigblock (*SignalMask*)
int *SignalMask*;

Description

The **sigprocmask** subroutine is used to examine or change the calling process signal mask.

Typically, you would use the **sigprocmask(SIG_BLOCK)** subroutine to block signals during a critical section of code, and then use the **sigprocmask(SIG_SETMASK)** subroutine to restore the mask to the previous value returned by the **sigprocmask(SIG_BLOCK)** subroutine.

If there are any pending unblocked signals after the call to the **sigprocmask** subroutine, at least one of those signals will be delivered before the **sigprocmask** subroutine returns.

The **sigprocmask** subroutine does not allow the **SIGKILL** or **SIGSTOP** signals to be blocked. If a program attempts to block one of these signals, the **sigprocmask** subroutine gives no indication of the error.

Parameters

How Indicates the manner in which the set is changed; it can have one of the following values:

SIG_BLOCK The resulting set is the union of the current set and the signal set pointed to by the *Set* parameter.

SIG_UNBLOCK The resulting set is the intersection of the current set and the complement of the signal set pointed to by the *Set* parameter.

SIG_SETMASK The resulting set is the signal set pointed to by the *Set* parameter.

Set Specifies the signal set. If the value of the *Set* parameter is not null, it points to a set of signals to be used to change the currently blocked set. If the value of the *Set* parameter is null, the value of the *How* parameter is not significant and the process signal mask is unchanged; thus, the call can be used to inquire about currently blocked signals.

OSet If the *OSet* parameter is not the **NULL** value, the signal mask in effect at the time of the call is stored in the spaced pointed to by the *OSet* parameter.

SignalMask Specifies the signal mask of the process.

Compatibility Interfaces

The **sigsetmask** subroutine allows changing the process signal mask for signal values 1 to 31. This same function can be accomplished for all values with the **sigprocmask(SIG_SETMASK)** subroutine. The signal of value *i* will be blocked if the *i*-th bit of *SignalMask* parameter is set.

Upon successful completion, the **sigsetmask** subroutine returns the value of the previous signal mask. If the subroutine fails, a value of –1 is returned and the global variable **errno** is set to indicate the error as in the **sigprocmask** subroutine.

The **sigblock** subroutine allows signals with values 1 to 31 to be ORed into the current process signal mask. This same function can be accomplished for all values with the **sigprocmask(SIG_BLOCK)** subroutine. The signal of value *i* will be blocked, in addition to those currently blocked, if the *i*-th bit of the *SignalMask* parameter is set.

It is not possible to block **SIGKILL** or **SIGSTOP** signals using **sigblock** or **sigsetmask** subroutines. This restriction is silently imposed by the system without causing an error to be indicated.

Upon successful completion, the **sigblock** subroutine returns the value of the previous signal mask. If the subroutine fails, a value of –1 is returned and the global variable **errno** is set to indicate the error as in the **sigprocmask** subroutine.

Return Values

Upon completion, a value of 0 is returned. If the **sigprocmask** subroutine fails, the signal mask of the process is unchanged, a value of –1 is returned, and the global variable **errno** is set to indicate the error.

Error Code

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

EINVAL The value of the *How* parameter is not equal to one of the defined values.

Example

To set the signal mask to block only the **SIGINT** signal from delivery, enter:

```
#include <signal.h>

int return_value;
sigset_t newset;
sigset_t *newset_p;
 . . .
newset_p = &newset;
sigemptyset(newset_p);
sigaddset(newset_p, SIGINT);
return_value = sigprocmask (SIG_SETMASK, newset_p, NULL);
```

Implementation Specifics

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

Related Information

The **kill, killpg** subroutines, **sigaction**, **sigvec, signal** subroutine, **sigsuspend** subroutine.

The **sigpause** subroutine.

sigset, sighold, sigrelse, or sigignore Subroutine

Purpose

Enhance the signal facility and provide signal management.

Library

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

Syntax

#include<signal.h>

void (*sigset(*Signal*,*Function***))()**
int *Signal*;
void (**Function***)();**

int sighold(*Signal***)**
int *Signal*;

int sigrelse(*Signal***)**
int *Signal*;

int sigignore(*Signal***)**
int *Signal*;

Description

The **sigset**, **sighold**, **sigrelse**, and **sigignore** subroutines enhance the signal facility and provide signal management for application processes.

The **sigset** subroutine specifes the system signal action to be taken upon receipt of *Signal*.

The **sighld** and **sigrelse** subroutines establish critical regions of code. A call to the **sighold** subroutine is analogous to raising the priority level and deferring or holding a signal until the priority is lowered by **sigrelse**. The **sigrelse** subroutine restores the system signal action to the action that was previously specified by **sigset**.

The **sigignore** subroutine sets the action for *Signal* to **SIG_IGN**.

The other signal management routine, **signal**, should not be used in conjunction with these routines for a particular signal type.

Parameters

Signal Specifies the signal. The *Signal* parameter can be assigned any one of the following signals:

SIGHUP	Hangup
SIGINT	Interrupt
SIGQUIT	Quit*
SIGILL	Illegal instruction (not reset when caught)*
SIGTRAP	Trace trap (not reset when caught)*

SIGABRT	Abort*
SIGFPE	Floating point exception*
SIGSYS	Bad argument to routine*
SIGPIPE	Write on a pipe with no one to read it
SIGALRM	Alarm clock
SIGTERM	Software termination signal
SIGUSR1	User–defined signal 1
SIGUSR2	User–defined signal 2

* The default action for these signals is an abnotermination termination.

For portability, application programs should use or catch only the signals listed above; other signals are hardware and implementation dependent and may have very different meanings or results across systems. (For example, the System V signals **SIGEMT**, **SIGBUS**, **SIGSEGV**, and **SIGIOT** are implementation dependent and are not listed above.) Specific implementations may have other implementation dependent signals.

Function Specifies the choice. The *Function* parameter is assigned one of four values: **SIG_DFL**, **SIG_IGN**, **SIG_HOLD**, or an *address* of a signal–catching function. The *Function* parameter is declared as type pointer to a function returning void. The following actions are prescribed by these values:

SIG_DFL Terminate process upon receipt of a signal.

Upon receipt of the signal specified by the *Signal* parameter, the receiving process is to be terminated with all of the consequences outlined in **exit**. In addition, if *Signal* is one of the signals marked with an asterisk above, implementation–dependent abnormal process termination routines, such as a core dump, may be invoked.

SIG_IGN Ignore signal.

Any pending signal specified by the *Signal* parameter is discarded. A pending signal is a signal that has occurred but for which no action has been taken. The system signal action is set to ignore future occurrences of this signal type.

SIG_HOLD Hold signal.

The signal specified by the Signal parameter is to be held. Any pending signal of this type remains held. Only one signal of each type is held.

address Catch signal.

Upon receipt of the signal specified by the *Signal* parameter, the receiving process is to execute the signal catching function pointed to by *Function*. Pending signal of this type is released. This address is retained across calls to the other signal management functions, **sighold** and **sigrelse**. The signal number *Signal* will be passed as the only argument to the signal–catching function. Before entering the signal–catching function, the value of *Function* for the caught signal will be set to **SIG_HOLD**. During normal return from the signal–catching handler, the system signal action is restored to *Function* and any held signal of this type is released. If a non–local goto (see **setjmp**) is taken, the **sigrelse** subroutine must be invoked to restore the system signal action and to release any held signal of this type.

Upon return from the signal–catching function, the receiving process will resume execution at the point at which it was interrupted, except for implementation defined signals where this may not be true.

When a signal to be caught occurs during a non–atomic operation such as a call to the **read**, **write**, **open**, or **ioctl** subroutine on a slow device (such as a terminal); or occurs during a **pause** subroutine; or occurs during a wait subroutine that does not return immediately, the signal–catching function will be executed and then the interrupted routine may return a value of –1 to the calling process with **errno** set to **EINTR**.

Return Values

Upon successful completion, the **sigset** subroutine returns the previous value of the system signal action for the specified *Signal*. Otherwise, it returns **SIG_ERR** and the global variable **errno** is set to indicate the error.

For the **sighold**, **sigrelse**, and **sigignore** subroutines, a value of 0 is returned upon success. Otherwise, a value of –1 is returned and the global variable **errno** is set to indicate the error.

Error Code

The **sigset**, **sighold**, **sigrelse**, or **sigignore** subroutine fails if the following is true:

EINVAL *Signal* is an either an illegal signal number or **SIGKILL**, or the default handling of *Signal* cannot be changed.

Implementation Specifics

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

Related Information

The **kill** subroutine, **setjmp** subroutine, **signal** subroutine, **wait** subroutine.

sigsetjmp or siglongjmp Subroutine

Purpose

Saves or restores stack context and signal mask.

Library

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

Syntax

#include <setjmp.h>

int sigsetjmp (*Environment,SaveMask*)
sigjmp_buf *Environment*;
int *SaveMask*;

void siglongjmp (*Environment, Value*)
sigjmp_buf *Environment*;
int *Value*;

Description

The **sigsetjmp** subroutine saves the current stack context, and if the value of the *SaveMask* parameter is not 0, the **sigsetjmp** subroutine also saves the current signal mask of the process as part of the calling environment.

The **siglongjmp** subroutine restores the saved signal mask if and only if the *Environment* parameter was initialized by a call to the **sigsetjmp** subroutine with a non–zero *SaveMask* parameter argument.

Parameters

Environment	Specifies an address for a **sigjmp_buf** structure.
SaveMask	Specifies the flag used to determine if the signal mask is to be saved.
Value	Specifies the return value from **siglongjmp**.

Return Values

The **sigsetjmp** subroutine returns a value of 0. The **siglongjmp** subroutine returns a non–zero value.

Implementation Specifics

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

Related Information

The **setjmp**, **longjmp** subroutines, **sigaction** subroutine, **sigprocmask** subroutine, **sigsuspend** subroutine.

sigstack Subroutine

Purpose

Sets and gets signal stack context.

Library

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

Syntax

#include <signal.h>

int sigstack (*Instack, Outstack***)**
**struct sigstack ****Instack*, ******Outstack*;

Description

The **sigstack** subroutine defines an alternate stack on which signals are to be processed.

When a signal occurs and its handler is to run on the signal stack, the system checks to see if the process is already running on that stack. If so, it continues to do so even after the handler returns. If not, the signal handler runs on the signal stack, and the original stack is restored when the handler returns.

Use the **sigvec** or **sigaction** subroutine to specify whether a given signal handler routine is to run on the signal stack.

Warning: A signal stack does not automatically increase in size as a normal stack does. If the stack overflows, unpredictable results can occur.

Parameters

Instack
: Specifies the stack pointer of the new signal stack.

If the value of the *Instack* parameter is nonzero, it points to a **sigstack** structure, which has the following members:

```
caddr_t ss_sp;
int          ss_onstack;
```

The value of *Instack*–>ss_sp specifies the stack pointer of the new signal stack. Since stacks grow from numerically greater addresses to lower ones, the stack pointer passed to the **sigstack** subroutine should point to the numerically high end of the stack area to be used. *Instack–>ss_onstack* should be set to a value of 1 if the process is currently running on that stack; otherwise, it should be a value of 0.

If the value of the *Instack* parameter is 0 (that is, a **NULL** pointer), the signal stack state is not set.

Outstack
: Points to structure where current signal stack state is stored.

If the value of the *Outstack* parameter is nonzero, it points to a **sigstack** structure into which the **sigstack** subroutine stores the current signal stack state.

If the value of the *Outstack* parameter is 0, the previous signal stack state is not reported.

Return Values

Upon successful completion, the **sigstack** 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 Code

The **sigstack** subroutine fails and the signal stack context remains unchanged if the following is true:

EFAULT The *Instack* or *Outstack* parameter points outside of the address space of the process.

Implementation Specifics

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

Related Information

The **sigaction**, **signal, sigvec** subroutines, **setjmp** subroutine, **longjmp** subroutine.

sigsuspend or sigpause Subroutine

Purpose

Atomically changes the set of blocked signals and waits for a signal.

Library

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

Syntax

#include <signal.h>

int sigsuspend (*SignalMask***)**
**sigset_t ***SignalMask*;

int sigpause (*SignalMask***)**
int *SignalMask*;

Description

The **sigsuspend** subroutine replaces the signal mask of the process with the set of signals pointed to by the *SignalMask* parameter, and then suspends execution of the process until delivery of a signal whose action is either to execute a signal–catching function or to terminate the process. The **sigsuspend** subroutine does not allow the **SIGKILL** or **SIGSTOP** signals to be blocked. If a program attempts to block one of these signals, the **sigsuspend** subroutine gives no indication of the error.

If delivery of a signal causes the process to end, the **sigsuspend** subroutine does not return. If delivery of a signal causes a signal–catching function to execute, the **sigsuspend** subroutine returns after the signal–catching function returns, with the signal mask restored to the set that existed prior to the **sigsuspend** subroutine.

The **sigsuspend** subroutine sets the signal mask and waits for an unblocked signal as one atomic operation. This means that signals cannot occur between the operations of setting the mask and waiting for a signal. If a program invokes **sigprocmask(SIG_SETMASK)** and **pause** separately, a signal that occurs between these subroutines might not be noticed by **pause**.

In normal usage, a signal is blocked by using the **sigprocmask(SIG_BLOCK,...)** subroutine at the beginning of a critical section. The process then determines whether there is work for it to do. If no work is to be done, the process waits for work by calling **sigsuspend** with the mask previously returned by the **sigprocmask** subroutine.

The **sigpause** subroutine call uses **sigsuspend** to block the signals specified by the *SignalMask* parameter, and then suspends execution of the process until delivery of a signal whose action is either to execute a signal–catching function or to end the process. Signal of value *i* is blocked if the *i*–th bit of the mask is set. Only signals with values 1 to 31 can be blocked with the **sigpause** subroutine.

Parameter

SignalMask	Points to a set of signals.

Return Values

If a signal is caught by the calling process and control is returned from the signal handler, the calling process resumes execution after **sigsuspend** or **sigpause**, which always return a value of –1 and set the global variable **errno** to **EINTR.**

Implementation Specifics

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

The **sigpause** subroutine is provided for compatibility with older UNIX systems; its function is a subset of the **sigsuspend** subroutine.

Related Information

The **pause** subroutine, **sigprocmask** subroutine, **sigaction**, **sigvec**, **signal** subroutine.

The **sigsetmask**, **sigblock** subroutines.

sin, cos, tan, asin, acos, atan, or atan2 Subroutine

Purpose

Computes the trigonometric and inverse trigonometric functions.

Library

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

Syntax

#include <math.h>

double sin (*x*)
double *x*;

double cos (*x*)
double *x*;

double tan (*x*)
double *x*;

double asin (*x*)
double *x*;

double acos (*x*)
double *x*;

double atan (*x*)
double *x*;

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

Description

The **sin** subroutine, **cos** subroutine, and **tan** subroutine return the sine, cosine, and tangent, respectively, of their parameters, which are in radians.

The **asin** subroutine returns the principal value of the arc sine of *x,* in the range [–pi/2, pi/2].

The **acos** subroutine returns the principal value of the arc cosine of *x,* in the range [0, pi].

The **atan** subroutine returns the principal value of the arc tangent of *x,* in the range [–pi/2, pi/2].

The **atan2** subroutine returns 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 some double-precision floating-point value.

y Specifies some double-precision floating-point value.

Error Codes

When using **libm.a** (**–lm**):

asin, acos Return a NaNQ and set **errno** 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.

The **sin**, **cos**, and **tan** subroutines lose accuracy when passed a large value for the x parameter. In the **sin** subroutine, for example, values of *x* that are greater than pi are argument-reduced by first dividing them by the machine value for 2 * pi , and then using the IEEE remainder of this division in place of *x*. Since the machine value of `pi` can only approximate the infinitely precise value of `pi`, the remainder of x/(2 * pi) becomes less accurate as *x* becomes larger. Similar loss of accuracy occurs for the **cos** and **tan** subroutines during argument reduction of large arguments.

sin, cos When the parameter *x* is extremely large, these functions return 0 when there would be a complete loss of significance. In this case, a message indicating TLOSS error is printed on the standard error output. For less extreme values causing partial loss of significance, a PLOSS error is generated but no message is printed. In both cases, **errno** is set to ERANGE.

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

Implementation Specifics

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

Related Information

The **sinh, cosh, tanh** subroutines, **matherr** subroutine.

sinh, cosh, or tanh Subroutine

Purpose

Computes hyperbolic functions.

Library

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

Syntax

#include <math.h>

double sinh (*x*)
double *x*;

double cosh (*x*)
double *x*;

double tanh (*x*)
double *x*;

Description

The **sinh** subroutine, **cosh** subroutine, and **tanh** subroutine compute the hyperbolic trigonometric functions of their parameters.

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

```
cc tanh.c –lm
```

Parameter

x Specifies some double-precision floating-point value.

Error Codes

If the correct value overflows, the **sinh** and **cosh** subroutines return a correctly signed HUGE_VAL, and the global variable **errno** is set to ERANGE.

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

Implementation Specifics

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

Related Information

The **sin, cos, tan, asin, acos, atan, atan2** subroutines, **matherr** subroutine.

sqrt or cbrt Subroutine

Purpose

Computes square root and cube root functions.

Library

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

Syntax

#include <math.h>

double sqrt (*x*)
double *x*;

double cbrt (*x*)
double *x*;

Description

The **sqrt** subroutine and **cbrt** subroutine compute the square root and cube root, respectively, of their parameters.

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

```
cc sqrt.c –lm
```

Parameter

x Specifies some double-precision floating-point value.

Return Values

The **sqrt** (–0.0) = –0.0.

Error Codes

When using **libm.a** (**–lm**):

For the **sqrt** subroutine, if the value of *x* is negative, a NaNQ is returned and **errno** is set to EDOM.

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

For **sqrt**, if the value of *x* is negative, a 0 is returned and **errno** is set to EDOM. A message indicating 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 AIX Base Operating System (BOS) Runtime.

The **cbrt** subroutine is not part of the ANSI C Library.

Related Information

The **exp, expm1, log, log10, log1p, pow** subroutines.

src_err_msg Subroutine

Purpose

Retrieves an SRC error message.

Library

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

Syntax

int src_err_msg (*errno*, *ErrorText***)**
int *errno*;
char ***ErrorText*;

Description

The **src_err_msg** subroutine retrieves a System Resource Controller error message.

Parameters

errno	Specifies the SRC error code
ErrorText	Points to a character pointer to place the SRC error message.

Return Values

Upon successful completion, the **src_err_msg** subroutine returns a value of 0. Otherwise, a value of –1 is returned. An error message is not returned.

Implementation Specifics

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

Related Information

srcsbuf subroutine, **srcsrpy** subroutine, **srcsrqt** subroutine, **srcrrqs** subroutine, **srcstat** subroutine, **srcstathdr** subroutine, **srcstattxt** subroutine, **srcstop** subroutine, **srcstrt** subroutine, **addssys** subroutine, **chssys** subroutine, **delssys** subroutine, **defssys** subroutine, **getsubsvr** subroutine, **getssys** subroutine.

The System Resource Controller Overview in *General Programming Concepts.*

srcrrqs Subroutine

Purpose

Gets subsystem reply information from the SRC request received.

Library

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

Syntax

#include <spc.h>

struct srchdr *srcrrqs (*Packet***)**
**char ***Packet*;

Description

The **srcrrqs** subroutine saves the **srchdr** information that is contained in the packet the subsystem received from the System Resource Controller. The **srchdr** structure is defined in the **spc.h** header file. This routine must be called by the subsystem to complete the reception process of any packet received from the SRC. The subsystem requires this information to reply to any request that the subsystem receives from the SRC.

Note: The saved **srchdr** information is over-written the next time this subroutine is called.

Parameter

Packet — Points to the SRC request packet received by the subsystem. If the subsystem received the packet on a message queue, the *Packet* parameter must point past the message type of the packet to the start of the request information. If the subsystem received the information on a socket, the *Packet* parameter points to the start of packet received on the socket.

Return Value

The **srcrrqs** subroutine returns a pointer to the static **srchdr** structure that contains the return address for the subsystem response.

Example

```
int rc;
struct sockaddr addr;
int addrsz;
struct srcreq packet;

/* wait to receive packet from SRC daemon */
rc=recvfrom(0, &packet, sizeof(packet), 0, &addr, &addrsz);
/* grab the reply information from the SRC packet */
if (rc>0)
   srchdr=srcrrqs (&packet);
```

Implementation Specifics

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

Related Information

srcsbuf subroutine, **srcsrpy** subroutine, **srcsrqt** subroutine, **srcstat** subroutine, **srcstathdr** subroutine, **srcstattxt** subroutine, **srcstop** subroutine, **srcstrt** subroutine.

The System Resource Controller Overview in *General Programming Concepts.*

srcsbuf Subroutine

Purpose

Gets status for a subserver or a subsystem and returns status text to be printed.

Library

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

Syntax

#include <sys/spc.h>

int srcsbuf(*Host, Type, SubsystemName, SubserverObject, SubsystemPID, StatusType, StatusFrom, StatusText, Continued***)**
char **Host*;
short *Type*;
char **SubsystemName*;
char **SubserverObject*;
int *SubsystemPID*;
short *StatusType*;
int *StatusFrom*;
char ***StatusText*;
int ** Continued*;

Description

The **srcsbuf** subroutine gets the status of a subserver or subsystem and returns printable text for the status in the address pointed to by the *StatusText* parameter.

When the *StatusType* parameter is **SHORTSTAT** and the *Type* parameter is **SUBSYSTEM**, the **srcstat** subroutine is called to get the status of one or more subsystems. When the *StatusType* parameter is **LONGSTAT** and the *Type* parameter is **SUBSYSTEM**, the **srcrsqt** subroutine is called to get the long status of one subsystem. When the *Type* parameter is not **SUBSYSTEM**, the **srcsrqt** subroutine is called to get the long or short status of a subserver.

Parameters

Host — Specifies the foreign host on which this status action is requested. If the host is null, the status request is sent to the System Resource Controller on the local host.

Type — Specifies whether the status request applies to the subsystem or subserver. If the *Type* parameter is set to **SUBSYSTEM**, the status request is for a subsystem. If not, the status request is for a subserver and the *Type* parameter is a subserver code point.

SubsystemName — Specifies the name of the subsystem on which to get status. To get the status of all subsystems, use the constant **SRCALLSUBSYS**. To get the status of a group of subsystems, the *SubsystemName* parameter must start with the constant **SRCGROUP**, followed by the name of the group for which you want status appended. If you specify a null *SubsystemName* parameter, you must specify a *SubsystemPID* parameter.

SubserverObject Specifies a subserver object. The *SubserverObject* parameter modifies the *Type* parameter. The *SubserverObject* parameter is ignored if the *Type* parameter is set to **SUBSYSTEM**. The use of the *SubserverObject* parameter is determined by the subsystem and the caller. This parameter will be placed in the objname field of the **subreq** structure that is passed to the subsystem.

SubsystemPID Specifies the PID of the subsystem on which to get status, as returned by the **srcstrt** subroutine. You must specify the *SubsystemPID* parameter if multiple instances of the subsystem are active and you request a long subsystem status or subserver status. If you specify a null *SubsystemPID* parameter, you must specify a *SubsystemName* parameter.

StatusType Specifies **LONGSTAT** for long status or **SHORTSTAT** for short status.

StatusFrom Specifies whether status errors and messages are to be printed to standard output or just returned to the caller. When the *StatusFrom* parameter is **SSHELL**, the errors are printed to standard output.

StatusText Allocates memory for the printable text and sets *StatusText* to point to this memory. It is the responsibility of the calling process to free the memory allocated for this buffer after the calling process prints the text.

Continued Specifies whether this call to the **srcsbuf** subroutine is a continuation of a status request. If the *Continued* parameter is set to **NEWREQUEST**, a request for status is sent and the **srcsbuf** subroutine then waits for another. On return from the **srcsbuf** subroutine is updated to the new continuation indicator from the reply packet. On return, the *Continued* parameter will be sent to **END** or **STATCONTINUED** by the subsystem. If the *Continued* parameter is set to something other than **END**, this field must remain equal to that value; otherwise, this function will not be able to receive any more packets for the original status request. The calling process should not set the value of the *Continued* parameter to a value other than **NEWREQUEST**. *Continued* should not be changed while more responses are expected.

Return Value

If the **srcsbuf** subroutine succeeds, it returns the size (in bytes) of printable text pointed to by the *StatusText* parameter.

Error Codes

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

SRC_BADSOCK The request could not be passed to the subsystem because of some socket failure.

SRC_CONT The subsystem uses signals. The request cannot complete.

SRC_DMNA The SRC daemon is not active.

SRC_INET_AUTHORIZED_HOST	The local host is not in the remote **/etc/hosts.equiv** file.
SRC_INET_INVALID_HOST	On the remote host, the local host is not known.
SRC_INVALID_USER	The user is not root or group system.
SRC_MMRY	An SRC component could not allocate the memory it needs.
SRC_NOCONTINUE	*Continued* was not set to **NEWREQUEST** and no continuation is currently active.
SRC_NORPLY	The request timed out waiting for a response.
SRC_NSVR	The subsystem is not active.
SRC_SOCK	There is a problem with SRC socket communications.
SRC_STPG	The request was not passed to the subsystem. The subsystem is stopping.
SRC_UDP	The SRC port is not defined in the **/etc/services** file.
SRC_UHOST	The foreign host is not known.
SRC_WICH	There are multiple instances of the subsystem active.

Examples

1. To get the status of a subsystem:

```
char *status;
int continued=NEWREQUEST;
int rc;

do {
  rc=srcsbuf("MaryC", SUBSYSTEM, "srctest", "", 0,
     SHORTSTAT, SSHELL, &status, continued);
   if (status!=0)
    {
        printf(status);
        free(status);
        status=0;
    }
}  while (rc>0);
```

   This gets short status of the `srctest` subsystem on the `MaryC` machine and prints the formatted status to standard output.

2. To get the status of a subserver:

```
char *status;
int continued=NEWREQUEST;
int rc;

do {
  rc=srcsbuf("", 12345, "srctest", "", 0,
     LONGSTAT, SSHELL, &status, continued);
   if (status!=0)
    {
        printf(status);
        free(status);
        status=0;
    }
}  while (rc>0);
```

This gets long status of the `tester` subserver on the local machine and prints the formatted status to standard output.

Implementation Specifics

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

File

/etc/services Defines sockets and protocols used for Internet services.

Related Information

srcrrqs subroutine, **srcsrpy** subroutine, **srcsrqt** subroutine, **srcstat** subroutine, **srcstathdr** subroutine, **srcstattxt** subroutine, **srcstop** subroutine, **srcstrt** subroutine.

The System Resource Controller Overview in *General Programming Concepts*.

srcsrpy Subroutine

Purpose

Sends a reply to a request from the SRC back to the client process.

Library

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

Syntax

#include <spc.h>

int srcsrpy (*SRChdr,PPacket,PPacketSize,Continued*)
struct srchdr **SRChdr*;
char **PPacket*;
int *PPacketSize*;
ushort *Continued*;

Description

The **srcsrpy** subroutine returns a subsystem reply to an SRC subsystem request. The format and content of the reply are determined by the subsystem and the requester but must start with a **srchdr** structure. The subsystem must reply with an already defined specific format and content for the following requests: **START**, **STOP**, **STATUS**, **REFRESH**, and **TRACE**. The **START**, **STOP**, **REFRESH**, and **TRACE** requests must be answered with a **srcrep** structure. The **STATUS** request must be answered with a reply in the form of a **statbuf** structure.

Note: The **srcsrpy** subroutine creates its own socket to send the subsystem reply packets.

Sending a Subsystem Reply Packet

Your subsystem should use the **srcsrpy** subroutine to return a packet to the requester. The packet that your system sends should be in the form of a **srcrep** structure as defined in the **spc.h** header file. The **svrreply** structure that is part of the **srcrep** structure contains the subsystem reply. The **svrreply** structure immediately follows the **srchdr** structure in memory.

```
struct srcrep
{
   struct srchdr
   struct srvreply
};
```

```
struct svrreply
{
    short rtncode;
    short objtype;
    char objtext [65];
    char objname [30];
    char rtnmsg[256];
};
```

Fill in the rtncode field with the SRC error code that applies. Use **SRC_SUBMSG** as rtncode to return a subsystem–specific NLS message.

Fill in the objtype field with **SUBSYSTEM** to indicate that this reply is for a subsystem or subserver code point to indicate that this is a subserver.

Fill in the objname field with the subsystem name, subserver type, or subserver object to which this reply applies.

Fill in the rtnmsg field with a subsystem–specific NLS message.

The last packet from the subsystem must always have **END** specified in the *Continued* parameter to the **srcsrpy** subroutine.

When responding from the subsystem, there are two types of continuation packets. The first type of continuation packet is an informative message. This packet is not passed back to the client but is simply printed to the client's standard output. This message must be NLS text with message tokens filled in by the sending subsystem. To send this type of continuation message, specify **CONTINUED** in the *Continued* parameter to the **srcsrpy** subroutine. The **STOP** subsystem action does not allow any continuation; all other action requests received by the subsystem from the SRC can be sent this type of reply message.

The second type of continuation packet is a reply packet and is passed back to the client for the client to process. This type of continuation must be agreed upon by the subsystem and the requester. Status requests sent to the subsystem use the second type of continuation. To respond to subsystem status, specify **STATCONTINUED** in the *Continued* parameter to the **srcsrpy** subroutine. For the status packet to be passed back to the client, the subsystem must return packets with **STATCONTINUED** as the *Continued* parameter to the **srcsrpy** subroutine. After all status or all subsystem-defined request reply packets are sent, an end packet must be sent. The end packet is passed back to the client.

Sending Error Packets

When returning an SRC error, the reply packet should be the **srcrep** structure with svrreply.objname filled in with the subsystem name, the subserver type, or the subserver object in error. You may send a short int as a reply packet. Your subsystem can only return a short as a packet when you are returning an SRC error number with an NLS message that does not include any tokens.

When returning a non–SRC error, the reply packet should be the **srcrep** structure, with svrreply.rtncode set to the constant **SRC_SUBMSG** and svrreply.rtnmsg set to a subsystem specific NLS message. The rtnmsg field is printed on the client's standard output.

Sending Subsystem Status

To return status from the subsystem (short or long), allocate an array of **statcode** structures plus a **srchdr** structure. The **srchdr** structure must start the buffer that you are sending in response to the status request. The **statcode** structure is defined in the **spc.h** header file.

```
struct statcode
{
    short objtype;
    short status;
    char objtext[65];
    char objname[30];
};
```

Fill in the statcode.objtype field with the constant **SUBSYSTEM** to indicate that this is status for a subsystem, or with a subserver code point to indicate that this is the status for a subserver.

Fill the statcode.status field with one of the SRC status constants defined in the **spc.h** header file.

Fill in the statcode.objtext field with the NLS text that you wish displayed as status.

Fill in the statcode.objname field with the name of the subsystem or subserver for which the objtext applies.

Note: The subsystem and the requester can agree to send other subsystem–defined information back to the requester.

Parameters

SRChdr Points to the reply address buffer as returned by the **srcrrqs** subroutine.

PPacket Points to the reply packet. The first element of the reply packet is a **srchdr** structure, the **cont** element of the *PPacket*->**srchdr** structure is modified on returning from the **srcsrpy** subroutine. The second element of the reply packet should be a **svrreply** structure, an array of **statcode** structures, or another format upon which the subsystem and the requester have agreed.

PPacketSize Specifies the number of bytes in the reply packet pointed to by the *PPacket* parameter. The *PPacketSize* parameter may be the size of a short, or it may be between the size of a **srchdr** structure and **SRCPKTMAX**, which is defined in the **spc.h** file.

Continued Indicates whether this reply is to be continued. If the *Continued* parameter is set to the constant **END**, no more reply packets are sent for this request. If the *Continued* parameter is set to **CONTINUED**, the second element of what is indicated by the *PPacket* parameter must be a **svrreply** structure, since the **rtnmsg** element of the **svrreply** structure is printed to standard output. For a status reply, the *Continued* parameter is set to **STATCONTINUED**, and the second element of what is pointed to by the *PPacket* parameter must be an array of **statcode** structures. If a **STOP** subsystem request is received, only one reply packet can be sent and the *Continued* parameter must be set to **END**. Other continuation, as determined by the subsystem and the requester, must be defined using positive values for the *Continued* parameter other than the following:

0	**END**
1	**CONTINUED**
2	**STATCONTINUED**

Return Value

If the **srcsrpy** subroutine succeeds, it returns the value **SRC_OK**.

Error Code

The **srcsrpy** subroutine fails if one or both of the following are true:

SRC_SOCK There is a problem with SRC socket communications.

SRC_REPLYSZ SRC reply size is invalid.

Examples

1. To send a **STOP** subsystem reply:

```
struct srcrep return_packet;
struct srchdr *srchdr;

bzero(&return_packet,sizeof(return_packet));
return_packet.svrreply.rtncode=SRC_OK;
strcpy(return_packet.svrreply,"srctest");

srcsrpy(srchdr,return_packet,sizeof(return_packet),END);
```

   This sends a message that the subsystem `srctest` is stopping successfully.

2. To send a **START** subserver reply:

```
struct srcrep return_packet;
struct srchdr *srchdr;

bzero(&return_packet,sizeof(return_packet));
return_packet.svrreply.rtncode=SRC_SUBMSG;
strcpy(return_packet.svrreply,objname,"mysubserver");
strcpy(return_packet.svrreply,objtext,"The subserver, mysubserver,
has been started");

srcsrpy(srchdr,return_packet,sizeof(return_packet),END);
```

   This sends a message that the start subserver request was successful.

3. To send a status reply:

```
int rc;
struct sockaddr addr;
int addrsz;
struct srcreq packed;
struct
{
   struct srchdr srchdr;
   struct statcode statcode[10];
} status;
struct srchdr *srchdr;
struct srcreq packet;
        .
        .
        .
/* grab the reply information fron the SRC packet */
srchdr=srcrrqs(&packet);
bzero(&status.statcode[0].objname,

/* get SRC status header */
srcstathdr(status.statcode[0].objname,
  status.statcode[0].objtext);
        .
        .
        .
/* send status packet(s) */
srcsrpy(srchdr,&status,sizeof(status),STATCONTINUED);
        .
        .
        .
srcsrpy(srchdr,&status,sizeof(status),STATCONTINUED);

/* send final packet */
srcsrpy(srchdr,&status,sizeof(struct srchdr),END);
```

This sends several status packets.

Implementation Specifics

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

Related Information

srcrrqs subroutine, **srcsbuf** subroutine, **srcsrqt** subroutine, **srcstat** subroutine, **srcstathdr** subroutine, **srcstattxt** subroutine, **srcstop** subroutine, **srcstrt** subroutine.

The System Resource Controller Overview in *General Programming Concepts*.

srcsrqt Subroutine

Purpose

Sends a request to a subsystem.

Library

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

Syntax

#include <sys/spc.h>

srcsrqt(*Host,SubsystemName,SubsystemPID,RequestLength,*
SubsystemRequest,ReplyLength,ReplyBuffer,StartItAlso,Continued **)**

char **Host*;
char **SubsystemName*;
int *SubsystemPID,*;
short *RequestLength*;
char ** SubsystemRequest*;
short **ReplyLength*;
char **ReplyBuffer*;
int *StartItAlso*;
int ** Continued*;

Description

The **srcsrqt** subroutine sends a request to a subsystem and returns one or more replies to the caller. The format of the request and the reply is determined by the caller and the subsystem.

Note: The **srcsrqt** subroutine creates its own socket to send a request to the subsystem. The socket that this function opens remains open until an error or an end packet is received.

Two types of continuation are returned by the **srcsrqt** subroutine:

No continuation	*ReplyBuffer*–>srchdr.continued is set to the constant **END**.
Reply continuation	*ReplyBuffer*–>srchdr.continued is not set to the constant **END** but to an agreed upon positive value between the calling process and the subsystem, and the packet is returned to the caller.

Parameters

SubsystemPID	The process ID of the subsystem.
Host	Specifies the foreign host on which this subsystem request is to be sent. If the host is null, the request is sent to the subsystem on the local host.
SubsystemName	Specifies the name of the subsystem to which this request is to be sent. You must specify a *SubsystemName* if you do not specify a *SubsystemPID.*
RequestLength	Specifies the length, in bytes, of the request to be sent to the subsystem.

SubsystemRequest	Points to the subsystem request packet.
ReplyLength	Specifies the maximum length, in bytes, of the reply to be received from the subsystem. On return from the **srcsrqt** subroutine, the *ReplyLength* parameter is set to the actual length of the subsystem reply packet.
ReplyBuffer	Points to a buffer for the receipt of the reply packet from the subsystem.
StartItAlso	Specifies whether the subsystem should be started if it is non-active. When nonzero, the System Resource Controller attempts to start a non-active subsystem, and then passes the request to the subsystem.
Continued	Specifies whether this call to the **srcsrqt** subroutine is a continuation of a previous request. If the *Continued* parameter is set to **NEWREQUEST**, a request for it is sent to the subsystem and waits for another response. The calling process should never set *Continued* to any value other than **NEWREQUEST.** The last response from the subsystem will set *Continued* to **END**.

Return Value

If the **srcsrqt** subroutine is successful, the value **SRC_OK** is returned.

Error Codes

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

SRC_BADSOCK	The request could not be passed to the subsystem because of a socket failure.
SRC_CONT	The subsystem uses signals. The request cannot complete.
SRC_DMNA	The SRC daemon is not active.
SRC_INET_AUTHORIZED_HOST	The local host is not in the remote **/etc/hosts.equiv** file.
SRC_INET_INVALID_HOST	On the remote host, the local host is not known.
SRC_INVALID_USER	The user is not root or group system.
SRC_MMRY	An SRC component could not allocate the memory it needs.
SRC_NOCONTINUE	*Continued* was not set to **NEWREQUEST** and no continuation is currently active
SRC_NORPLY	The request timed out waiting for a response.
SRC_NSVR	The subsystem is not active.
SRC_REQLEN2BIG	The *RequestLength* is greater than 2000 bytes. (Only 2000 bytes are allowed.)

SRC_SOCK	There is a problem with SRC socket communications.
SRC_STPG	The request was not passed to the subsystem. The subsystem is stopping.
SRC_UDP	The SRC port is not defined in the **/etc/services** file.
SRC_UHOST	The foreign host is not known.

Examples

1. To request long subsystem status:

```
int rc
short reqlen
short reqlen
struct
{
    struct srchdr srchdr;
    struct statcode statcode[20];
} statbuf;
struct subreq subreq;

subreq.action=STATUS
subreq.object=SUBSYSTEM
subreq.parm1=LONGSTAT;
strcpy(subreq.objname,"srctest");
statbuf.srchdr.cont=NEWREQUEST;
reqlen=sizeof(statbuf);
reqlen=sizeof(subreq);
rc=srcsrqt("MaryC", "srctest", 0, reqlen,
 &subreq, &reqlen, &statbuf, SRC_NO);
```

   This gets long status of the subsystem `srctest` on the `MaryC` machine. The subsystem keeps sending status packets until statbuf.srchdr.cont=END.

2. To start a subserver:

```
int rc
short reqlen
short reqlen
struct
{
    struct srchdr srchdr;
    struct statcode statcode[20];
} statbuf;
struct subreq subreq;

subreq.action=START
subreq.object=1234
statbuf.srchdr.cont=NEWREQUEST;
reqlen=sizeof(statbuf);
reqlen=sizeof(subreq);
rc=srcsrqt("", "", 987, reqlen, &subreq,
 &reqlen, &statbuf, SRC_NO);
```

   This starts the subserver with the code point of `1234`, but only if the subsystem is already active.

3. To start a subserver and a subsystem:

```
int rc
short reqlen
short reqlen
struct
{
    struct srchdr srchdr;
    struct statcode statcode[20];
} statbuf;
struct subreq subreq;

subreq.action=START
subreq.object=1234
statbuf.srchdr.cont=NEWREQUEST;
reqlen=sizeof(statbuf);
reqlen=sizeof(subreq);
rc=srcsrqt("", "", 987, reqlen, &subreq,
         &replen, &statbuf, SRC_YES);
```

This starts the subserver with the code point of `1234` and if the subsystem to which this subserver belongs is not active, the subsystem is started.

Implementation Specifics

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

File

/etc/services	Defines sockets and protocols used for Internet services.

Related Information

srcrrqs subroutine, **srcsbuf** subroutine, **srcsrpy** subroutine, **srcstat** subroutine, **srcstathdr** subroutine, **srcstattxt** subroutine, **srcstop** subroutine, **srcstrt** subroutine.

The System Resource Controller Overview in *General Programming Concepts.*

srcstat Subroutine

Purpose

Gets short status on a subsystem.

Library

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

Syntax

#include <spc.h>

int srcstat(*Host, SubsystemPID, SubsystemName, ReplyLength, StatusReply, Continued***)**
char **Host*;
char **SubsystemName*;
int *SubsystemPID*;
short **ReplyLength*;
struct statrep **StatusReply*;
int ** Continued*;

Description

The **srcstat** subroutine sends a short status request to the System Resource Controller and returns status on one or more subsystems to the caller.

Parameters

Host — Specifies the foreign host on which this status action is requested. If the host is null, the status request is sent to the System Resource Controller on the local host.

SubsystemName — Specifies the name of the subsystem on which to get short status. To get status of all subsystems, use the constant **SRCALLSUBSYS**. To get status of a group of subsystems, the *SubsystemName* parameter must start with the constant **SRCGROUP**, followed by the name of the group for which you want status appended. If you specify a null *SubsystemName,* you must specify a *SubsystemPID* parameter.

SubsystemPID — Specifies the PID of the subsystem on which to get status as returned by the **srcstat** subroutine. You must specify the *SubsystemPID* parameter if multiple instances of the subsystem are active and you request a long subsystem status or subserver status. If you specify a null *SubsystemPID* parameter, you must specify a *SubsystemName* parameter.

ReplyLength — Specifies size of a **srchdr** structure plus the number of **statcode** structures times the size of one **statcode** structure. On return from the the **srcstat** subroutine, this value is updated.

StatusReply — Specifies a pointer to an array of **statcode** structures to receive the status reply for the requested subsystem.The first element of the **statcode** array returned contains the status title line. The **statcode** structure is defined in the **spc.h** include file.

Continued	Specifies whether this call to the **srcstat** subroutine is a continuation of a previous status request. If the *Continued* parameter is set to **NEWREQUEST**, a request for short subsystem status is sent to the System Resource Controller and **srcstat** waits for the first status response. The calling process should never set *Continued* to a value other than **NEWREQUEST**. The last response for the System Resource Controller sets *Continued* to **END**.

Return Value

If the **srcstat** subroutine succeeds, it returns the size of the **statcode** buffer, which is a multiple of the **statcode** structure size.

Error Codes

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

SRC_DMNA	The SRC daemon is not active.
SRC_INET_AUTHORIZED_HOST	The local host is not in the remote **/etc/hosts.equiv** file.
SRC_INET_INVALID_HOST	On the remote host, the local host is not known.
SRC_INVALID_USER	The user is not root or group system.
SRC_MMRY	An SRC component could not allocate the memory it needs.
SRC_NOCONTINUE	*Continued* was not set to **NEWREQUEST** and no continuation is currently active.
SRC_NORPLY	The request timed out waiting for a response.
SRC_SOCK	There is a problem with SRC socket communications.
SRC_UDP	The SRC port is not defined in the **/etc/services** file.
SRC_UHOST	The foreign host is not known.

Examples

1. To request the status of a subsystem:

```
struct statcode statcode[20];
short replen=sizeof(statcode);

srcstat("MaryC","srctest",0,&replen,statcode);
```

   This requests short status of all instances of the subsystem `request` on the `MaryC` machine.

2. To request the status of all subsystems:

```
struct statcode statcode[20];
short replen=sizeof(statcode);

srcstat("",SRCALLSUBSYS,0,&replen,statcode);
```

This requests short status of all subsystems on the local machine.

3. To request the status for a group of subsystems, enter:

```
struct statcode statcode[20];
short replen=sizeof(statcode);
char subsysname[30];

strcpy(subsysname,SRCGROUP);
strcpy(subsysname,"tcpip");
srcstat("",subsysname,0,&replen,statcode);
```

This requests short status of all members of the subsystem group `tcpip` on the local machine.

Implementation Specifics

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

File

/etc/services Defines the sockets and protocols used for Internet services.

Related Information

srcrrqs subroutine, **srcsbuf** subroutine, **srcsrpy** subroutine, **srcsrqt** subroutine, **srcstathdr** subroutine, **srcstattxt** subroutine, **srcstop** subroutine, **srcstrt** subroutine.

The System Resource Controller Overview in *General Programming Concepts*.

srcstathdr Subroutine

Purpose

Gets the SRC status text title line.

Library

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

Syntax

void srcstathdr(*Title1*, *Title2***)**
char **Title1*;
char **Title2*;

Description

The **srcstathdr** subroutine returns the SRC line header for status.

Parameters

Title1 Specifies the objname field of a **statcode** structure. The subsystem name title will be placed in the *Title1* parameter.

Title2 Specifies the objtext field of a **statcode** structure. The remaining titles will be placed in the *Title2* parameter.

Return Values

The subsystem name title is returned in the *Title1* parameter. The remaining titles are returned in the *Title2* parameter.

Implementation Specifics

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

Related Information

srcrrqs subroutine, **srcsbuf** subroutine, **srcsrpy** subroutine, **srcsrqt** subroutine, **srcstat** subroutine, **srcstattxt** subroutine, **srcstop** subroutine, **srcstrt** subroutine.

The System Resource Controller Overview in *General Programming Concepts*.

srcstattxt Subroutine

Purpose

Gets the SRC status text representation for a status code.

Library

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

Syntax

char *srcstatxt (*StatusCode*)
short *StatusCode*;

Description

The **srcstattxt** subroutine, given an SRC status code, gets the text representation and returns a pointer to this text.

Parameter

StatusCode Specifies an SRC status code to be translated into meaningful text.

Return Value

The **srcstattxt** subroutine returns a pointer to the text representation of a status code.

Implementation Specifics

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

Related Information

The **srcsbuf** subroutine, **srcrrqs** subroutine, **srcsrpy** subroutine, **srcsqrt** subroutine, **srcstat** subroutine, **srcstathdr** subroutine, **srcstop** subroutine, **srcstrt** subroutine.

The System Resource Controller Overview in *General Programming Concepts.*

srcstop Subroutine

Purpose

Stops a subsystem.

Library

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

Syntax

#include <sys/spc.h>

srcstop(*Host,SubsystemName,SubsystemPID,StopType,ReplyLength,*
*ServerReply,StopFrom***)**
char **Host*;
char **SubsystemName*;
int *SubsystemPID*;
short *StopType* ;
short **ReplyLength*;
struct srcrep ** ServerReply*;
int *StopFrom*;

Description

The **srcstop** subroutine sends a stop subsystem request to a subsystem and waits for a stop reply from the SRC or the subsystem. The **srcstop** subroutine can only stop a subsystem that was started by the System Resource Controller.

Parameters

Host Specifies the foreign host on which this stop action is requested. If the host is the **NULL** value, the request is sent to the System Resource Controller on the local host.

SubsystemName Specifies the name of the subsystem to stop.

SubsystemPID Specifies the process ID of the system to stop as returned by the **srcstrt** subroutine. If you specify a null *SubsystemPID* parameter, you must specify a *SubsystemName* parameter.

StopType Specifies the type of stop requested of the subsystem. If this parameter is null, a normal stop is assumed. The *StopType* parameter must be one of the following values:

CANCEL Requires a quick stop of the subsystem. The subsystem is sent a **SIGTERM** signal, and after the wait time defined in the subsystem object, the System Resource Controller issues a **SIGKILL** to the subsystem. This waiting period allows the subsystem to clean up all its resources and terminate. The stop reply is returned by the SRC.

FORCE Requests a quick stop of the subsystem and all its subservers. The stop reply is returned by the SRC for subsystems that use signals and by the subsystem for other communication types.

NORMAL	Requests the subsystem to terminate after all current subsystem activity has completed. The stop reply is returned by the SRC for subsystems that use signals and by the subsystem for other communication types.

ReplyLength — Specifies the maximum length, in bytes, of the stop reply. On return from the **srcstop** subroutine will be set to the actual length of the subsystem reply packet received.

ServerReply — Points to an **svrreply** structure that will receive the subsystem stop reply.

StopFrom — Specifies whether the **srcstop** is to display stop results to standard output. If the *StopFrom* parameter is set to **SSHELL**, the stop results are displayed to standard output and the **srcstop** subroutine always returns successfully. If the *StopFrom* parameter is set to **SDAEMON**, the stop results are not displayed to standard output, but are passed back to the caller.

Return Values

Upon successful completion, the **srcstop** subroutine returns **SRC_OK** or **SRC_STPOK**.

Error Codes

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

SRC_BADFSIG	The stop force signal is an invalid signal.
SRC_BADNSIG	The stop normal signal is an invalid signal.
SRC_BADSOCK	The stop request could not be passed to the subsystem on its communication socket.
SRC_DMNA	The SRC daemon is not active.
SRC_INET_AUTHORIZED_HOST	The local host is not in the remote **/etc/hosts.equiv** file.
SRC_INET_INVALID_HOST	On the remote host, the local host is not known.
SRC_INVALID_USER	The user is not root or group system.
SRC_MMRY	An SRC component could not allocate the memory it needs.
SRC_NORPLY	The request timed out waiting for a response.
SRC_NOTROOT	The SRC daemon is not running as root.
SRC_SOCK	There is a problem with SRC socket communications.
SRC_STPG	The request was not passed to the subsystem. The subsystem is stopping.
SRC_SVND	The subsystem is unknown to the SRC daemon.

SRC_UDP	The remote SRC port is not defined in the **/etc/services** file.
SRC_UHOST	The foreign host is not known.
SRC_PARM	Invalid parameter passed.

Examples

1. To stop all instances of a subsystem:

```
int rc;
struct svrreply svrreply;
short replen=sizeof(svrreply);

rc=srcstop("MaryC","srctest",0,FORCE,&replen,&svrreply,SDAEMON);
```

   This will request a stop subsystem with a stop type of force for all instances of the subsystem `srctest` on the `MaryC` machine and does not print a message to standard output about the status of the stop.

2. To stop a single instance of a subsystem:

```
struct svrreply svrreply;
short replen=sizeof(svrreply);

rc=srcstop("","",999,CANCEL,&replen,&svrreply,SSHELL);
```

   This will request a stop subsystem with a stop type of cancel, with the PID of `999` on the local machine and prints a message to standard output about the status of the stop.

Implementation Specifics

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

File

/etc/services Defines sockets and protocols used for Internet services.

Related Information

The **srcrrqs** subroutine, **srcsbuf** subroutine, **srcsrpy** subroutine, **srcsrqt** subroutine, **srcstat** subroutine, **srcstathdr** subroutine, **srcstattxt** subroutine, **srcstrt** subroutine.

The System Resource Controller Overview in *General Programming Concepts*.

srcstrt Subroutine

Purpose

Starts a subsystem.

Library

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

Syntax

#include <sys/spc.h>

srcstrt(*Host,SubsystemName,Environment,Arguments,Restart,StartFrom***)**
char **Host*;
char **SubsystemName*;
char **Environment*;
char ** Arguments*;
unsigned int *Restart*;
int *StartFrom*;

Description

The **srcstrt** subroutine sends a start subsystem request packet and waits for a reply from the SRC.

Parameters

Host — Specifies the foreign host on which this start subsystem action is requested. If the host is null, the request is sent to the System Resource Controller on the local host.

SubsystemName — Specifies the name of the subsystem to start.

Environment — Specifies a string that is placed in the subsystem environment when the subsystem is executed. A maximum of 2400 characters is permitted between *Environment* and *Arguments*. The **srcstrt** subroutine fails if more than 2400 characters are specified. The environmental string is parsed by the SRC according to the same rules that are used by the shell; for example, quoted strings are assigned to a single *Environment* variable and blanks outside a quoted string delimit each environmental variable.

Arguments — Specifies a string that is passed to the subsystem when the subsystem is executed. The string is parsed from the command line and appended to the command line arguments from the subsystem object class. A maximum of 2400 characters is permitted between *Environment* and *Arguments*. The **srcstrt** subroutine fails if more than 2400 characters are specified. The command argument is parsed by the SRC according to the same rules that are used by the shell; for example, quoted strings are passed as a single argument and blanks outside a quoted string delimit arguments.

Restart	Specifies override on subsystem restart. If the *Restart* parameter is set to **SRC_NO**, the subsystem's restart definition from the subsystem object class is used. If the *Restart* parameter is set to **SRC_YES**, the restart of a subsystem is not attempted if it terminates abnormally.
StartFrom	Specifies whether the **srcstrt** subroutine is to display start results to standard output. If the *StartFrom* parameter is set to **SSHELL**, the start results are displayed to standard output, and the **srcstrt** subroutine always returns successfully. If the *StartFrom* parameter is set to **SDAEMON**, the start results are not displayed to standard output but are passed back to the caller.

Return Values

When *StartFrom* is equal to **SSHELL**, the **srcstrt** subroutine returns the value **SRC_OK**. Otherwise, it returns the subsystem PID.

Error Codes

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

SRC_AUDITID	The audit user ID is invalid.
SRC_DMNA	The SRC daemon is not active.
SRC_FEXE	The subsystem could not be **fork**ed and **exec**ed.
SRC_INET_AUTHORIZED_HOST	The local host is not in the remote **/etc/hosts.equiv** file.
SRC_INET_INVALID_HOST	On the remote host, the local host is not known.
SRC_INVALID_USER	The user is not root or group system.
SRC_INPT	The subsystem standard input file could not be established.
SRC_MMRY	An SRC component could not allocate the memory it needs.
SRC_MSGQ	The subsystem message queue could not be created.
SRC_MULT	Multiple instance of the subsystem are not allowed.
SRC_NORPLY	The request timed out waiting for a response.
SRC_OUT	The subsystem standard output file could not be established.
SRC_PIPE	A pipe could not be established for the subsystem.
SRC_SERR	The subsystem standard error file could not be established.

SRC_SUBSOCK	The subsystem communication socket could not be created.
SRC_SUBSYSID	The system user ID is invalid.
SRC_SOCK	There is a problem with SRC socket communications.
SRC_SVND	The subsystem is unknown to the SRC daemon.
SRC_UDP	The SRC port is not defined in the **/etc/services** header file.
SRC_UHOST	The foreign host is not known.

Examples

1. To start a subsystem passing the *Environment* and *Arguments* parameters:

   ```
   rc=srcstrt("","srctest","HOME=/tmp TERM=ibm6155",
   "-z \"the z flag argument\"",SRC_YES,SSHELL);
   ```

   This starts the `srctest` subsystem on the local host placing `HOME=/tmp`, `TERM=ibm6155` in the environment and `-z` and `the z flag argument` as two arguments to the subsystem. This also displays the results of the start command to standard output and allows the SRC to restart the subsystem should it end abnormally.

2. To start a subsystem on a foreign host:

   ```
   rc=srcstrt("MaryC","srctest","","",SRC_NO,SDAEMON);
   ```

   This starts the `srctest` subsystem on the `MaryC` machine. This does not display the results of the start command to standard output and does not allow the SRC to restart the subsystem should it end abnormally.

Implementation Specifics

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

File

/etc/services	Defines sockets and protocols used for Internet services.

Related Information

The **srcrrqs** subroutine, **srcsbuf** subroutine, **srcsrqt** subroutine, **srcsrpy** subroutine, **srcstat** subroutine, **srcstathdr** subroutine, **srcstattxt** subroutine, **srcstop** subroutine.

The System Resource Controller Overview in *General Programming Concepts*.

ssignal or gsignal Subroutine

Purpose

Implements a software signal facility.

Library

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

Syntax

#include <signal.h>

void (*ssignal (*Signal*, *Action***))( )**
int *Signal*;
void (**Action***)( );**

int gsignal (*Signal***)**
int *Signal*;

Description

The **ssignal** and **gsignal** subroutines implement a software facility similar to that of the **signal** subroutine and the **kill** subroutine. However, there is no connection between the two facilities. User programs can use the **ssignal** and **gsignal** subroutines to handle exceptional processing within an application. The **signal** subroutine and related subroutines handle system–defined exceptions.

The software signals available are associated with integers in the range 1 through 16. Other values are reserved for use by the C library and should not be used.

The **ssignal** subroutine associates the procedure specified by the *Action* parameter with the software signal specified by the *Signal* parameter. The **gsignal** subroutine *raises* the *Signal*, causing the procedure specified by the *Action* parameter to be taken.

The *Action* parameter is either a pointer to a user–defined subroutine, or one of the constants **SIG_DFL** (default action) and **SIG_IGN** (ignore signal). The **ssignal** subroutine returns the procedure that was previously established for that signal. If no procedure was established before, or if the signal number is illegal, then **ssginal** returns the value **SIG_DFL**.

The **gsignal** subroutine *raises* the signal specified by the *Signal* parameter by doing the following:

- If the procedure for *Signal* is **SIG_DFL**, the **gsignal** subroutine returns a value of 0 and takes no other action.
- If the procedure for *Signal* is **SIG_IGN**, the **gsignal** subroutine returns a value of 1 and takes no other action.
- If the procedure for *Signal* is a subroutine, the *Action* value is reset to **SIG_DFL** and the subroutine is called with *Signal* passed as its parameter. The **gsignal** subroutine returns a value of 2.
- If the procedure for *Signal* is illegal or if no procedure is specified for that signal, **gsignal** returns a value of 0 and takes no other action.

Parameters

Signal Specifies a signal.

Action Specifies a procedure.

Implementation Specifics

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

Related Information

The **signal** subroutine.

The **kill, killpg** system calls.

statacl or fstatacl Subroutine

Purpose

Retrieves the access control information for a file.

Library

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

Syntax

#include <sys/acl.h>
#include <sys/stat.h>

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

int fstatacl (*FileDescriptor*, *Command*, *ACL*, *ACLSize*)
int *FileDescriptor*;
int *Command*;
struct acl **ACL*;
int *ACLSize*;

Description

The **statacl** and **fstatacl** subroutines return the access control information for a file system object.

Parameters

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

FileDescriptor Specifies the file descriptor of an open file.

Command Specifies the mode of the path interpretation for *Path*, specifically whether to retrieve information about a symbolic link or mount point. Valid values for *Command* are defined in the **stat.h** file and include:

STX_LINK, **STX_MOUNT**, or **STX_NORMAL**

ACL Specifies a pointer to a buffer to contain the Access Control List of the file system object. The format of an ACL is defined in the **sys/acl.h** file and includes the following members:

acl_len the size of the Access Control List

acl_mode the file mode

u_access the access permissions for the file owner

g_access the access permissions for the file group

o_access the access permissions for default class *others*

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

The valid values for the **acl_mode** parameter are defined in **sys/mode.h**.

The fields for the base ACL, owner, group, and others, may contain the following bits which are defined in **sys/access.h**:

R_ACC Allows read permission.

W_ACC Allows write permission.

X_ACC Allows execute or search permission.

ACLSize Specifies the size of the buffer to contain the Access Control List. If this value is too small, the first word of the ACL is set to the size of the buffer needed.

Return Values

On successful completion, the **statacl** and **fstatacl** subroutines return 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 **statacl** subroutine fails 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.

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

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

ESTALE The process's 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 **fstatacl** subroutine fails if the following is true:

EBADF The file descriptor *FileDescriptor* is not valid.

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

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

EINVAL The *Command* parameter is not one of the valid values, **STX_LINK**, **STX_MOUNT**, **STX_NORMAL**.

ENOSPC The *ACLSize* parameter indicates the buffer at *ACL* is too small to hold the Access Control List. In this case, the first word of the buffer is set to the size of the buffer required.

EIO An I/O error occurred during the operation.

If NFS is installed on your system, the **statacl** and **fstatacl** subroutines can also fail if the following is true:

ETIMEDOUT The connection timed out.

Implementation Specifics

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

Related Information

The **chacl** subroutine, **statacl** subroutine, **stat** subroutine.

The **acl_get** subroutine, **acl_put** subroutine, **acl_set** subroutine, **acl_chg** subroutine.

The **acl_get** command, **acl_put** command, **chmod** command.

statfs, fstatfs, or ustat Subroutine

Purpose

Gets file system statistics.

Syntax

#include <sys/statfs.h>

int statfs(*Path*, *StatusBuffer***)**
char **Path*;
struct statfs **StatusBuffer*;

int fstatfs(*FileDescriptor*, *StatusBuffer***)**
int *FileDescriptor*;
struct statfs **StatusBuffer*;

#include <sys/types.h>
#include <ustat.h>

int ustat(*Device*, *Buffer***)**
dev_t *Device;*
struct ustat **Buffer;*

Description

The **statfs** and **fstatfs** subroutines return information about a mounted file system that contains the file described by *Path* or *FileDescriptor*. The returned information is in the format of a **statfs** structure, described in the **sys/statfs.h** header file.

The **ustat** subroutine also returns information about a mounted file system identified by *Device*. This device identifier is for any given file and can be determined by examining the **st_dev** field of the **stat** structure defined in the **sys/stat.h** header file. The returned information is in the format of a **ustat** structure, described in the **ustat.h** header file. This subroutine is superseded by **statfs** and **fstatfs**, and it is recommended that one of these latter subroutines be used instead.

Parameters

Path	The path name of any file within the mounted file system.
FileDescriptor	A file descriptor obtained by a successful **open** or **fcntl** subroutine.
StatusBuffer	A pointer to a **statfs** buffer to hold the returned information from **statfs** or **fstatfs**.
Device	The ID of the device. It corresponds to the st_rdev member of the structure returned by the **stat** subroutine. The **stat** subroutine and the **sys/stat.h** header file provide more information about the device driver.
Buffer	A pointer to a **ustat** buffer to hold the returned information.

Return Values

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

Error Codes

The **statfs**, **fstatfs**, and **ustat** subroutines fail if the following is true:

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

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

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

EIO An I/O error occurred while reading from the file system.

The **statfs** subroutine can also fail if additional errors on page A–1 occur.

Implementation Specifics

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

Related Information

The **stat** subroutine.

The **statfs.h** file, **ustat.h** file.

statx, stat, fstatx, fstat, fullstat, or ffullstat Subroutine

Purpose

Provides information about a file.

Library

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

Syntax

#include <sys/stat.h>

stat(*Path*, *Buffer* **)**
char **Path*;
struct stat **Buffer*;

lstat(*Path*, *Buffer***)**
char **Path*;
struct stat **Buffer*;

fstat(*FileDescriptor*, *Buffer***)**
int *FileDescriptor*;
struct stat **Buffer*;

int statx(*Path*, *Buffer*, *Length*, *Command***)**
char **Path*;
struct stat **Buffer*;
int *Length*;
int *Command*;

int fstatx(*FileDescriptor*, *Buffer, Length, Command***)**
int *FileDescriptor*;
struct stat **Buffer*;
int *Length*;
int *Command*;

#include <sys/fullstat.h>

fullstat(*Path*, *Command*, *Buffer***)**
struct fullstat **Buffer*;
char **Path*;
int *Command*;

ffullstat(*FileDescriptor*, *Command*, *Buffer***)**
struct fullstat **Buffer*;
int *FileDescriptor*;
int *Command*;

Description

The **stat** subroutine obtains information about the file named by *Path*. Read, write, or execute permission for the named file is not required, but all directories listed in the path name leading to the file must be searchable. The file information, which is a subset of the **stat** structure, is written to the area specified by the *Buffer* parameter.

The **lstat** subroutine is like **stat** except in the case where the named file is a symbolic link, in which case **lstat** returns information about the link, while **stat** returns information about the file the link references.

The **fstat** subroutine is like **stat** except that the information obtained is about an open file referenced by the *FileDescriptor* parameter.

The **statx** subroutine is an extension of **stat**. It can obtain a greater set of file information, and the *Path* parameter can be processed differently, depending on the contents of the *Command* parameter. The *Command* parameter provides the ability to collect information about symbolic links, as in **lstat**, as well as information about mount points and hidden directories. The **statx** subroutine returns only as much information as is specified by the *Length* parameter.

The **fstatx** subroutine is like **statx** except that the information obtained is about an open file referenced by the *FileDescriptor* parameter, as in **fstat**.

The **fullstat** and **ffullstat** subroutines are interfaces that are maintained for backward compatibility. The **fullstat** structure is identical to the **stat** structure with the exception of some field names.

Parameters

Path — The path name identifying the file. This name can be interpreted differently depending on the interface used.

FileDescriptor — The file descriptor identifying the open file.

Buffer — A pointer to the **stat** structure in which information is returned. The **stat** structure is described in the **sys/stat.h** header file.

Length — Indicates the amount of information, in bytes, to be returned. Any value between 0 and **STATXSIZE** may be specified. The following macros may be used:

- **STATSIZE** — The subset of the **stat** structure that is normally returned for a **stat** call.
- **FULLSTATSIZE** — The subset of the **stat** (**fullstat**) structure that is normally returned for a **fullstat** call.
- **STATSIZE** — The complete **stat** structure. A *Length* of 0 is equivalent to **STATXSIZE**.

Command — Specifies processing options.

For **statx**, the *Command* parameter determines how to interpret the provided path name; specifically, whether to retrieve information about a symbolic link, hidden directory or mount point. Options can be combined by logically ORing them together.

- **STX_LINK** — If the *Command* parameter specifies **STX_LINK** and the *Path* parameter is a path name that refers to a symbolic link, **statx** returns information about the symbolic link. If **STX_LINK** is not specified, **statx** returns information about the file to which the link refers.

  If *Command* specifies **STX_LINK** and *Path* refers to a symbolic link, the st_mode field of the returned **stat** structure indicates the file is a symbolic link.

STX_HIDDEN If *Command* specifies **STX_HIDDEN** and *Path* is a path name that refers to a hidden directory, **statx** returns information about the hidden directory. If **STX_HIDDEN** is not specified, **statx** returns information about a subdirectory of the hidden directory.

If *Command* specifies **STX_HIDDEN** and *Path* refers to a hidden directory, the st_mode field of the returned **stat** structure indicates this is a hidden directory.

STX_MOUNT If *Command* specifies **STX_MOUNT** and *Path* is the name of a file or directory which has been mounted over, **statx** returns information about the mounted–over file. If **STX_MOUNT** is not specified, **statx** returns information about the mounted file or directory (the root of a virtual file system).

If *Command* specifies **STX_MOUNT**, the **FS_MOUNT** bit in the st_flag field of the returned **stat** structure is set if (and only if) this file is mounted over.

If *Command* does not specify **STX_MOUNT**, the **FS_MOUNT** bit in the st_flag field of the returned **stat** structure is set if (and only if) this file is the root of a virtual file system.

STX_NORMAL If *Command* specifies **STX_NORMAL**, then no special processing is performed on the *Path*. It should be used when **STX_LINK**, **STX_HIDDEN**, and **STX_MOUNT** options are not desired.

For **fstatx**, there are currently no special processing options. The only valid value for *Command* is **STX_NORMAL**.

For **fullstat** and **ffullstat**, *Command* may be **FL_STAT**, which is equivalent to **STX_NORMAL**, or **FL_NOFOLLOW**, which is equivalent to **STX_LINK**.

Return Values

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

Error Codes

The **stat, lstat, statx**, and **fullstat** subroutines fail if one or more of the following are true:

EFAULT The file named by *Path* does not exist.

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

ENOENT The file named by *Path* does not exist.

The **stat**, **lstat**, **statx**, and **fullstat** subroutines also fail if additional errors on page A–1 occur.

The **fstat**, **fstatx**, and **ffullstat** subroutines fail if one or more of the following are true:

EBADF	*FileDescriptor* is not a valid file descriptor.
EFAULT	The *Buffer* parameter points to a location outside of the allocated address space of the process.
EIO	An I/O error occurred while reading from the file system.

The **statx** and **fstatx** subroutines fail if one or more of the following are true:

EINVAL	*Length* is not a value between 0 and **STATSIZE**.
EINVAL	An illegal value was provided for the *Command* parameter.

Implementation Specifics

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

Related Information

The **chmod** subroutine, **chown** subroutine, **link** subroutine, **mknod** subroutine, **mount** subroutine, **open** subroutine, **pipe** subroutine, **symlink** subroutine, **vtimes** subroutine.

The **stat.h** file, **fullstat.h** file, **mode.h** file.

strerror Subroutine

Purpose

Maps an error number to an error message string.

Library

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

Syntax

#include <string.h>

char *strerror(*ErrorNumber*);
int *ErrorNumber*;

Description

The **strerror** subroutine maps the error number in *ErrorNumber* to the error message string.

Parameter

ErrorNumber Specifies the error number to be associated with the error message.

Return Values

The **strerror** subroutine returns a pointer to the error message.

Implementation Specifics

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

Related Information

The **feof, ferror, clearerr, fileno** macros, **perror** subroutine.

string Subroutines

Purpose

Perform operations on strings.

Library

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

Syntax

#include <string.h>
char *strcat (*String1, String2***)**
char **String1*, **String2*;

char *strncat (*String1*, *String2*, *Number***)**
char **String1*, **String2*;
size_t *Number*;

int strcmp (*String1*, *String2***)**
char **String1*, **String2*;

int strncmp (*String1*, *String2*, *Number***)**
char **String1*, **String2*;
size_t *Number*;

int strcoll (*String1*, *String2***)**
char **String1*, **String2*;

size_t strxfrm (*String1*, *String2*, *Number***)**
char **String1*, **String2*;
size_t *Number*;

char *strcpy (*String1*, *String2***)**
char **String1*, **String2*;

char *strncpy (*String1*, *String2*, *Number***)**
char **String1*, **String2*;
size_t *Number*;

size_t strlen (*String***)**
char **String*;

char *strchr (*String*, *Character***)**
char **String*, *Character*;

char *strrchr (*String*, *Character***)**
char **String*, *Character*;

char *strpbrk (*String1*, *String2***)**
char **String1*, **String2*;

size_t strspn (*String1*, *String2***)**
char **String1*, **String2*;

size_t strcspn (*String1*, *String2***)**
char **String1*, **String2*;

char *strstr (*String1*, *String2*)
char **String1*, *String2*;

char *strtok (*String1*, *String2*)
char **String1*, **String2*;

char *strdup(*String1*)
char **String1*;

char *index (*String*, *Character*)
char **String*, *Character*;

char *rindex (*String*, *Character*)
char **String*, *Character*;

Description

The **string** subroutines copy, compare, and append strings in memory, and they determine such values as location, size, and the existence of strings in memory.

The parameters *String1*, *String2* and *String* point to strings. A string is an array of characters terminated by a null character. The subroutines **strcat**, **strncat**, **strcpy**, and **strncpy** all alter the string in the *String1* parameter. They do not check for overflow of the array to which *String1* points. All string movement is performed character by character and starts at the left. Overlapping moves toward the left work as expected, but overlapping moves to the right may give unexpected results. All of these subroutines are declared in the **string.h** header file.

The **strcat** subroutine adds a copy of the string pointed to by the *String2* parameter to the end of the string pointed to by the *String1* parameter. The **strcat** subroutine returns a pointer to the null–terminated result.

The **strncat** subroutine copies at most *Number* bytes of *String2* to the end of the string pointed to by the *String1* parameter. Copying stops before *Number* bytes if a null character is encountered in the *String2* string. The **strncat** subroutine returns a pointer to the null–terminated result.

The **strcmp** subroutine lexicographically compares the string pointed to by the *String1* parameter to the string pointed to by the *String2* parameter. The **strcmp** subroutine uses native character comparison, which may be signed or unsigned. The **strcmp** subroutine returns a value that is:

- Less than 0 if *String1* is less than *String2*
- Equal to 0 if *String1* is equal to *String2*
- Greater than 0 if *String1* is greater than *String2*.

The **strncmp** subroutine makes the same comparison as **strcmp**, but it compares at most *Number* pairs of characters.

The **strcoll** subroutine works the same as **strcmp**, except that the comparison is based on a collating sequence affected by the **setlocale** subroutine.

The **strxfrm** subroutine transforms the string pointed to by *String2* and places it in the array pointed to by *String1*, such that if **strcmp** is used on transformed strings it returns the same result as **strcoll** would for the corresponding untransformed strings. No more than *Number* characters are transformed. The **strxfrm** subroutine returns the length of the transformed string, not including the terminating null character. If the Number parameter is zero; **strxfrm** returns the length required to store the transformed string; not including the terminating null character.

The **strcpy** subroutine copies the string pointed to by the *String2* parameter to the character array pointed to by the *String1* parameter. Copying stops when the null character is copied. The **strcpy** subroutine returns the value of the *String1* parameter.

The **strncpy** subroutine copies *Number* bytes from the string pointed to by the *String2* parameter to the character array pointed to by the *String1* parameter. If *String2* is less than *Number* characters long, then **strncpy** pads *String1* with trailing null characters to fill *Number* bytes. If *String2* is *Number* or more characters long, then only the first *Number* characters are copied and the result is not terminated with a null character. The **strncpy** subroutine returns the value of the *String1* parameter.

The **strlen** subroutine returns the number of characters in the string pointed to by the *String* parameter, not including the terminating null character.

The **strchr** subroutine returns a pointer to the first occurrence of the character specified by the *Character* parameter in the string pointed to by the *String* parameter. A **NULL** pointer is returned if the character does not occur in the string. The null character that terminates a string is considered to be part of the string.

The **strrchr** subroutine returns a pointer to the last occurrence of the character specified by the *Character* parameter in the string pointed to by the *String* parameter. A **NULL** pointer is returned if the character does not occur in the string. The null character that terminates a string is considered to be part of the string.

The **strpbrk** subroutine returns a pointer to the first occurrence in the string pointed to by the *String1* parameter of any character from the string pointed to by the *String2* parameter. A **NULL** pointer is returned if no character matches.

The **strspn** subroutine returns the length of the initial segment of the string pointed to by the *String1* parameter that consists entirely of characters from the string pointed to by the *String2* parameter.

The **strcspn** subroutine returns the length of the initial segment of the string pointed to by the *String1* parameter that consists entirely of characters **not** from the string pointed to by the *String2* parameter.

The **strstr** subroutine finds the first occurrence in the *String1* string of the sequence of characters in the *String2* string (excluding the terminating null character). It returns a pointer to the found string in *String1*. It returns a **NULL** pointer if the string was not found. If *String2* points to a zero length string, the function returns *String1*.

The **strtok** subroutine returns a pointer to an occurrence of a text token in the string pointed to by the *String1* parameter. The *String2* parameter specifies a set of token delimiters. If the *String1* parameter is anything other than **NULL**, then the **strtok** subroutine reads the string pointed to by the *String1* parameter until it finds one of the delimiter characters specified by the *String2* parameter. It then stores a null character into the string, replacing the delimiter, and returns a pointer to the first character of the text token. The **strtok** subroutine keeps track of its position in the string so that subsequent calls with a **NULL** *String1* parameter step through the string. The delimiters specified by the *String2* parameter can be changed for subsequent calls to **strtok**. When no tokens remain in the string pointed to by the *String1* parameter, the **strtok** subroutine returns a **NULL** pointer.

The **strdup** subroutine returns a pointer to a new string, which is a duplicate of the string pointed to by the *String1* parameter. Space for the new string is obtained by using the malloc subroutine. A NULL pointer is returned if the new string cannot be created.

Compatibility Interface

The **index** and **rindex** subroutines are included for compatibility with BSD and are not part of the **ANSI C Library**. The **index** subroutine is implemented as a call to the **strchr** subroutine. The **rindex** subroutine is implemented as a call to the **strrchr** subroutine.

Implementation Specifics

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

Related Information

The **memccpy**, **memchr, memcmp, memcpy, memmove** subroutines, **NCstring** subroutines, **NLstring** subroutines, **setlocale** subroutine, and **swab** subroutine.

strncollen Subroutine

Purpose

Returns the number of collation values for a given string.

Library

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

Syntax

include <string.h>

int strncollen (*String*, *Number***)**
**const char ****String*;
const int *Number*;

Description

The **strncollen** subroutine returns the number of collation values for a given string pointed to by the *String* parameter. The count of collation values is terminated when either a null character is encountered or when the number of bytes indicated by the *Number* parameter have been examined.

The collation values are set by the **setlocale** subroutine for the **LC_COLLATE** category. For example, if the locale is set to Sp_SP (Spanish spoken Spain) for the **LC_COLLATE** category, where 'ch' has one collation value, then (**strncollen** ('abchd', 5) returns 4.

In German, there is the ß (double s) character, which has two collation values so that: **strncollen** ('straßa', 6) returns 7.

If a character has no collation value, then its collation length is 0.

Parameters

Number	The number of bytes in a string to be examined.
String	Pointer to a string to be examined for collation value.

Return Values

Upon successful completion, the **strncollen** subroutine returns the collation value for a given string, pointed to by the *String* parameter.

Implementation Specifics

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

Related Information

The **NCstring** subroutines, **setlocale** subroutine, **string** subroutines.

National Language Support Overview in *General Programming Concepts.*

strtol, strtoul, atol, or atoi Subroutine

Purpose

Converts a string to an integer.

Library

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

Syntax

long strtol (*String*, *Pointer*, *Base*)
char **String*, ***Pointer*;
int *Base*;

long atol (*String***)**
char **String*;

unsigned long strtoul (*String*, *Pointer*, *Base*)
char **String*, ***Pointer*;
int *Base*;

int atoi (*String***)**
char **String*;

Description

The **strtol** subroutine returns a long integer whose value is represented by the character string, *String*. The **strtol** subroutine scans the string up to the first character that is inconsistent with the *Base* parameter. Leading white-space characters are ignored, and an optional sign may precede the digits.

The **strtoul** subroutine differs in that it does not accept a leading sign character and returns an unsigned long integer.

The **atol** (*String*) subroutine is equivalent to **strtol** (*String*, (char **) NULL, 10).

The **atoi** (*String*) subroutine is equivalent to (**int**) **strtol** (*String*, (char **) NULL, 10).

The **atoi** and **atol** subroutines do not actually call the **strtol** subroutine.

Parameters

String	Specifies a character string.
Pointer	Specifies a pointer to a character string.
Base	Specifies the base to use for the conversion.

Return Values

If the value of the *Pointer* parameter is not (char **) NULL, then a pointer to the character that terminated the scan is stored in **Pointer*. If an integer cannot be formed, **Pointer* is set to *String*, and 0 is returned.

If the *Base* parameter is positive and not greater than 36, then it is used as the base for conversion. After an optional leading sign, leading zeros are ignored. "0x" or "0X" is ignored if *Base* is 16.

If the *Base* parameter is 0, the string determines the base. Thus, after an optional leading sign, a leading 0 indicates octal conversion, and a leading "0x" or "0X" indicates hexadecimal conversion. The default is to use decimal conversion.

The **strtol, atol,** and **atoi** subroutines perform conversions to integers. See the **strtod** subroutine for information on conversions to floating-point numbers.

Note: The **setlocale** function may affect the conversion in certain situations: for example, in programs using the radix character and the thousands separator.

Error Codes

On error, the global variable **errno** is set to:

EBADF The correct value of the converted number causes underflow or overflow.

Implementation Specifics

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

Related Information

The **scanf, fscanf**, **sscanf**, **NLscanf**, **NLfscanf**, **NLsscanf** subroutines, **atof, atoff, strtod, strtof** subroutines, **wstrtol, watol, watoi** subroutines, **wstrtod, watof** subroutines, **setlocale** subroutine.

stty or gtty Subroutine

Purpose

Sets or gets terminal state.

Library

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

Syntax

#include <sgtty.h>

stty(*FileDescriptor, Buffer***)**
int *FileDescriptor*;
**struct sgttyb ****Buffer*;

gtty(*FileDescriptor, Buffer***)**
int *FileDescriptor*;
**struct sgttyb ****Buffer*;

Description

This interface is made obsolete by the **ioctl** subroutine.

The **stty** subroutine sets the state of the terminal associated with the *FileDescriptor* parameter. The **gtty** subroutine retrieves the state of the terminal associated with *FileDescriptor.* To set the state of a terminal, the calling process must have write permission.

The **stty** subroutine is actually **ioctl(** *FileDescriptor,* TIOSETP, *Buffer***)**, while the **gtty** subroutine is actually **ioctl(** *FileDescriptor,* TIOGETP, *Buffer***)**.

Parameters

FileDescriptor	Specifies an open file descriptor.
Buffer	Specifies the buffer.

Return Values

If the **stty** or **gtty** subroutine is successful, a value of 0 is returned. Otherwise, a value of –1 is returned and the global variable **errno** is set to indicate the error.

Implementation Specifics

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

Related Information

The **ioctl** subroutine.

swab Subroutine

Purpose

Copies bytes.

Library

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

Syntax

void swab (*From*, *To*, *NumberOfBytes*)
char **From*, **To*;
int *NumberOfBytes*;

Description

The **swab** subroutine copies the number of bytes pointed to by the *NumberOfBytes* parameter from the location pointed to by the *From* parameter to the array pointed to by the *To* parameter, exchanging adjacent even and odd bytes.

The *NumberOfBytes* parameter should be even and non–negative. If the *NumberOfBytes* parameter is odd and positive, the **swab** subroutine uses *NumberOfBytes* –1 instead. If the *NumberOfBytes* parameter is negative, the **swab** subroutine does nothing.

Parameters

From	Points to the location of data to be copied.
To	Points to the array to which the data is to be copied.
NumberOfBytes	Specifies the number of even and non–negative bytes to be copied.

Implementation Specifics

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

Related Information

The **memccpy, memchr, memcmp, memmove, memset** subroutines, **string** subroutines.

swapon Subroutine

Purpose

Activates paging or swapping to a designated block device.

Syntax

int swapon (*PathName***);**
char **PathName;*

Description

The **swapon** subroutine makes the designated block device available to the system for allocation for paging and swapping.

The specified block device must be a logical volume on a disk device. The paging space size is determined from the current size of the logical volume.

Parameters

PathName	Specifies the full path name of the block device.

Error Codes

If an error occurs, **errno** is set to indicate the error:

EACCES	A component of the *PathName* prefix does not denies search permission, or permission is denied for the named file.
EINTR	Signal was received while processing request.
EINVAL	Invalid argument (size of device is invalid).
ENODEV	Device does not exist.
ENOENT	The *PathName* file does not exist.
ENOMEM	The maximum number of paging space devices (16) are already defined or no memory is available.
ENOTBLK	Block device required.
ENOTDIR	A component of the *PathName* prefix is not a directory.
ENXIO	No such device address.

Other errors are from calls to the device driver's **open** subroutine or **ioctl** subroutine.

Implementation Specifics

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

Related Information

The **swapqry** subroutine.

The **swapon** command.

swapqry Subroutine

Purpose

Returns paging device status.

Syntax

#include <sys/vminfo.h>
int swapqry (*PathName*, *Buffer*)
char *PathName*;
struct pginfo **Buffer;*

Description

The **swapqry** subroutine returns information to a user-designated buffer about active paging and swap devices.

Parameters

PathName	Specifies the full path name of the block device.
Buffer	Points to the buffer into which the status is stored.

Return Values

The **swapqry** subroutine returns 0 if *PathName* is an active paging device; if *Buffer* is non-null, it also returns status information.

Error Codes

If an error occurs, the subroutine returns –1 and **errno** is set to indicate the error as follows:

EACCES	A component of the *PathName* prefix denies search permission, or permission is denied for the named file.
EFAULT	Buffer pointer is invalid.
EINVAL	Invalid argument.
EINTR	Signal was received while processing request.
ENODEV	Device is not an active paging device.
ENODEV	Device does not exist.
ENOENT	The *PathName* file does not exist.
ENOTBLK	Block device required.
ENOTDIR	A component of the *PathName* prefix is not a directory.
ENXIO	No such device address.

Implementation Specifics

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

Related Information

The **swapon** subroutine.

The **swapon** command.

symlink Subroutine

Purpose

Makes a symbolic link to a file.

Library

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

Syntax

int symlink (*Path1*, *Path2***)**
char **Path1*;
char **Path2*;

Description

The **symlink** subroutine creates a symbolic link with the file named by the *Path2* parameter which refers to the file named by the *Path1* parameter.

Like a hard link (described in the **link** subroutine), a symbolic link allows a file to have multiple names. The presence of a hard link guarantees the existence of a file, even after the original name has been removed. A symbolic link provides no such assurance; in fact, the file named by the *Path1* parameter need not exist when the link is created. In addition, a symbolic link can cross file system boundaries.

When a component of a path name refers to a symbolic link rather than a directory, the path name contained in the symbolic link is resolved. If the path name in the symbolic link starts with / (slash), the symbolic link path name is resolved relative to the process root directory. If the path name in the symbolic link does not start with / (slash), the symbolic link path name is resolved relative to the directory that contains the symbolic link.

If the symbolic link is not the last component of the original path name, remaining components of the original path name are resolved from there.

If the last component of the path name supplied to a subroutine refers to a symbolic link, the symbolic link path name may or may not be traversed. Most subroutines always traverse the link; for example, **chmod**, **link**, and **open**. The **statx** subroutine takes an argument that determines whether the link is to be traversed.

Other subroutines refer only to the symbolic link itself, rather than to the object to which the link refers. These subroutines are:

chown This call changes the owner and/or group of the **symlink** itself.

Note: **chmod** does follow the link. This behavior is consistent with 4.3 BSD.

mkdir This call will fail with **EEXIST** if the target is a symbolic link.

mknod It is an error if a symbolic link exists with the same name as the file to be created (the *Path* parameter in **mknod** and **mkfifo**). The call will fail with **EEXIST** if the target is a symbolic link.

open When **O_CREAT** and **O_EXCL** are specified and a symbolic link exists for the name, the **open** call will fail with **EEXIST**.

readlink This call applies only to symbolic links.

rename If the file to be renamed (the *FromPath* parameter in **rename**) is a symbolic link, the symbolic link is renamed. If the new name (the *ToPath* parameter in **rename**) refers to an existing symbolic link, the symbolic link is destroyed.

rmdir The call will fail with **ENOTDIR** if the target is a symbolic link.

symlink Running this subroutine causes an error if a symbolic link named by the *Path2* parameter already exists. A symbolic link can be created that refers to another symbolic link; that is, the *Path1* parameter can refer to a symbolic link.

unlink This call removes the symbolic link.

Since the mode of a symbolic link cannot be changed, its mode is ignored during the lookup process. Any files and directories referenced by a symbolic link are checked for access normally.

Parameters

Path1 Specifies the contents of the *Path2* symbolic link. It is a null–terminated string representing the object to which the symbolic link will point. *Path1* cannot be the **NULL** value and cannot be more than **MAXLINKLEN** characters long.

Path2 Names the symbolic link to be created.

Return Values

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

Error Codes

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

EEXIST *Path2* already exists.

EACCESS The requested operation requires writing in a directory with a mode that denies write permission.

EROFS The requested operation requires writing in a directory on a read–only file system.

ENOSPC The directory in which the entry for the symbolic link is being placed cannot be extended because there is no space left on the file system containing the directory.

EDQUOT The directory in which the entry for the symbolic link is being placed cannot be extended because the user's quota of disk blocks on the file system containing the directory has been exhausted.

The **symlink** subroutine can also fail if additional errors on page A–1 occur.

Implementation Specifics

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

Related Information

The **link** subroutine, **readlink** subroutine, **statx** subroutine, **unlink** subroutine.

The **ln** command.

sync Subroutine

Purpose

Updates all file systems.

Library

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

Syntax

void sync ( )

Description

The **sync** subroutine causes all information in memory that should be on disk to be written out. The writing, although scheduled, is not necessarily complete upon return from the **sync** system call. Types of information to be written include modified superblocks, i-nodes, data blocks, and indirect blocks.

The **sync** subroutine should be used by programs that examine a file system, such as the **df** command and the **fsck** command.

If Network File System is installed on your system, information in memory relating to remote files is scheduled to be sent to the remote node.

Implementation Specifics

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

Related Information

The **fsync** subroutine.

The **sync** command.

sysconf Subroutine

Purpose

Provides a method to determine the current value of a specified system limit or option.

Library

Standard C Library **<libc.a>**

Syntax

#include <unistd.h>
long sysconf (*Name***)**
int *Name*;

Description

The **sysconf** subroutine allows an application to determine the current setting of certain system parameters, limits, or options.

Parameter

Name Specifies which system variable's setting should be returned. The valid values for the *Name* parameter are defined in the **unistd.h** header file and are described below:

_SC_ARG_MAX The maximum byte length of the arguments for one of the **exec** functions, including environment data.

_SC_CHILD_MAX The number of simultaneous processes per real user ID.

_SC_CLK_TCK The clock tick increment as defined by CLK_TCK in the **time.h** header file.

_SC_NGROUPS_MAX The maximum number of simultaneous supplementary group IDs per process.

_SC_OPEN_MAX The maximum number of files that one process can have open at any one time.

_SC_PASS_MAX The maximum number of significant characters in a password (not including the terminating null character).

_SC_JOB_CONTROL If this symbol is defined (does not return a –1) then job control is supported.

_SC_SAVED_IDS If this symbol is defined (does not return a –1) then each process has a savedset-user ID and set-group ID.

_SC_VERSION The version or revision number of the POSIX standard implemented to indicate the 4-digit year and 2-digit month that the standard was approved by the IEEE Standards Board. This value is currently the long integer 198808.

The values returned for the above variables supported by AIX for RISC System/6000 will not change during the lifetime of the process making the call.

Return Values

If the **sysconf** subroutine is successful, the value of the kernel variable or limit specified by *Name* is returned.

Error Codes

If the name parameter is invalid, a –1 is returned and **errno** is set to EINVAL. If the name parameter is valid but is a variable not supported by AIX for RISC System/6000, a value of –1 is returned but the value of **errno** is not changed.

Implementation Specifics

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

syslog, openlog, closelog, or setlogmask Subroutine

Purpose

Controls the system log.

Library

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

Syntax

#include <syslog.h>

int openlog (*ID*, *LogOption*, *Facility*)
**char ***ID*;
int *LogOption*, *Facility*;

int syslog (*Priority*, *Message, Value...*)
int *Priority*;
char *Message:*

int closelog ( )

int setlogmask(*MaskPriority*)
int *MaskPriority*;

Description

The **syslog** subroutine writes messages onto the system log maintained by the **syslogd** command.

The message is similar to the **printf** *fmt* string, with the difference that %m is replaced by the current error message obtained from the **errno** global variable. A trailing new-line can be added to the message if needed.

Messages are read by the **syslogd** command and written to the system console or log file, or forwarded to the **syslogd** on the appropriate host.

If special processing is required, the **openlog** subroutine can be used to initialize the log file.

Messages are tagged with codes indicating the type of *Priority* for each. A *Priority* is encoded as a *Facility*, which describes the part of the system generating the message, and as a level, which indicates the severity of the message.

If **syslog** cannot pass the message to **syslogd**, it writes the message on /**dev/console**, provided the **LOG_CONS** option is set.

The **closelog** subroutine closes the log file.

The **setlogmask** subroutine uses the bit mask in *MaskPriority* to set the new log priority mask and returns the previous mask.

The LOG_MASK and LOG_UPTO macros in the **sys/syslog.h** file are used to create the priority mask. Calls to **syslog** with a priority mask that does not allow logging of that particular level of message cause the subroutine to return without logging the message.

Parameters

ID
Contains a string that is attached to the beginning of every message. The *Facility* parameter encodes a default facility from the previous list to be assigned to messages that do not have an explicit facility encoded.

LogOption
Specifies a bit field that indicates logging options. The values of *LogOption* are:

LOG_CONS
Sends messages to the console if unable to send them to **syslogd**. This option is useful in daemon processes that have no controlling terminal.

LOG_NDELAY
Opens the connection to **syslogd** immediately, instead of when the first message is logged. This option is useful for programs that need to manage the order in which file descriptors are allocated.

LOG_NOWAIT
Logs messages to the console without waiting for forked children. Use this option for processes that enable notification of child termination through SIGCHLD; otherwise, **syslog** may block, waiting for a child whose exit status has already been collected.

LOG_ODELAY
Delays opening until **syslog** is called.

LOG_PID
Logs the process ID with each message. This option is useful for identiifying daemons.

Facility
Specifies which of the following generated the message:

LOG_AUTH
The security authorization system: **login**, **su**, and so on.

LOG_DAEMON
System daemons.

LOG_KERN
Messages generated by the kernel. These cannot be generated by any user processes.

LOG_LPR
The line printer spooling system.

LOG_LOCAL0
].

through].

LOG_LOCAL7
Reserved for local use.

LOG_MAIL
The mail system.

LOG_NEWS
The news sub-system.

LOG_RFS
Remote file systems (Andrew File System and RVD).

LOG_UUCP
UUCP sub-system.

LOG_USER
Messages generated by user processes. This is the default facility when none is specified.

Priority Specifies the part of the system generating the message, and as a level, indicates the severity of the message. The level of severity is selected from the following list:

LOG_ALERT A condition that should be corrected immediately; for example, a corrupted database.

LOG_CRIT Critical conditions; for example, hard device errors.

LOG_DEBUG Messages containing information useful to debug a program.

LOG_EMERG A panic condition reported to all users; system is unusable.

LOG_ERR Error conditions.

LOG_INFO General information messages.

LOG_NOTICE Not an error condition, but a condition requiring special handling.

LOG_WARNING Warning messages.

MaskPriority Enables logging for the levels indicated by the bits in the mask that are set and disabled where the bits are not set. The default mask allows all priorities to be logged.

Message Specifies the number of the message as listed in the message log.

Value Specifies the values given in the *Value* parameters of the **printf** subroutine.

Examples

1. To log an error message concerning a possible security breach, such as the following, enter:

```
syslog (LOG_ALERT, "who:internal error 23");
```

2. To initialize the log file, set the log priority mask, and log an error message, enter:

```
openlog ("ftpd", LOG_PID, LOG_DAEMON);
setlogmask (LOG_UPTO (LOG_ERR));
syslog (LOG_INFO)
```

3. To log an error message from the system, enter:

```
syslog (LOG_INFO | LOG_LOCAL2, "foobar error: %m");
```

Implementation Specifics

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

Related Information

The **profil** subroutine, **end, etext** identifiers.

The **cc** command, **prof** command.

system Subroutine

Purpose

Runs a shell command.

Library

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

Syntax

#include <stdio.h>

int system (*String***)**
**char ****String*;

Description

The **system** subroutine passes the *String* parameter to the **sh** command as input. Then the **sh** command interprets *String* as a command and runs it.

The **system** subroutine invokes the **fork** subroutine to create a child process that in turn uses the **exec** subroutine to run /**bin**/**sh**, which interprets the shell command contained in the *String* parameter. If the system subroutine is invoked on the Trusted Path, it runs the Trusted Path shell (/**bin**/**tsh**). The current process waits until the shell has completed, then returns the exit status of the shell.

Parameter

String A valid **sh** shell command.

Note: The **system** subroutine runs only **sh** shell commands. The results are unpredictable if the *String* parameter is not a valid **sh** shell command.

Return Values

Upon successful completion, the **system** subroutine returns the exit status of the shell.

If the **fork** subroutine fails, then the **system** subroutine returns a value of –1. If the **exec** subroutine fails, then the system subroutine returns 127. In either case, **errno** is set to indicate the error.

Implementation Specifics

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

Related Information

The **exec** subroutine, **exit** subroutine, **fork** subroutine, and **wait** subroutine.

The **sh** command.

tcb Subroutine

Purpose

Alters the Trusted Computing Base status of a file.

Library

Security Library (**libs.a**)

Syntax

#include <sys/tcb.h>

int tcb (*Path*, *Flag*)
char **Path*;
int *Flag*;

Description

The **tcb** subroutine provides a mechanism to query or set the Trusted Computing Base attributes of a file.

Parameters

Path Specifies the path name of the file whose Trusted Computing Base status is to be changed.

Flag Specifies the function which is to be performed. Valid values are defined in the **sys/tcb.h** file and include the following:

TCB_ON Enables the Trusted Computing Base attribute of a file.

TCB_OFF Disables the Trusted Process and Trusted Computing Base attributes of a file.

TCB_QUERY Queries the Trusted Computing Base status of a file. This function will return one of the above values.

Return Values

Upon successful completion, the **tcb** subroutine returns a value of 0 if the *Flags* parameter is either **TCB_ON** or **TCB_OFF**; or if the *Flags* parameter is **TCB_QUERY**, the current status is returned. If the **tcb** subroutine fails, a value of –1 is returned and errno is set to indicate the error

Error Codes

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

EINVAL The *Flags* parameter is not one of **TCB_ON**, **TCB_OFF**, or **TCB_QUERY**.

Additional error codes are returned by the **stax** and **chmod** subroutines.

Security

Access Control: The calling process must have search permission for the object named by the *Path* parameter.

Implementation Specifics

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

Related Information

The **statx** subroutine, **chmod** subroutine.

The **chmod** command.

tcdrain Subroutine

Purpose

Waits for output to complete.

Library

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

Syntax

#include <termios.h>

int tcdrain(*FileDescriptor***)**
int *FileDescriptor*;

Description

The **tcdrain** subroutine waits until all output written to the object referred to by the *FileDescriptor* parameter has been transmitted.

Parameter

FileDescriptor	Specifies an open file descriptor.

Return Values

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

Error Codes

The **tcdrain** subroutine can fail if one or more of the following are true:

EBADF	The *FileDescriptor* parameter does not specify a valid file descriptor.
EINTR	A signal interrupted the **tcdrain** subroutine.
ENOTTY	The file associated with the *FileDescriptor* parameter is not a terminal.

Example

1. To wait until all output has been transmitted, enter:

```
rc = tcdrain(stdout);
```

Implementation Specifics

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

Related Information

The **tcflow** subroutine, **tcflush** subroutine, **tcsendbreak** subroutine.

The **termios.h** header file.

tcflow Subroutine

Purpose

Performs flow control functions.

Library

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

Syntax

#include <termios.h>

int tcflow(*FileDescriptor*, *Action***)**
int *FileDescriptor*;
int *Action*;

Description

The **tcflow** subroutine suspends transmission or reception of data on the object referred to by the *FileDescriptor* parameter, depending on the value of the *Action* parameter.

Parameters

FileDescriptor — Specifies an open file descriptor.

Action — Specifies one of the following:

TCOOFF	Suspend output.
TCOON	Restart suspended output.
TCIOFF	Transmit a STOP character, which is intended to cause the terminal device to stop transmitting data to the system. (See the description of IXOFF in the Input Modes section of the **termios.h** header file.)
TCION	Transmit a START character, which is intended to cause the terminal device to start transmitting data to the system. (See the description of IXOFF in the Input Modes section of the **termios.h** header file.)

Return Values

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

Example

To restart output from a terminal device, enter:

```
rc = tcflow(stdout, TCION);
```

Implementation Specifics

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

Related Information

The **tcdrain** subroutine, **tcflush** subroutine, **tcsendbreak** subroutine.

The **termios.h** header file.

tcflush Subroutine

Purpose

Discards data from the specified queue.

Library

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

Syntax

#include <termios.h>

int tcflush(*FileDescriptor, QueueSelector***)**
int *FileDescriptor*;
int *QueueSelector;*

Description

The **tcflush** subroutine discards any data written to the object referred to by the *FileDescriptor* parameter, or data received but not read by the object referred to by *FileDescriptor*, depending on the value of the *QueueSelector* parameter.

Parameters

FileDescriptor Specifies an open file descriptor.

QueueSelector Specifies one of the following:

TCIFLUSH Flush data received but not read.

TCOFLUSH Flush data written but not transmitted.

TCIOFLUSH Flush both of the following:

- data received but not read
- data written but not transmitted.

Return Values

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

Error Codes

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

EBADF The *FileDescriptor* parameter does not specify a valid file descriptor.

EINVAL The *QueueSelector* parameter does not specify a proper value.

ENOTTY The file associated with the *FileDescriptor* parameter is not a terminal.

Example

To flush the output queue, enter:

```
rc = tcflush(2, TCOFLUSH);
```

Implementation Specifics

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

Related Information

The **tcdrain** subroutine, **tcflow** subroutine, **tcsendbreak** subroutine.

The **termios.h** header file.

tcgetattr Subroutine

Purpose

Gets terminal state.

Library

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

Syntax

#include <termios.h>

int tcgetattr (*FileDescriptor*, *TermiosPointer***)**
int *FileDescriptor*;
struct termios **TermiosPointer*;

Description

The **tcgetattr** subroutine gets the parameters associated with the object referred to by the *FileDescriptor* parameter and stores them in the **termios** structure referenced by the *TermiosPointer* parameter. This subroutine is allowed from a background process; however, the terminal attributes may subsequently be changed by a foreground process.

Whether or not the terminal device supports having the input and output baud rates differ, the baud rates stored in the **termios** structure returned by the **tcgetattr** subroutine reflect the actual baud rates, even if they are equal. Returning the number zero as the input baud rate if differing baud rates are not supported is obsolete.

Parameters

FileDescriptor	Specifies an open file descriptor.
TermiosPointer	Points to a **termios** structure.

Return Values

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

Error Codes

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

EBADF	The *FileDescriptor* parameter does not specify a valid file descriptor.
ENOTTY	The file associated with the *FileDescriptor* parameter is not a terminal.

Example

To get the current terminal state information, enter:

```
rc = tcgetattr(stdout, &my_termios);
```

Implementation Specifics

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

Related Information

The **tcsetattr** subroutine.

The **termios.h** file.

tcgetpgrp Subroutine

Purpose

Gets foreground process group ID.

Library

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

Syntax

#include <termios.h>

pid_t tcgetpgrp(*FileDescriptor***)**
int *FileDescriptor*;

Description

The **tcgetpgrp** subroutine returns the value of the process group ID of the foreground process group associated with the terminal. The function can be called from a background process; however the information may be subsequently changed by the foreground process.

The baud rates stored in the **termios** structure returned by the **tcgetattr** subroutine reflects the actual baud rates, even if they are equal, whether or not the terminal device supports different input and output baud rates.

Note: Returning 0 as the input baud rate if differing baud rates are not supported is obsolete.

Parameter

FileDescriptor Indicates the open file descriptor for the terminal special file.

Return Values

Upon successful completion, the process group ID of the foreground process is returned. Otherwise, a value of –1 is returned and the global variable **errno** is set to indicate the error.

Error Codes

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

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

EINVAL The function is not appropriate for the file associated with the *FileDescriptor* argument.

ENOTTY The calling process does not have a controlling terminal or the file is not the controlling terminal.

Implementation Specifics

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

Related Information

The **tcsetpgrp** subroutine.

tcsendbreak Subroutine

Purpose

Sends a break on an asynchronous serial data line.

Library

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

Syntax

#include <termios.h>

int tcsendbreak(*FileDescriptor, Duration***)**
int *FileDescriptor*;
int *Duration*;

Description

If the terminal is using asynchronous serial data transmission, the **tcsendbreak** subroutine causes transmission of a continuous stream of zero–valued bits for a specific duration.

If the terminal is not using asynchronous serial data transmission, the **tcsendbreak** subroutine returns without taking any action.

Parameters

FileDescriptor Specifies an open file descriptor.

Duration Specifies the number of milliseconds that zero–valued bits are transmitted. If the value of the *Duration* parameter is 0, it causes transmission of zero–valued bits for 25 milliseconds. If *Duration* is not 0, it sends zero–valued bits for *Duration* milliseconds.

Return Values

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

Examples

1. To send a break condition for 500 milliseconds:

```
rc = tcsendbreak(stdout, 500);
```

2. To send a break condition for 25 milliseconds:

```
rc = tcsendbreak(1, 25);
```

This could also be performed using the default *Duration*:

```
rc = tcsendbreak(1, 0);
```

Error Codes

The **tcsendbreak** subroutine fails if one or both of the following are true:

EBADF The *FileDescriptor* parameter does not specify a valid open file descriptor.

ENOTTY The file associated with the *FileDescriptor* parameter is not a terminal.

Implementation Specifics

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

Pseudo terminals and HFTs do not generate a break condition. They return without taking any action.

Related Information

The **tcdrain** subroutine, **tcflush** subroutine, **tcflow** subroutine.

The **termios.h** header file.

tcsetattr Subroutine

Purpose

Sets terminal state.

Library

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

Syntax

#include <termios.h>

int tcsetattr(*FileDescriptor,OptionalActions,TermiosPointer***)**
int *FileDescriptor*, *OptionalActions*;
struct termios **TermiosPointer*;

Description

The **tcsetattr** subroutine sets the parameters associated with the object referred to by the *FileDescriptor* parameter (unless support required from the underlying hardware is unavailable), from the **termios** structure referenced by the *TermiosPointer* parameter.

The value of the *OptionalActions* parameter determines how the **tcsetattr** subroutine is handled.

The 0 baud rate (B0) is used to terminate the connection. If B0 is specified as the output baud rate when the **tcsetattr** subroutine is called, the modem control lines are no longer asserted. Normally, this will disconnect the line.

Using 0 as the input baud rate in the **termios** structure to cause **tcsetattr** to change the input baud rate to the same value as that specified by the value of the output baud rate, is obsolete.

If an attempt is made using the **tcsetattr** subroutine to set:

- an unsupported baud rate,
- baud rates where the input and output baud rates differ and the hardware does dot support that combination,
- other features not supported by the hardware,

but it is able to perform some of the requested actions, it returns successfully, having set all the attributes that the implementation supports as requested, and leaving all the attributes not supported by the hardware unchanged.

If no part of the request can be honored, the **tcsetattr** subroutine returns a value of –1 and the global variable **errno** is set to **EINVAL**.

If the input and output baud rates differ and are a combination that is not supported, neither baud rate is changed. A subsequent call to the **tcgetattr** subroutine returns the actual state of the terminal device (reflecting both the changes made and not made in the previous **tcsetattr** call). The **tcsetattr** subroutine does not change the values in the **termios** structure whether of not it actually accepts them.

Parameters

FileDescriptor Specifies an open file descriptor.

OptionalActions Specifies one of the following values:

TCSANOW The change occurs immediately.

TCSADRAIN The change occurs after all output written to the object referred to by *FileDescriptor* has been transmitted. This function should be used when changing parameters that affect output.

TCSAFLUSH The change occurs after all output written to the object referred to by *FileDescriptor* has been transmitted. All input that has been received but not read is discarded before the change is made.

TermiosPointer Points to a **termios** structure.

Return Values

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

Error Codes

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

EBADF The *FileDescriptor* parameter does not specify a valid file descriptor.

EINVAL The *OptionalActions* argument is not a proper value, or an attempt was made to change an attribute represented in the **termios** structure to an unsupported value.

ENOTTY The file associated with the *FileDescriptor* parameter is not a terminal.

Example

To set the terminal state after the current output completes, enter:

```
rc = tcsetattr(stdout, TCSADRAIN, &my_termios);
```

Implementation Specifics

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

Related Information

The **tcgetattr** subroutine.

The **termios.h** file.

tcsetpgrp Subroutine

Purpose

Sets foreground process group ID.

Library

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

Syntax

#include <termios.h>

int tcsetpgrp(*FileDescriptor*, *ProcessGroupID***)**
int *FileDescriptor*;
pid_t *ProcessGroupID*;

Description

If the process has a controlling terminal, the **tcsetpgrp** subroutine sets the foreground process group ID associated with the terminal to the value of the *ProcessGroupID* parameter. The file associated with the *FileDescriptor* parameter must be the controlling terminal of the calling process, and the controlling terminal must be currently associated with the session of the calling process. The value of the *ProcessGroupID* parameter must match a process group ID of a process in the same session as the calling process.

Parameters

FileDescriptor Specifies an open file descriptor.

ProcessGroupID Specifies the process group identifier.

Return Value

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

Error Codes

This function can fail for the following reasons:

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

EINVAL The function is not appropriate for the file associated with the *FileDescriptor* parameter.

EINVAL The *ProcessGroupID* parameter is invalid.

ENOTTY The calling process does not have a controlling terminal or the file is not the controlling terminal.

EPERM The *ProcessGroupID* parameter is valid, but matches a process ID or process group ID of a process in another session.

Implementation Specifics

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

Related Information

The **tcgetpgrp** subroutine.

termdef Subroutine

Purpose

Queries terminal characteristics.

Library

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

Syntax

char *termdef (*FileDescriptor*, *Characteristic*)
int *FileDescriptor*;
char *Characteristic*;

Description

The **termdef** subroutine returns a pointer to a null-terminated static character string that identifies a characteristic of the terminal that is open on the file descriptor specified by the *FileDescriptor* parameter.

Parameters

FileDescriptor Specifies an open file descriptor.

Characteristic Specifies the characteristic that is to be queried. The following values can be specified:

c This causes **termdef** to query for the number of "columns" for the terminal. This is determined by performing the following actions:

1. It requests a copy of the terminal's **winsize** structure by issuing the **TIOCGWINSZ ioctl**. If ws_col is not 0 (zero), the ws_col value is used.
2. If the **TIOCGWINSZ ioctl** fails or if ws_col is 0 (zero), **termdef** queries the terminal device using the **Query HFT Device** command. If the HFT query is successful, **termdef** uses the value returned by the query.
3. If the **Query HFT Device** command fails, **termdef** attempts to use the value of the **COLUMNS** environment valuable.
4. If the **COLUMNS** environment variable is not set, **termdef** returns a pointer to a NULL string.

l This causes **termdef** to query for the number of "lines" (or rows) for the terminal. This is determined by performing the following actions:

1. It requests a copy of the terminal's **winsize** structure by issuing the **TIOCGWINSZ ioctl**. If ws_row is not 0 (zero), the ws_row value is used.
2. If the **TIOCGWINSZ ioctl** fails or if ws_row is 0 (zero), **termdef** queries the terminal device using the **Query HFT**

Device command. If the HFT query is successful, **termdef** uses the value returned by the query.

3. If the **Query HFT Device** command fails, **termdef** attempts to use the value of the **LINES** environment valuable.
4. If the **LINES** environment variable is not set, **termdef** returns a pointer to a NULL string.

Any other character (besides c or l)

This causes **termdef** to query for the "terminal type" of the terminal. This is determined by performing the following actions:

1. It queries the terminal device, using the **Query HFT Device** command.
2. If the **Query HFT Device** command fails, **termdef** attempts to use the value of the **TERM** environment variable.
3. If the **TERM** environment variable is not set, **termdef** returns a pointer to string set to "dumb".

Examples

1. To display the terminal type of the standard input device:

```
printf("%s\n", termdef(0, 't'));
```

2. To display the current lines and columns of the standard output device:

```
printf("lines\tcolumns\n%s\t%s\n", termdef(2, 'l'),
     termdef(2, 'c'));
```

Note: If **termdef** is unable to determine a value for lines or columns, it returns pointers to **NULL** strings.

Implementation Specifics

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

When *FileDescriptor* identifies an asynchronous terminal, the **Query HFT Device** command always fails and the environment variable is always checked. Shell profiles usually set the **TERM** variable each time you log in. the **stty** command allows you to change the lines and columns (by using the *lines* and *cols* options). This is preferred over changing the **LINES** and **COLUMNS** environment variables, since **termdef** examines the environment variables last. You may wish to set lines and columns if:

- You are using an asynchronous terminal and want to override the *lines* and *cols* setting in the **terminfo** data base, or
- Your asynchronous terminal has an unusual number of lines or columns and you are running an application that uses **termdef**, but not **terminfo**.

This is true because the **terminfo** initialization subroutine, **setupterm**, calls **termdef** to determine the number of lines and columns on the display. If **termdef** cannot supply this information, **setupterm** uses the values in the **terminfo** data base.

Related Information

The **Query HFT Device** command, **stty** command.

tmpfile Subroutine

Purpose

Creates a temporary file.

Library

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

Syntax

#include <stdio.h>

FILE *tmpfile ( )

Description

The **tmpfile** subroutine creates a temporary file and returns its **FILE** pointer. The file is opened for update. The temporary file is automatically deleted when the process using it terminates.

Return Values

If the file cannot be opened, the **tmpfile** subroutine writes an error message to the standard error output and returns a **NULL** pointer.

Implementation Specifics

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

Related Information

The **unlink** subroutine, **fopen**, **freopen**, **fdopen** subroutines, **mktemp** subroutine, **tmpnam**, **tempnam** subroutines.

tmpnam or tempnam Subroutine

Purpose

Constructs the name for a temporary file.

Library

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

Syntax

#include <stdio.h>

char *tmpnam (*String***)**
**char ****String*;

char *tempnam (*Directory*, *FileXPointer***)**
**char ****Directory*, ***FileXPointer*;

Description

The **tmpnam** subroutine and **tempnam** subroutine generate file names for temporary files.

The **tmpnam** subroutine generates a file name using the path name defined as P_tmpdir in the **stdio.h** header file.

Warning: The **tmpnam** subroutine generates a different file name each time it is called. If it is called more than 16,384 times by a single process, it starts recycling previously used names.

Files created using this subroutine reside in a directory intended for temporary use, and their names are unique. It is the user's responsibility to use the **unlink** subroutine to remove the file when no longer needed.

Between the time a file name is created and the file is opened, it is possible for some other process to create a file with the same name. This should not happen if that other process uses these subroutines or the **mktemp** subroutine, and if the file names are chosen to make duplication by other means unlikely.

Parameters

String — The address of an array of at least the number of bytes specified by L_tmpnam, a constant defined in the **stdio.h** header file.

If the *String* parameter is **NULL**, the **tmpnam** subroutine places its result into an internal static area and returns a pointer to that area. The next call to this subroutine destroys the contents of the area.

If the *String* parameter is not **NULL**, it is assumed to be the address of an array of at least the number of bytes specified by L_tmpnam. L_tmpnam is a constant defined in **stdio.h**. The **tmpnam** subroutine places its results into that array and returns the value of the *String* parameter.

Directory — A pointer to the path name of the directory in which the file is to created.

The **tempnam** subroutine allows you to control the choice of a directory. If the *Directory* parameter is **NULL** or points to a string that is not a path name for an appropriate directory, the path name defined as P_tmpdir in the **stdio.h** header file is used. If that path name is not accessible, **/tmp** is used. You can bypass the selection of a path name by providing an environment variable, **TMPDIR**, in the user's environment. The value of the **TMPDIR** variable is a path name for the desired temporary file directory.

FileXPointer A pointer to an initial character sequence with which the file name begins. The *FileXPointer* parameter can be **NULL**, or it can point to a string of characters to be used as the first characters of the temporary file name. The number of characters allowed is file system dependent, but five is the minimum allowed.

Implementation Specifics

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

Related Information

The **openx**, **open**, **creat** subroutines, **unlink** subroutine, **fopen**, **freopen**, **fdopen** subroutines.

The **malloc**, **free**, **realloc**, **calloc**, **mallopt**, **mallinfo**, **alloca** subroutines, **mktemp** subroutine, **mkstemp** subroutine, **tmpfile** subroutine.

The **environment** facility.

trcgen, trcgent Subroutines

Purpose

Records a trace event for a generic trace channel.

Syntax

#include <sys/trchkid.h>

void trcgen(*Channel, HkWord, DataWord, Length, Buffer***)**
unsigned int *Channel, HkWord, DataWord, Length;*
char **Buffer*

void trcgent(*Channel, HkWord, DataWord, Length, Buffer***)**
unsigned int *Channel, HkWord, DataWord, Length;*
char **Buffer*

Description

The **trcgen** subroutine records a trace event for a generic trace entry consisting of a hook word, a data word, and a variable number of bytes of trace data.

The **trcgent** subroutine records a trace event for a generic trace entry consisting of a hook word, a data word, a variable number of bytes of trace data, and a time stamp.

The **trcgen** subroutine and **trcgent** subroutine are located in pinned kernel memory.

Parameters

Buffer Pointer to a buffer of trace data.

Channel Channel number for the trace session, obtained from the **trcstart** subroutine.

Data Word A word of user-defined data.

HkWord An integer consisting of two bytes of user-defined data (*Data*), a hook ID (*HkID*), and a hook type (*HkType*).

Data Two bytes of user-defined data.

HkID A hook identifier which consists of a major hook ID and a minor hook ID. For user programs, the major hook ID value ranges from 0x01 to 0x0F. The minor hook ID value ranges from 0x0 to 0xF. There are no reserved values. For each major hook ID, there are 16 possible *HkID*s.

HkType A 4-bit hook type.

A unique hook type value is recorded by each trace subroutine. The valid hook type values are the following:

value	recorded by
0	**trchk** subroutine
1	**trchkt** subroutine
2	**trchkl** subroutine
3	**trchklt** subroutine

4	**trchkg** subroutine
5	**trchkgt** subroutine
6	**trcgen** subroutine
7	**trcgent** subroutine

Length Length in bytes of the *Buffer* parameter.

Implementation Specifics

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

File

sys/trchkid.h Trace Hookword Header file.

Related Information

The **trace** daemon.

The **trcgenk** kernel service, **trcgenkt** kernel service.

The **trchk** subroutine, **trcon** subroutine, **trcoff** subroutine, **trcstart** subroutine, **trcstop** subroutine.

trchk, trchkt, trchkl, trchklt, trchkg, trchkgt Subroutines

Purpose

Records a trace event.

Syntax

#include <sys/trchkid.h>

void trchk(*HkWord***)**
unsigned int *HkWord*

void trchkt(*HkWord***)**
unsigned int *HkWord*

void trchkl(*HkWord, HkData***)**
unsigned int *HkWord, HkData*

void trchklt(*HkWord, HkData***)**
unsigned int *HkWord, HkData*

void trchkg(*HkWord, D1, D2, D3, D4, D5***)**
unsigned int *HkWord, D1, D2, D3, D4, D5*

void trchkgt(*HkWord, D1, D2, D3, D4, D5***)**
unsigned int *HkWord, D1, D2, D3, D4, D5*

Description

The **trchk** subroutine, in all its forms records a trace event if a trace session is active. The **trchk** subroutines are located in pinned kernel memory.

The **trchk** subroutine records a *HkWord*.

The **trchkt** subroutine records a *HkWord* and a time stamp.

The **trchkl** subroutine records a *HkWord* and a data word (*HkData*).

The **trchklt** subroutine records a *HkWord* and a data word (*HkData*), and a time stamp.

The **trchkg** subroutine records a trace entry consisting of a *HkWord*, 5 words of user-defined data.

The **trchkgt** subroutine records a trace entry consisting of a *HkWord*, 5 words of user-defined data, and a time stamp.

Parameters

D1, D2, D3, D4, D5 User-defined data words.

HkData A word of user-defined data.

HkWord An integer consisting of two bytes of user-defined data (*Data*), a hook ID (*HkID*), and a hook type (*HkType*).

Data Two bytes of user-defined data.

HkID A hook identifier which consists of a major hook ID and a minor hook ID. For user programs, the major hook ID value ranges from 0x01 to 0x0F. The minor hook ID value ranges from 0x0 to 0xF. There are no reserved values. For each major hook ID, there are 16 possible *HkIDs*.

HkType A 4-bit hook type.

A unique hook type value is recorded by each trace subroutine. The valid hook type values are the following:

value	recorded by
0	**trchk** subroutine
1	**trchkt** subroutine
2	**trchkl** subroutine
3	**trchklt** subroutine
4	**trchkg** subroutine
5	**trchkgt** subroutine
6	**trcgen** subroutine
7	**trcgent** subroutine

Implementation Specifics

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

File

sys/trchkid.h Trace Hookword Header file.

Related Information

The **trace** daemon.

The **trcgenk** kernel service, **trcgenkt** kernel service.

The **trcgen** subroutine, **trcgent** subroutine **trcon** subroutine, **trcoff** subroutine, **trcstart** subroutine, **trcstop** subroutine.

trcoff Subroutine

Purpose

Halts the collection of trace data from within a process.

Syntax

int trcoff(*channel***)**
int *channel*

Description

The **trcoff** subroutine issues an **ioctl** subroutine to the trace device driver to stop trace data collection for a particular trace channel. The trace session must have already been started using the **trace** command or the **trcstart** subroutine.

Return Values

If the **ioctl** subroutine is successful, a value of 0 is returned and trace data collection is stopped. If the **ioctl** fails, a value of −1 is returned.

Implementation Specifics

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

Related Information

The **trace** daemon.

The **trcgenk** kernel service, **trcgenkt** kernel service.

The **trchk** subroutine, **trcgen** subroutine, **trcstart** subroutine, **trcon** subroutine, **trcstop** subroutine.

trcon Subroutine

Purpose

Starts the collection trace data.

Library

Run-time Services Library.

Syntax

int trcon(*Channel***)**

int *Channel*

Description

The **trcon** subroutine issues an **ioctl** subroutine to the trace device driver to start trace data collection for a particular trace channel. A trace session must have already been started for the trace channel using the **trace** command or the **trc_start** subroutine.

Parameter

Channel	Specifies one of 8 trace channels. Channel number 0 always refers to the Event/Performance trace. Channel numbers 1 – 7 specify generic trace channels.

Return Values

If the **ioctl** subroutine is successful, a value of 0 is returned and trace data collection is started. If the **ioctl** fails, a value of –1 is returned.

Implementation Specifics

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

File

librts.a	Run-time Services Library.

Related Information

The **trace** daemon.

The **trcgenk** kernel service, **trcgenkt** kernel service.

The **ioctl** subroutine, **trcgen** subroutine, **trchk** subroutine, **trcoff** subroutine, **trcstart** subroutine, **trcstop** subroutine.

trcstart Subroutine

Purpose

Starts a trace session.

Library

Run–time Services Library.

Syntax

int *(Argument)*.
char **Argument*

Description

The **trcstart** subroutine starts a trace session. The *Argument* parameter points to a character string containing the flags that are invoked with the **trace** daemon. To specify that a generic trace session is to be started, include the **–g** flag.

Return Values

If **trace** is started successfully, the channel number is returned. Channel number 0 is returned if a generic trace was not requested. If trace is not started successfully, a value of –1 is returned.

Implementation Specifics

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

Files

librts.a	Run-time Services Library.
/dev/trace	Trace Special File.

Related Information

The **trace** daemon.

The **trcon** subroutine.

trcstop Subroutine

Purpose

Stops a trace session.

Library

Run-time Services Library.

Syntax

int trcstop(*Channel***)**
int (*Channel***)**

Description

The **trcstop** subroutine stops a trace session for a particular trace channel.

Parameter

Channel	Specifies one of 8 trace channels. Channel number 0 always refers to the Event/Performance trace. Channel numbers 1 – 7 specify generic trace channels.

Return Values

If successful, a value of 0 is returned. If not successful, a value of –1 is returned.

Implementation Specifics

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

File

librts.a	Run-time Services Library.

Related Information

The **trace** daemon.

The **trcgenk** kernel service, **trcgenkt** kernel service.

The **ioctl** subroutine, **trchk** subroutine, **trcgen** subroutine, **trcoff** subroutine, **trcon** subroutine, **trcstart** subroutine.

truncate or ftruncate Subroutine

Purpose

Makes a file shorter.

Library

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

Syntax

int truncate (*Path*, *Length*)
char **Path*;
off_t *Length*;

int ftruncate (*FileDescriptor*, *Length*)
int *FileDescriptor*;
off_t *Length*;

Description

The **truncate** and **ftruncate** subroutines remove all data beyond the *Length* parameter bytes from the beginning of the specified file.

Full blocks are returned to the file system so that they can be used again, and the file size is changed to the lesser of the value of the *Length* parameter or the current length of the file.

These subroutines do not modify the seek pointer of the file.

These subroutines cannot be applied to a file that a process has open with **O_DEFER**.

Successful completion of the **ftruncate** or **truncate** subroutines clears the *SetUserID* and *SetGroupID* attributes of the file unless the caller has root user authority.

Parameters

Path	Specifies the name of a file that is opened, truncated, and then closed.
FileDescriptor	Specifies the descriptor of a file that must be open for writing.
Length	Specifies the number of bytes in the truncated file.

Return Values

Upon successful completion, a value of 0 is returned. If the **truncate** or **ftruncate** subroutine is unsuccessful, a value of –1 is returned, and the global variable **errno** is set to indicate the error.

Error Codes

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

EROF	An attempt was made to truncate a file that resides on a read–only file system.

The **truncate** subroutine can also fail if additional errors on page A–1 occur.

Note: In addition, the **truncate** subroutine can return the same errors as the **open** subroutine if there is a problem opening the file.

The **truncate** and **ftruncate** subroutines fail if one or more of the following are true:

EINVAL The file is not a regular file.

EMFILE The file is open with **O_DEFER** by one or more processes.

EAGAIN The write operation failed due to an enforced write lock on the file.

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

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

Implementation Specifics

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

Related Information

The **fclear** subroutine, **open** subroutine.

tsearch, tdelete, or twalk Subroutine

Purpose

Manages binary search trees.

Library

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

Syntax

#include <search.h>

void *tsearch (*Key*, *RootPointer*, *ComparisonPointer*)
int (*ComparisonPointer*) **( );**

void *tdelete (*Key*, *RootPointer*, *ComparisonPointer*)
int (*ComparisonPointer*) **( );**

void twalk (*Root*, *Action*)
void (*Action*) **( );**

Description

The **tsearch** subroutine performs a binary tree search.

The algorithm returns a pointer into a tree indicating where the data specified by the *Key* parameter can be found. If the data specified by the *Key* parameter is not found, the data is added to the tree in the correct place. If there is not enough space available to create a new node, a NULL pointer is returned. The *RootPointer* parameter points to a variable that points to the root of the tree. If the *RootPointer* parameter is the NULL value, the variable is set to point to the root of a new tree. If the *RootPointer* parameter is the NULL value on entry, then a NULL pointer is returned.

The **tdelete** subroutine deletes the data specified by the *Key* parameter. The *RootPointer* and *ComparisonPointer* parameters perform the same function as they do for the **tsearch** subroutine. The variable pointed to by the *RootPointer* parameter is changed if the deleted node is the root of the binary tree. The **tdelete** subroutine returns a pointer to the parent node of the deleted node. If the data is not found, a **NULL** pointer is returned. If the *RootPointer* parameter is **NULL** on entry, then a **NULL** pointer is returned.

The **twalk** subroutine steps through the binary search tree whose root is pointed to by the *RootPointer* parameter. (Any node in a tree can be used as the root to step through the tree below that node.) The *Action* parameter is the name of a routine to be invoked at each node.The routine specified by the *Action* parameter is called with three parameters. The first parameter is the address of the node currently being pointed to. The second parameter is a value from an enumeration data type:

```
typedef enum [preorder, postorder, endorder, leaf] VISIT;
```

(This data type is defined in the **search.h** header file.) The actual value of the second parameter depends on whether this is the first, second, or third time that the node has been visited during a depth-first, left-to-right traversal of the tree, or whether the node is a *leaf*. A *leaf* is a node that is not the parent of another node. The third parameter is the level of the node in the tree, with the root node being level zero.

Although declared as type pointer-to-void, the pointers to the key and the root of the tree 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.

Parameters

Key	Points to the data to be located.
ComparisonPointer	Points to the comparison function, which is called with two parameters that point to the elements being compared.
RootPointer	Points to a variable that points to the root of the tree.
Action	Names a routine to be invoked at each node.
Root	Points to the roots of a binary search node.

Return Values

The comparison function compares its parameters and return 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.

Implementation Specifics

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

Related Information

The **bsearch** subroutine, **hsearch** subroutine, **lsearch** subroutine.

Donald E. Knuth's *The Art of Computer Programming*, Volume 3, 6.2.2, Algorithm T. This book was published in Reading, Massachusetts by Addison-Wesley in 1981.

ttylock, ttywait, ttyunlock, or ttylocked Subroutine

Purpose

Controls tty locking functions.

Library

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

Syntax

int ttylock (*DeviceName***)**
char ***DeviceName;**

int ttywait (*DeviceName***)**
char ***DeviceName;**

int ttyunlock (*DeviceName***)**
char ***DeviceName;**

int ttylocked (*DeviceName***)**
char ***DeviceName;**

Description

The **ttylock** subroutine creates a file, **LCK..***DeviceName* in **/etc/locks** directory and writes the process ID of the calling process in that file. If **LCK..***DeviceName* exists and the process whose ID is contained in this file is active, **ttylock** returns an error

There are programs like uucp, connect, etc., that create tty locks in **/etc/locks**. The convention followed by these programs is to call **ttylock** with an argument of *DeviceName* for locking **/dev/***DeviceName*. This convention must be followed by all callers of **ttylock** to make the locking mechanism work.

The **ttywait** subroutine blocks the calling process until the lock file associated with *DeviceName*, **/etc/locks/LCK..***DeviceName*, is removed.

The **ttyunlock** subroutine removes the lock file, **/etc/locks/LCK..***DeviceName*, if it is held by the current process.

The **ttyunlocked** subroutine checks to see if the lock file, **/etc/locks/LCK..***DeviceName*, exists and the process that created the lock file is still active. If the process is no longer active, the lock file is removed..

Parameter

DeviceName	Specifies the name of the device.

Return Values

Upon successful completion, the **ttylock** subroutine returns a value of 0. Otherwise, a value of –1 is returned.

The **ttylocked** subroutine returns a value of 0 if no process has a lock on device. Otherwise, a value of –1 is returned.

Examples

1. To create a lock for **/dev/tty0**:

```
rc = ttylock("tty0");
```

2. To lock **/dev/tty0** device and wait for lock to be cleared if it exists:

```
if (ttylock("tty0"))
    ttywait("tty0");
rc = ttylock("tty0");
```

3. To remove the lock file for device **/dev/tty0** created by a previous call to **ttylock**:

```
ttyunlock("tty0);
```

4. To check for a lock on **/dev/tty0**:

```
rc = ttylocked("tty0");
```

Implementation Specifics

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

ttyname or isatty Subroutine

Purpose

Gets the name of a terminal or determines if the device is a terminal.

Library

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

Syntax

char *ttyname(*FileDescriptor***)**
int *FileDescriptor*;

int isatty(*FileDescriptor***)**
int *FileDescriptor*;

Description

The **ttyname** subroutine gets the name of a terminal.

The **isatty** subroutine determines if the device associated with the file descriptor specified by the *FileDescriptor* parameter is a terminal.

Parameter

FileDescriptor	Specifies an open file descriptor.

Return Values

The **ttyname** subroutine returns a pointer to a string containing the null–terminated path name of the terminal device associated with the file descriptor specified by the *FileDescriptor* parameter. A **NULL** pointer is returned if the file descriptor does not describe a terminal device in the /**dev** directory.

The return value of the **ttyname** subroutine points to static data whose contents are overwritten by each call.

If the specified file descriptor is associated with a terminal, the **isatty** subroutine returns a value of 0. If the file descriptor is not associated with a terminal, a value of –1 is returned and the global variable **errno** is set to indicate the error.

Error Code

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

ENOTTY	The file associated with the *FileDescriptor* parameter is not a terminal.

Implementation Specifics

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

File

/dev/*	Terminal device.

Related Information

The **ttyslot** subroutine.

ttyslot Subroutine

Purpose

Finds the slot in the **utmp** file for the current user.

Library

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

Syntax

int ttyslot ( )

Description

The **ttyslot** subroutine returns the index of the current user's entry in the **/etc/utmp** file. The **ttyslot** subroutine scans the /**etc/utmp** file for the name of the terminal associated with the standard input, the standard output, or the error output file descriptors (0, 1, or 2).

The **ttyslot** subroutine returns –1 if an error is encountered while searching for the terminal name, or if none of the first three file descriptors (0, 1, and 2) is associated with a terminal device.

Files

/etc/inittab	The path to the **inittab** file, which controls the initialization process.
/etc/utmp	The path to the **utmp** file, which contains a record of users logged into the system.

Implementation Specifics

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

Related Information

The **getutent** subroutine, **ttyname**, **isatty** subroutines.

ulimit Subroutine

Purpose

Sets and gets user limits.

Library

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

Syntax

#include <ulimit.h>

off_t ulimit (*Command*, *NewLimit*)
int *Command*;
off_t *NewLimit*;

Description

The **ulimit** subroutine controls process limits.

With remote files, the **ulimit** subroutine values of the client node or local node are used.

Parameters

Command Specifies the form of control. The *Command* parameter values follow:

GET_FSIZE (1) Returns the process file size limit. The limit is in units of **UBSIZE** blocks (see the **sys/param.h** file) and is inherited by child processes. Files of any size can be read.

SET_FSIZE (2) Sets the process file size limit to the value of the *NewLimit* parameter. Any process can decrease this limit, but only a process with root user authority can increase the limit.

GET_DATALIM (3)
Returns the maximum possible break value (described in the **brk** and **sbrk** subroutines).

SET_DATALIM (1004)
Sets the maximum possible break value (described in the **brk** and **sbrk** subroutines). Returns the new maximum break value, which is the *NewLimit* parameter rounded up to the nearest page boundary.

GET_STACKLIM (1005)
Returns the lowest valid stack address. (Note that stacks grow from high addresses to low addresses.)

SET_STACKLIM (1006)
Sets the lowest valid stack address. Returns the new minimum valid stack address, which is the *NewLimit* parameter rounded down to the nearest page boundary.

GET_REALDIR (1007)
Returns the current value of the *real directory read* flag. If

this flag is a value of 0, a **read** system call(or **readx** with *Extension* parameter value of 0) against a directory returns fixed–format entries compatible with the System V UNIX operating system. Otherwise, a **read** (or **readx** with *Extension* parameter value of 0) against a directory returns the underlying physical format.

SET_REALDIR (1008)
Set the value of the *real directory read* flag. If the *NewLimit* parameter is a value of 0, this flag is cleared; otherwise, it is set. The old value of the *real directory read* flag is returned.

NewLimit Specifies the new limit. The value of the *NewLimit* parameter depends on the *Command* parameter value that is used.

Example

To increase the size of the stack by 4096 bytes (use `4096` or `PAGESIZE`), and set the rc to the new lowest valid stack address, enter:

```
rc = ulimit(SET_STACKLIM, ulimit(GET_STACKLIM, 0) - 4096);
```

Return Values

Upon successful completion, a non–negative value is returned. Otherwise, a value of –1 is returned and the global variable **errno** is set to indicate the error.

Error Codes

The **ulimit** subroutine fails and the limit remains unchanged if one or both of the following are true:

EPERM A process without root user authority attempts to increase the file size limit.

EINVAL The *Command* parameter is a value other than **GET_FSIZE, SET_FSIZE, GET_DATALIM, SET_DATALIM, GET_STACKLIM, SET_STACKLIM, GET_REALDIR**, or **SET_REALDIR**.

Implementation Specifics

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

Related Information

The **brk** subroutine, **sbrk** subroutine, **getrlimit**, **setrlimit** subroutines, **pathconf** subroutine, **vlimit** subroutine, **write** subroutine.

The **param.h** file.

umask Subroutine

Purpose

Sets and gets the value of the file creation mask.

Library

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

Syntax

mode_t umask (*CreationMask***)**
mode_t *CreationMask*;

Description

The **umask** subroutine sets the file mode creation mask of the process to the value of the *CreationMask* parameter and returns the previous value of the mask.

Whenever a file is created (by the **open**, **mkdir**, or **mknod** subroutine), all file permission bits set in the file mode creation mask are cleared in the mode of the created file. This clearing allows users to restrict the default access to their files.

The mask is inherited by child processes.

Parameter

CreationMask	Specifies the value of the file mode creation mask. The *CreationMask* parameter is constructed by logically ORing file permission bits defined in the **sys/mode.h** header file. Nine bits of the *CreationMask* parameter are significant.

Return Value

Upon successful completion, the previous value of the file mode creation mask is returned.

Implementation Specifics

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

Related Information

The **chmod** subroutine, **mknod** subroutine, **open** subroutine, **stat** subroutine.

The **sh** command, **ksh** command.

The **sys/mode.h** header file.

umount or uvmount Subroutine

Purpose

Removes a virtual file system from the file tree.

Library

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

Syntax

int umount(*Device***)**
**char ***Device;*

#include <sys/vmount.h>

int uvmount(*VirtualFileSystemID*, *Flag***)**
int *VirtualFileSystemID*;
int *Flag*;

Description

The **umount** and **uvmount** subroutines remove a virtual file system from the file tree.

The **umount** subroutine unmounts only file systems mounted from a block device (special file that is identified by the path to the block device).

In addition to local devices, the **uvmount** subroutine unmounts local or remote directories, identified by the *VirtualFileSystemID* parameter.

Only a calling process with root user authority can unmount a device mount. Either a process with root user authority or a user that has write access to the mounted–over file or directory can unmount file and directory mounts.

Parameters

Device — The path name of the block device to be unmounted for the **umount** subroutine.

VirtualFileSystemID — The unique identifier of the virtual file system to be unmounted for the **uvmount** subroutine. This value is returned when a virtual file system is created by the **vmount** subroutine and may subsequently be obtained by the **mntctl** subroutine. The *VirtualFileSystemID* is also reported via the **stat** subroutine in the st_vfs field.

Flag — Specifies special action for the **uvmount** system call. Currently only one value is defined:

UVMNT_FORCE — Force the unmount. This flag is ignored for device mounts.

Return Values

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

Error Codes

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

EPERM	The calling process does not have write permission to the root of the virtual file system, or the mounted object is a device or remote and the calling process does not have root user authority.
EINVAL	There is no virtual file system with the specified *VirtualFileSystemID*.
EBUSY	A device that is still in use is being unmounted.

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

EPERM	The calling process does not have root user authority.
ENOENT	The *Device* parameter does not exist.
ENOBLK	The *Device* parameter is not a block device.
EINVAL	The *Device* parameter is not mounted.
EINVAL	The *Device* parameter is not local.
EBUSY	A process is holding a reference to a file located on the file system.

The **umount** sytem call can also fail if additional errors on page A–1 occur.

Implementation Specifics

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

Related Information

The **mount** subroutine.

The **mount** command, **umount** command.

uname or unamex Subroutine

Purpose

Gets the name of the current AIX system.

Library

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

Syntax

#include <sys/utsname.h>

int uname (*Name***)**
struct utsname **Name*;

int unamex (*Name***)**
struct xutsname **Name*;

Description

The **uname** subroutine stores information identifying the current system in the structure pointed to by the *Name* parameter.

The **uname** subroutine uses the **utsname** structure, which is defined in the **sys/utsname.h** file, and it contains the following members:

```
char     sysname[SYS_NMLN];
char     nodename[SYS_NMLN];
char     release[SYS_NMLN];
char     version[SYS_NMLN];
char     machine[SYS_NMLN];
```

The **uname** subroutine returns a null–terminated character string naming the current system in the sysname character array. The nodename array contains the name that the system is known by on a communications network. The release and version arrays further identify the system. The machine array identifies the system unit hardware being used.

The **unamex** subroutine uses the **xutsname** structure, which is defined in the **sys/utsname.h** file, and it contains the following members:

```
unsigned long   nid;
long    reserved[3];
```

The xutsname.nid field is the binary form of the utsname.machine field. For local area networks in which a binary node name is appropriate, xutsname.nid contains such a name.

Parameter

Name A pointer to the **utsname** or **xutsname** structure.

Return Values

Upon successful completion, the **uname** or **unamex** subroutines return a non–negative value. Otherwise, a value of –1 is returned and the global variable **errno** is set to indicate the error.

Error Code

The **uname** and **unamex** subroutines fail if the following is true:

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

Implementation Specifics

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

Related Information

The **uname** command.

ungetc or ungetwc Subroutine

Purpose

Pushes a character back into the input stream.

Library

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

Syntax

#include <stdio.h>

int ungetc (*Character*, *Stream***)**
int *Character*;
FILE **Stream*;

Description

The **ungetc** subroutine inserts the character specified by the *Character* parameter into the buffer associated with the input stream specified by the *Stream* parameter. This causes the next call to the **getc** subroutine to return *Character*. The **ungetc** subroutine returns *Character*, and leaves the *Stream* parameter file unchanged.

If the *Character* parameter is **EOF**, the **ungetc** subroutine does not place anything in the buffer and a value of **EOF** is returned.

You can always push one character back onto a stream, provided that something has been read from the stream or the **setbuf** subroutine has been called. The **fseek** subroutine erases all memory of inserted characters.

The **ungetc** subroutine returns a value of **EOF** if it cannot insert the character.

For Japanese Language Support:

When running AIX with Japanese Language Support, the following subroutine, stored in **libc.a**, is provided:

#include <stdio.h>

int ungetwc (*Character*, *Stream***)**
int *Character*;
FILE **Stream*;

The **ungetwc** subroutine inserts the **NLchar** specified by the *Character* parameter into the buffer associated with the input stream. This causes the next call to the **getwc** subroutine to return the value of the *Character* parameter.

Parameters

Character	Specifies a character.
Stream	Specifies the input stream.

Return Value

The **ungetwc** subroutine returns a value of **EOF** if the character cannot be inserted.

Implementation Specifics

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

Related Information

The **fseek, rewind, ftell, fgetpos, fsetpos** subroutines, **getc, fgetc, getchar, getw, getwc, fgetwc, getwchar** subroutines, **setbuf**, **setvbuf**, **setbuffer**, **setlinebuf** subroutines.

The National Language Support Overview in *General Programming Concepts*.

unlink Subroutine

Purpose

Removes a directory entry.

Library

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

Syntax

int unlink (*Path*)
char **Path*;

Description

The **unlink** subroutine removes the directory entry specified by the *Path* parameter. If Network File System is installed on your system, this path can cross into another node.

Removing a link to a directory requires root user authority. Unlinking of directories is strongly discouraged since erroneous directory structures can result. The **rmdir** subroutine should be used to remove empty directories.

When all links to a file are removed and no process has the file open, all resources associated with the file are reclaimed, and the file is no longer accessible. If one or more processes have the file open when the last link is removed, the directory entry disappears, but the removal of the file contents is postponed until all references to the file are closed.

If the parent directory of *Path* has the *sticky* attribute (described in the **mode.h** header file), the calling process must have an *effective user ID* equal to the *owner ID* of *Path* or the *owner ID* of the parent directory of *Path*, or the calling process must have root user authority.

Parameter

Path	Specifies the directory entry to be removed.

Return Values

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

Error Codes

The **unlink** subroutine fails and the named file is not unlinked if one or more of the following are true:

ENOENT	The named file does not exist.
EACCES	Write permission is denied on the directory containing the link to be removed.
EPERM	The named file is a directory, and the calling process does not have root user authority.
EBUSY	The entry to be unlinked is the mount point for a mounted file system.
EROFS	The entry to be unlinked is part of a read–only file system.

The **unlink** subroutine can also fail if additional errors on page A–1 occur.

If Network File System is installed on the system, the **unlink** system call can also fail if the following is true:

ETIMEDOUT The connection timed out.

Implementation Specifics

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

Related Information

The **close** subroutine, **link** subroutine, **open** subroutine, **rmdir** subroutine.

The **rm** command.

unload Subroutine

Purpose

Unloads a program.

Syntax

int unload(*FunctionPointer***)**
int (**FunctionPointer***)( );**

Description

The **unload** subroutine unloads the object file and any imported object files that were automatically loaded with it. The pointer to the function returned by the **load** subroutine is passed to the **unload** subroutine as *FunctionPointer.*

The **unload** subroutine frees the storage used by the specified object file only if the object file is no longer in use. An object file is in use as long as any other object file that is in use imports symbols from it.

Parameter

FunctionPointer	Specifies the name of the function returned by the **load** subroutine.

Return Value

Upon successful completion, the **unload** subroutine returns a value of 0.

Error Codes

If the **unload** subroutine fails, a value of –1 is returned, the program is not unloaded, and **errno** is set to indicate the error.The **unload** subroutine returns the following error code:

EINVAL	The *FunctionPointer* parameter does not correspond to a program loaded by the **load** subroutine.

Implementation Specifics

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

Related Information

The **load** subroutine, **loadquery** subroutine, **loadbind** subroutine.

The **ld** command.

usrinfo Subroutine

Purpose

Gets and sets user information about the owner of the current process.

Library

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

Syntax

#include <uinfo.h>

int usrinfo (*Command*, *Buffer*, *Count***)**
int *Command*;
char **Buffer*;
int *Count*;

Description

The **usrinfo** subroutine gets and sets information about the owner of the current process. The information is a sequence of null–terminated *name=value* strings. The last string in the sequence is terminated by two successive null characters. A child process inherits the user information of the parent process.

Parameters

Command If the *Command* parameter is one of the following constants:

GETUINFO Copies up to the number of bytes specified by the *Count* parameter of user information into the buffer pointed to by the *Buffer* parameter.

SETUINFO Sets the user information for the process to the first number of bytes specified by the *Count* parameter in the buffer pointed to by the *Buffer* parameter. The calling process must have root user authority to set the user information.

The user information should at minimum consist of four strings that are typically set by the **login** program. These four strings are:

NAME=*UserName*

LOGIN=*LoginName*

LOGNAME=*LoginName*

TTY=*ttyName*

If the process has no terminal, the *ttyName* parameter should be null.

Buffer A pointer to a user buffer. This buffer is usually **UINFOSIZ** bytes long.

Count The number of bytes of user information to be copied from or to the user buffer.

Return Values

Upon successful completion, the **usrinfo** subroutine returns a non–negative integer giving the number of bytes transferred. Otherwise, a value of –1 is returned and the global variable **errno** is set to indicate the error.

Error Codes

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

EPERM	The *Command* parameter is set to **SETUINFO** and the calling process does not have root user authority.
EINVAL	The *Command* parameter is not set to **SETUINFO** or **GETUINFO**.
EINVAL	The *Command* parameter is set to **SETUINFO** and the *Count* parameter is larger than **UINFOSIZ**.
EFAULT	The *Buffer* parameter points outside of the address space of the process.

Implementation Specifics

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

Related Information

The **getuinfo** subroutine, **setpenv** subroutine.

The **login** command.

utimes or utime Subroutine

Purpose

Sets file access and modification times.

Library

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

Syntax

#include <sys/time.h>

int utimes (*Path*, *Times***)**
char ****Path*;**
struct timeval *Times***[2];**

#include <utime.h>

int utime (*Path*, *Times*)
char **Path*;
struct utimbuf **Times*;

Description

The **utimes** subroutine sets the access and modification times of the file pointed to by the *Path* parameter to the value of the *Times* parameter.

The **utime** call also sets file access and modification times; however, each time is contained in a single integer and is accurate only to the nearest second. The **utimes** subroutine allows time specifications accurate to the microsecond.

Parameters

Path Points to the file.

Times For **utimes**, this is an array of **timeval** structures, as defined in the **sys/time.h** header file. The first array element represents the date and time of last access, and the second element represents the date and time of last modification. The times in the **timeval** structure are measured in seconds and microseconds since the epoch (00:00:00 GMT, January 1, 1970), although rounding towards the nearest second may occur.

For **utime**, this parameter is a pointer to a **utimbuf** structure, defined in the **utime.h** header file. The first structure member represents the date and time of last access, and the second member represents the date and time of last modification. The times in the **utimbuf** structure are measured in seconds since the epoch.

If the *Times* parameter is **NULL**, the access and modification times of the file are set to the current time. If the file is a remote file, the current time at the remote node, rather than the local node, is used. To use the call this way, the effective user ID of the process must be the same as the owner of the file, or must have root user authority, or the process must have write permission to the file.

If the *Times* parameter is not the **NULL** value, the access and modification times are set to the values contained in the designated structure, regardless of whether those times correlate with the current time. Only the owner of the file or a user with root user authority can use the call this way.

Return Values

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

Error Codes

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

ENOENT The named file does not exist.

EPERM The *Times* parameter is not the **NULL** value and the calling process neither owns the file nor has root user authority.

EACCES The *Times* parameter is **NULL**, effective user ID is neither the owner of the file nor has root user authority, and write access is denied.

EROFS The file system that contains the file is mounted read–only.

The **utimes** subroutine can also fail if additional errors on page A–1 occur.

Implementation Specifics

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

Microsecond time stamps are not implemented, even though **utimes** provides a way to specify them.

Related Information

The **stat** subroutine.

The **sys/time.h** header file, **utime.h** header file.

varargs Macros

Purpose

Handles a variable-length parameter list.

Library

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

Syntax

#include <stdargs.h>

type va_arg (*Argp, Type***)**
va_list *Argp*;

void va_end (*Argp***)**
va_list *Argp*;

void va_start (*Argp, ParmN***)**
va_list *Argp*;

Description

The **varargs** set of macros allows you to write portable subroutines that accept a variable number of parameters. Subroutines that have variable-length parameter lists (such as the **printf** subroutine), but that do not use the **varargs** macros, are inherently nonportable because different systems use different parameter–passing conventions.

va_list Defines the type of the variable used to traverse the list.

va_start Initializes *Argp* to point to the beginning of the list. The optional parameter *ParmN* is the identifier of the rightmost parameter in the function definition. For compatiblity with older programs, it defaults to the address of the first parameter on the parameter list. The **va_start** macro will be invoked before any access to the unnamed arguments.

va_argp A variable that the **varargs** macros use to keep track of the current location in the parameter list. Do not modify this variable.

va_arg Returns the next parameter in the list pointed to by *Argp*.

va_end Cleans up at the end.

Your subroutine can traverse, or scan, the parameter list more than once. Start each traversal with a call to **va_start** and end it with **va_end**.

Note: The calling routine is responsible for specifying the number of parameters because it is not always possible to determine this from the stack frame. For example, **execl** is passed a **NULL** pointer to signal the end of the list. The **printf** subroutine determines the number of parameters from its *Format* parameter.

Parameters

Argp Specifies a variable that the **varargs** macros use to keep track of the current location in the parameter list. Do not modify this variable.

Type Specifies the type to which the expected argument will be converted when passed as an argument. In C, arguments that are char or short should be accessed as int; unsigned char or short are converted to unsigned int, and float arguments are converted to double. Different types can be mixed, but it is up to the routine to know what type of argument is expected, since it cannot be determined at runtime.

ParmN Specifies an optional parameter that is the identifier of the rightmost parameter in the function definition.

Example

The following example is a possible implementation of the **execl** system call:

```
\#include \<varargs.h>
\#define MAXargS 100
/*
**  execl is called by
**  execl(file, arg1, arg2, . . . , (char *) 0);
*/
execl(va_alist)
   va_dcl
{  va_list ap;
   char *file;
   char *args[MAXargS];
   int argno = 0;
   va_start(ap);
   file = va_arg(ap, char *);
   while ((args[argno++] = va_arg(ap, char *)) != (char *) 0)
      ;   /* Empty loop body */
   va_end(ap);
   return (execv(file, args));
}
```

Implementation Specifics

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

Related Information

The **printf** subroutine, **NLvprintf** subroutine.

The **exec** subroutines.

vmount or mount Subroutine

Purpose

Makes a file system available for use.

Library

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

Syntax

#include <sys/vmount.h>

int vmount (*VMount*, *Size*)
struct vmount * *VMount*;
int *Size*;

int mount (*Device*, *Path*, *Flags*)
char **Device*;
char **Path*;
int *Flags*;

Description

The **vmount** subroutine mounts a file system, thereby making the file in it available for use. The **vmount** subroutine effectively creates what is known as a *virtual file system*. After a file system is mounted, references to the path name that is to be mounted over refers to the root directory on the mounted file system.

The **vmount** subroutine provides the following types of mounts:

- A local file over a local or remote file
- A local directory over a local or remote directory
- A remote file over a local or remote file
- A remote directory over a local or remote directory.

A directory can only be mounted over a directory, and a file can only be mounted over a file.

A mount to a directory or a file can be issued if the user has both of the following:

- Search permission to the directory or file to mount
- Search and write permission to the directory or file to mount over.

In order to mount a block device, remote file, or remote directory, the calling process must also have root user authority.

The **mount** subroutine only allows mounts of a block device over a local directory with the default file system type. **mount** searches **/etc/filesystems** to find a corresponding stanza for the desired file system. The mount interface is provided only for compatibility with previous releases of AIX. The use of **mount** is strongly discouraged by normal application programs.

Parameters

Device — A path name identifying the block device (also called a special file) that contains the physical file system.

Path A path name identifying the directory on which the file system is to be mounted.

Flags Values that define characteristics of the object to be mounted. Currently one value is defined in the **sys/vmount.h** header file:

MNT_READONLY Indicates that the object to be mounted is read–only and that write access is not allowed. If this value is not specified, writing is permitted according to individual file accessibility.

VMount A pointer to a variable length **vmount** structure. The **vmount** structure is defined in the **sys/vmount.h** header file.

The following fields of the *VMount* parameter must be initialized before the call to the **vmount** subroutine:

vmt_revision The revision code in effect when the program that created this virtual file system was compiled. This is the value **VMT_REVISION**.

vmt_length The total length of the structure with all its data. This must be a multiple of the word size (4 bytes) and correspond with the *Size* parameter.

vmt_flags Contains the general mount characteristics. The following values may be specified:

MNT_READONLY A read–only virtual file system is to be created.

vmt_gfstype The type of the generic file system underlying the **VMT_OBJECT**. Values for this field are defined in the **sys/vmount.h** header file and include:

MNT_JFS The AIX Version 3 Operating System native file system.

MNT_NFS A Network File System client.

MNT_CDROM The CD–ROM file system.

vmt_data An array of structures that describe variable length data associated with the **vmount** structure. The structure consists of the following fields:

vmt_off The offset of the data from the beginning of the **vmount** structure.

vmt_size The size, in bytes, of the data.

The array consists of the following elements:

vmt_data[**VMT_OBJECT**]
The name of the device, directory, or file that is to be mounted.

vmt_data[**VMT_STUB**]
: The name of the device, directory, or file that is to be mounted over.

vmt_data[**VMT_HOST**]
: The short (binary) name of the host that owns the mounted object. This need not be specified if **VMT_OBJECT** is local (has the same vmt_gfstype as /, the root of all file systems).

vmt_data[**VMT_HOSTNAME**]
: The long (character) name of the host that owns the mounted object. This need not be specified if **VMT_OBJECT** is local.

vmt_data[**VMT_INFO**]
: Binary information to be passed to the generic file system implementation that supports **VMT_OBJECT**; the interpretation of this field is specific to the gfs_type.

vmt_data[**VMT_ARGS**]
: A character string representation of **VMT_INFO**.

On return from the **vmount** subroutine, the following additional fields of the *VMount* parameter are initialized:

vmt_fsid — The two-word file system identifier; the interpretation of this identifier depends on the gfs_type.

vmt_vfsnumber — The unique identifier of the virtual file system. Virtual file systems do not survive the IPL; neither does this identifier.

vmt_time — The time at which the virtual file system was created.

Size — The size, in bytes, of the supplied data area.

Return Values

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

Error Codes

The **mount** and **vmount** subroutines fail and the virtual file system is not created if one or more of the following are true:

EACCES — The calling process does not have write permission on the stub directory (the directory to be mounted over).

EBUSY — **VMT_OBJECT** specifies a device that is already mounted or an object that is open for writing, or the kernel's mount table is full.

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

EFBIG — The size of the file system is too big.

EFORMAT An internal inconsistency has ben detected in the file system.

EINVAL The contents of the *VMount* parameter are unintelligible (for example, the *vmt_gfstype* is unrecognizable, or the file system implementation does not understand the **VMT_INFO** provided).

ENOSYS The file system type requested has not been configured.

ENOTBLK The object to be mounted is not a file, directory, or device.

ENOTDIR The types of **VMT_OBJECT** and **VMT_STUB** are incompatible.

EPERM **VMT_OBJECT** specifies a block device and the calling process does not have root user authority.

EROFS An attempt has been made to mount a file system for read/write when the file system cannot support writing.

The **mount** and **vmount** subroutines can also fail if additional errors on page A–1 occur.

Implementation Specifics

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

Related Information

The **umount** subroutine, **mntctl** subroutine.

The **mount** command, **umount** command.

vprintf, vfprintf, or vsprintf Subroutine

Purpose

Formats a **varargs** parameter list for output.

Library

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

Syntax

#include <stdio.h>
#include <stdarg.h>

int vprintf (*Format*, *PrintArgument*)
char **Format*;
va_list *PrintArgument*;

int vfprintf (*Stream*, *Format*, *PrintArgument*)
FILE **Stream*;
char **Format*;
va_list *PrintArgument*;

int vsprintf (*String*, *Format*, *PrintArgument*)
char **String*, **Format*;
va_list *PrintArgument*;

Description

The **vprintf**, **vfprintf**, and **vsprintf** subroutines format and write **varargs** parameter lists.

These subroutines are the same as the **printf**, **fprintf**, and **sprintf** 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 **varargs**.

Parameters

Format Specifies a character string that contains two types of objects:

- Plain characters, which are copied to the output stream
- Conversion specifications, each of which causes zero or more items to be fetched from the **varargs** parameter lists.

PrintArgument Specifies the arguments to be printed.

Stream Specifies the output stream.

String Specifies the buffer to which output is printed.

Example

The following example demonstrates how the **vfprintf** subroutine can be used to write an error routine:

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

/* error 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 error    */
   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 AIX Base Operating System (BOS) Runtime.

Related Information

The **NLvprintf**, **NLvfprintf**, **NLvsprintf** subroutines, **printf**, **fprintf**, **sprintf**, **NLprintf**, **NLfprintf**, **NLsprintf** subroutines.

wait, waitpid, or wait3 Subroutine

Purpose

Waits for a child process to stop or terminate.

Library

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

Syntax

#include <sys/wait.h>

pid_t wait (*StatusLocation*)
int **StatusLocation*;

pid_t wait ((void *) 0)

#include <sys/wait.h>

pid_t waitpid (*ProcessID, StatusLocation, Options*)
int **StatusLocation*;
pid_t *ProcessID*;
int *Options*;

#include <sys/time.h>
#include <sys/resource.h>
#include <sys/wait.h>

pid_t wait3 (*StatusLocation*, *Options*, *ResourceUsage*)
int **StatusLocation*;
int *Options*;
struct rusage **ResourceUsage*;

Description

The **wait** subroutine suspends the calling process until it receives a signal that is not blocked or ignored, or until any one of the calling process child processes stops or terminates. The **wait** subroutine returns without waiting if the child process that has not been waited for has already stopped or terminated prior to the call.

Note: The effect of the **wait** system call can be modified by the setting of the **SIGCHLD** signal. See the **sigaction** subroutine for details.

The **waitpid** subroutine includes a *ProcessID* parameter that allows the calling process to gather status from a specific set of child processes, according to the following rules:

- If the *ProcessID* parameter is equal to a value of –1, status is requested for any child process. In this respect, the **waitpid** subroutine is equivalent to the **wait** subroutine.
- If the *ProcessID* parameter is greater than 0, it specifies the process ID of a single child process for which status is requested.

- If the *ProcessID* parameter is equal to 0, status is requested for any child process whose process group ID is equal to that of the calling process.
- If the *ProcessID* parameter is less than 0, status is requested for any child process whose process group ID is equal to the absolute value of the *ProcessID* parameter.

The **waitpid** and **wait3** subroutine variants provide an *Options* parameter that can modify the behavior of the subroutine. Two values are defined, **WNOHANG** and **WUNTRACED**, which can be combined by specifying their bitwise–inclusive OR. The **WNOHANG** option prevents the calling process from being suspended even if there are child processes to wait for. In this case, a value of 0 is returned indicating that there are no child processes that have stopped or terminated. If the **WUNTRACED** option is set, the call should also return information when children of the current process are stopped because they received a **SIGTTIN, SIGTTOU, SIGSSTP,** or **SIGTSTOP** signal.

When multiprocess debugging mode is enabled, the following new values are returned from a **wait** subroutine:

W_SEWTED Process stopped during **exec**.

W_SFWTED Process stopped during **fork**.

W_SLWTED Process stopped during **load** or **unload** subroutine.

Note: W_SLWTED is also returned when multiprocess debugging is disabled.

Parameters

StatusLocation Points to structure that is filled in with the child process termination status, as defined in the **sys/wait.h** header file.

ProcessID Specifies the child process.

Options Modifies behavior of subroutine.

ResourceUsage Specifies the location of a structure to be filled in with resource utilization information for terminated children.

Return Values

If the **wait** subroutine returns due to a stopped or terminated child process, the process ID of the child is returned to the calling process. If the **wait** subroutine fails, a value of –1 is returned and the global variable **errno** is set to indicate the error. In addition, the **waitpid** and **wait3** subroutines will return a value of 0 if there are no stopped or exited children, and the **WNOHANG** option was specified.

Error Codes

The **wait**, **waitpid**, and **wait3** subroutines fail if one or more of the following are true:

ECHILD The calling process has no existing unwaited–for child processes.

EINTR This subroutine was terminated by receipt of a signal.

EFAULT The *StatusLocation* or *ResourceUsage* parameter points to a location outside of the address space of the process.

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

ECHILD The process or process group ID specified by the *ProcessID* parameter does not exist or is not a child process of the calling process.

The **waitpid** and **wait3** subroutines fail if the following is true:

EINVAL The value of the *Options* parameter is not valid.

Implementation Specifics

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

Related Information

The **exec** subroutines, **_exit, exit**, **atexit** subroutines**, fork** subroutine, **pause** subroutine, **ptrace** subroutine, **getrusage** subroutine, **sigaction** subroutine.

wcscat, wcschr, wcscmp, wcscpy ,or wcscspn Subroutine

Purpose

Performs operations on wide–character strings .

Library

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

Syntax

#include <wcstr.h>

wchar_t *wcscat(*WcString1, WcString2*)
wchar_t * *WcString1*, * *WcString2*;

wchar_t *wcschr(*WcString, WideCharacter*)
wchar_t * *WcString, WideCharacter;*

wchar_t *wcscmp (*WcString1, WcString2*)
wchar_t * *WcString1*, * *WcString2*;

wchar_t *wcscpy(*WcString1*, *WcString2*)
wchar_t * *WcString1*, * *WcString2*;

size__t wcscspn(*WcString1*, *WcString2*)
wchar_t * *WcString1*, * *WcString2*;

Description

The **wcscat**, **wcschr**, **wcscmp**, **wcscpy** and **wcscspn** subroutines operate on null terminated **wchar_t** strings. The string arguments to these subroutines are expected to contain a **wchar_t** null character marking the end of the string. Boundary checking is not done when a copy or concatenation operation is performed.

The **wcscat** subroutine appends a copy of the string pointed to by the *WcString2* parameter to the end of the string pointed to by the *WcString1* parameter and returns the value of *WcString1*.

The **wcschr** subroutine returns a pointer to the first occurrence of the *WideCharacter* parameter in the *WcString* parameter. The character value may be a **wchar_t null** character (\0); the **wchar_t null** character at the end of the string is included in the search. If the character is not found, a **NULL** pointer is returned.

The **wcscmp** subroutine compares two **wchar_t** strings. It returns an integer greater than zero if the *WcString1* parameter is greater than the *WcString2* parameter. It returns zero if the two strings are equivalent. It returns a number less than zero if *WcString1* is less than *WcString2*.

The **wcscpy** subroutine copies the contents of the *WcString2* parameter (including the ending **wchar_t null** character) into the *WcString1* parameter and returns the value of *WcString1*.

The **wcscspn** subroutine computes the number of **wchar_t** characters in the initial segment of the string pointed to by the *WcString1* parameter that do not appear in the string pointed to by the *WcString2* parameter. The **wcscspn** subroutine returns the number of **wchar_t** characters in the segment.

Implementation Specifics

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

Related Information

The **mbscat, mbscmp**, **mbscpy** subroutines, **wcsncat**, **wcsncmp**, **wcsncpy** subroutines.

National Language Support Overview in *General Programming Concepts*.

wcslen Subroutine

Purpose

Determines the number of characters in a wide–character string.

Library

Standard C Library

Syntax

#include <wcstr.h>

size_t wcslen(*Wcstring*)
wchar_t **Wcstring*;

Description

The **wcslen** subroutine computes the number of **wchar_t** characters in the string pointed to by the *Wcstring* parameter.

Parameter

Wcstring	A wide character string.

Return Value

The **wcslen** subroutine returns the number of **wchar_t** characters that precede the terminating **wchar_t null** character.

Implementation Specifics

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

Related Information

The **mbslen** subroutine.

National Language Support Overview in *General Programming Concepts.*

wcsncat, wcsncmp, or wcsncpy Subroutine

Purpose

Performs operations on a specified number of wide–characters from one string to another.

Library

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

Syntax

#include <wcstr.h>

wchar_t *wcsncat(*WcString1, WcString2, Number***)**
wchar_t **WcString1*, **WcString2*;
size_t *Number*;

wchar_t *wcsncmp(*WcString1*, *WcString2*, *Number***)**
wchar_t **WcString1*, **WcString2*;
size_t *Number*;

wchar_t *wcsncpy(*WcString1*, *WcString2*, *Number***)**
wchar_t **WcString1*, **WcString2*;
size_t *Number*;

Description

The **wcsncat, wcsncmp** and **wcsncpy** subroutines operate on null terminated wide–character strings.

The **wcsncat** subroutine appends up to the value of the *Number* parameter in characters from the *WcString2* parameter to the end of the *WcString1* parameter, appends a **wchar_t null** to the result, and returns *WcString1*.

The **wcsncmp** subroutine compares up to the value of the *Number* parameter in wide–characters in the *WcString1* parameter to the *WcString2* parameter and returns an integer greater than zero if *WcString1* is greater than *WcString2*; zero if the strings are equivalent; and an integer less than zero if *WcString1* is less than *WcString2*.

The **wcsncpy** subroutine copies up to the value of the *Number* parameter in wide–characters from the *WcString2* parameter to the *WcString1* parameter and returns *WcString1*. If *WcString2* is shorter than *Number* characters, *WcString1* is padded out to *Number* characters with **wchar_t null** characters

Implementation Specifics

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

Related Information

The **mbsncat**, **mbsncmp**, **mbsncpy** subroutines, **wcscat**, **wcschr**, **wcscmp**, **wcscpy**, **wcscspn** subroutines.

National Language Support Overview in *General Programming Concepts.*

wcspbrk Subroutine

Purpose

Locates the first occurrence of characters in a string.

Library

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

Syntax

#include <wcstr.h>

wchar_t *wcspbrk(*Wcs1*, *Wcs2***)**
wchar_t **Wcs1*, *Wcs2*;

Description

The **wcspbrk** subroutine locates the first occurrence in the string pointed to by the *Wcs1* parameter of any character from the string pointed to by the *Wcs2* parameter.

Parameters

Wcs1	Pointer to a string being searched.
Wcs2	Pointer to a set of characters string.

Return Values

The **wcspbrk** subroutine returns a pointer to the character, or **NULL** if no **wchar_t** from the *Wcs2* parameter occurs in the *Wcs1* parameter.

Implementation Specifics

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

Related Information

The **mbspbrk** subroutine, **wcswcs** subroutine.

National Language Support Overview in *General Programming Concepts.*

wcsrchr Subroutine

Purpose

Locates a **wchar_t** character in a wide–character string.

Library

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

Syntax

#include <wcstr.h>

wchar_t *wcsrchr(*WcString*, *WideCharacter***)**
wchar_t * *WcString*, *WideCharacter*;

Description

The **wcsrchr** subroutine locates the last occurrence of the *WideCharacter* parameter in the string pointed to by the *WcString* parameter. The terminating **wchar_t null** character is considered to be part of the string.

Parameters

WcString	Pointer to a string.
WideCharacter	A **wchar_t** character.

Return Values

The **wcsrchr** subroutine returns a pointer to the *WideCharacter* parameter, or a **NULL** pointer if *WideCharacter* does not occur in the string.

Implementation Specifics

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

Related Information

The **mbschr** subroutines, **mbsrchr** subroutine.

National Language Support Overview in *General Programming Concepts.*

wcsspn Subroutine

Purpose

Returns the number of wide–characters in the initial segment of a string.

Library

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

Syntax

#include <wcstr.h>

size_t wcsspn(*WcString1*, *WcString2*)
whcar_t **WcString1*, **WcString2*;

Description

The **wcsspn** subroutine computes the number of **wchar_t** characters in the initial segment of the string pointed to by the *WcString1* parameter. The *WcString1* parameter consists entirely of **wchar_t** characters from the string pointed to by the *WcString2* parameter.

Parameters

WcString1	Pointer to the initial segment of a string.
WcString2	Pointer to a set of characters string.

Return Values

The **wcsspn** subroutine returns the number of **wchar_t** characters in the segment.

Implementation Specifics

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

wcstombs Subroutine

Purpose

Converts a sequence of wide–characters into a sequence of multibyte characters

Library

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

Syntax

#include <stdlib.h>

size_t wcstombs(*String*, *WcString*, *Number***)**
char **String*;
wchar_t * *WcString*;
size_t *Number*;

Description

The **wcstombs** subroutine converts the sequence of wide–characters pointed to by the *WcString* parameter to a sequence of corresponding multibyte characters and places the results in the area pointed to by the *String* parameter. The conversion is terminated when the wide–character null is encountered or when the value of the *Number* parameter (or *Number*–1) in bytes have been placed in the area pointed to by the *String* parameter. If the amount of space available in the area pointed to by *String* would cause a malformed (partial) multibyte character to be stored, only *Number*–1 bytes would be used because only valid (complete) multibyte characters are allowed.

Parameters

String	Pointer to area where result of conversion is stored.
WcString	Pointer to a wide–character string.
Number	Number of bytes to be converted.

Return Values

The **wcstombs** subroutine returns the number of bytes modified. If an invalid wide–character is encountered a –1 is returned.

Implementation Specifics

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

Related Information

The **wctomb** subroutine, **mbtowc** subroutine, **mbstowcs** subroutine.

National Language Support Overview in *General Programming Concepts.*

wcswcs Subroutine

Purpose

Locates first occurrence of a wide–character in a string.

Library

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

Syntax

#include <wcstr.h>

wchar_t *wcswcs(*WcString1*, *WcString2*)
wchar_t * *WcString1*, * *WcString2*;

Description

The **wcswcs** subroutine locates the first occurrence in the string pointed to by the *WcString1* parameter of the sequence of **wchar_t** characters (excluding the terminating **wchar_t null** character) in the string pointed to by the *WcString2* parameter.

Parameters

WcString1	Pointer to a wide character string being searched.
WcString2	Pointer to a wide character string, which is source string.

Return Values

The **wcswcs** subroutine returns a pointer to the located string or **NULL** if the string is not found. If the *WcString2* parameter points to a string with zero length, the function returns the *WcString1* parameter.

Implementation Specifics

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

Related Information

The **mbspbrk** subroutine, **wcspbrk** subroutine.

National Language Support Overview in *General Programming Concepts*

wctomb Subroutine

Purpose

Converts a wide–character into a multibyte character.

Library

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

Syntax

#include <stdlib.h>

size_t wctomb(*Storage*, *WideCharacter***)**
char **Storage*;
wchar_t *WideCharacter*;

Description

The **wctomb** subroutine determines the number of bytes required to represent the wide–character whose value is the *WideCharacter* parameter as the corresponding multibyte character and converts *WideCharacter* to a multibyte character and stores the results in the area pointed by the *Storage* parameter.

Parameters

Storage	Pointer to an area where result of conversion is stored.
WideCharacter	A wide character value.

Return Values

The **wctomb** subroutine returns a 0 if the *Storage* parameter is a **NULL** pointer. If the *WideCharacter* parameter does not correspond to a valid multibyte character a –1 is returned. Otherwise, the number of bytes that comprise the multibyte character is returned.

Implementation Specifics

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

Related Information

The **wcstombs** subroutine, **mbstowcs** subroutine, **mbtowc** subroutine.

National Language Support Overview in *General Programming Concepts*.

write, writex, writev, or writevx Subroutine

Purpose

Writes to a file.

Syntax

int write(*FileDescriptor*, *Buffer*, *NBytes*)
int *FileDescriptor*;
char **Buffer*;
unsigned int *NBytes*;

int writex(*FileDescriptor, Buffer, NBytes, Extension*)
int *FileDescriptor*;
char **Buffer*;
unsigned int *NBytes*;
int *Extension*;

#include <sys/types.h>
#include <sys/uio.h>

int writev(*FileDescriptor*, *iov*, *iovCount*)
int *FileDescriptor*;
struct iovec **iov*;
int *iovCount*;

int writevx (*FileDescriptor*, *iov*, *iovCount*, *Extension*)
int *FileDescriptor*;
struct iovec **iov*;
int *iovCount*;
int *Extension*;

Description

The **write** subroutine attempts to write *NBytes* of data to the file associated with the *FileDescriptor* parameter from the buffer pointed to by the *Buffer* parameter.

The **writev** subroutine performs the same action but gathers the output data from the *iovCount* buffers specified by the array of **iovec** structures pointed to by the *iov* parameter. Each **iovec** entry specifies the base address and length of an area in memory from which data should be written. The **writev** subroutine always writes a complete area before proceeding to the next.

The **writex** and **writevx** subroutines are the same as **write** and **writev**, respectively, with the addition of an *Extension* parameter, which is used when writing to some device drivers.

With regular files and devices capable of seeking, the actual writing of data proceeds from the position in the file indicated by the file pointer. Upon return from the **write** subroutine, the file pointer increments by the number of bytes actually written.

With devices incapable of seeking, writing always takes place starting at the current position. The value of a file pointer associated with such a device is undefined.

Fewer bytes can be written than requested if there is not enough room to satisfy the request. In this case the number of bytes written is returned. The next attempt to write a nonzero number of bytes fails (except as noted in the following text). The limit reached can be either the **ulimit** or the end of the physical medium.

Successful completion of a **write** subroutine clears the SetUserID and SetGroupID attributes unless the calling process has root user authority.

If the **O_APPEND** flag of the file status is set, the file offset is set to the end of the file prior to each write.

If the *FileDescriptor* parameter refers to a regular file whose file status flags specify **O_SYNC**, this is a synchronous update (as described in the **open** subroutine).

If the *FileDescriptor* parameter refers to a regular file that some process has opened with **O_DEFER**, the data and file size is not updated on permanent storage until some process issues an **fsync** subroutine or performs a synchronous update. If all processes that have the file open with **O_DEFER** close the file before any process issues an **fsync** subroutine or performs a synchronous update, the data and file size is not updated on permanent storage.

Write requests to a pipe (or FIFO) are handled the same as a regular file with the following exceptions:

- There is no file offset associated with a pipe; hence each write request appends to the end of the pipe.
- If the size of the **write** request is less than or equal to the value of the PIPE_BUF system variable (described in the **pathconf** routine), the **write** subroutine is guaranteed to be atomic. The data is not interleaved with data from other processes doing writes on the same pipe. Writes of greater than PIPE_BUF bytes can have data interleaved, on arbitrary boundaries, with writes by other processes, whether or not **O_DELAY** or **O_NONBLOCK** are set.
- If **O_NDELAY** and **O_NONBLOCK** are clear (the default), a **write** request to a full pipe causes the process to block until enough space becomes available to handle the entire request.
- If **O_NDELAY** is set, a **write** to a full pipe returns zero.
- If **O_NONBLOCK** is set, a **write** to a full pipe returns a value of –1 and sets the global variable **errno** to **EAGAIN**.

When attempting to write to a regular file that supports enforcement mode record locks, and all or part of the region to be written is currently locked by another process:

- If **O_NDELAY** and **O_NONBLOCK** are clear (the default), the calling process blocks until the lock is released.
- If **O_NDELAY** or **O_NONBLOCK** is set, then the **write** subroutine returns a value of –1 and sets the global variable **errno** to **EAGAIN**.

The **fcntl** subroutine provides more information about record locks.

The behavior of an interrupted **write** subroutine depends on how the handler for the arriving signal was installed:

Note: A write to a regular file is not interruptible. Only writes to objects that may block indefinitely, such as FIFOs, sockets, and some devices, are generally interruptible.

- If the handler was installed with an indication that subroutines should not be restarted, the **write** subroutine returns a value of –1 and set the global variable **errno** to **EINTR** (even if some data was already written).

- If the handler was installed with an indication that subroutines should be restarted:
  - if no data had been written when the interrupt was handled, the **write** subroutine will not return a value (it is restarted).
  - if data had been written when the interrupt was handled, this **write** subroutine returns the amount of data already written.

Parameters

FileDescriptor Identifies the object to which the data is to be written.

Buffer Identifies the buffer containing the data to be written.

NBytes Specifies the number of bytes to write.

iov Points to an array of **iovec** structures, which identifies the buffers containing the data to be written. The **iovec** structure is defined in the **sys/uio.h** header file and contains the following members:

```
caddr_t  iov_base;
int      iov_len;
```

iovCount Specifies the number of **iovec** structures pointed to by the *iov* parameter.

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.

Return Values

Upon successful completion, the **write**, **writex**, **writev**, **and writevx** subroutines return the number of bytes that were actually written. Otherwise, the value –1 is returned and the global variable **errno** is set to indicate the error.

Error Codes

The **write**, **writex**, **writev**, and **writevx** subroutines fail when one or more of the following are true:

EBADF The *FileDescriptor* parameter does not specify a valid file descriptor open for writing.

EINVAL The file position pointer associated with the *FileDescriptor* parameter was negative.

EINVAL The *iovCount* parameter value was not between 1 and 16, inclusive.

EINVAL One of the iov_len values in the **iov** array was negative or the sum overflowed a 32–bit integer.

EFAULT The *Buffer* parameter or part of the *iov* parameter points to a location outside of the allocated address space of the process.

EPIPE An attempt was made to write to a file that is not opened for reading by any process, or to a socket or type **SOCK_STREAM** that is not connected to a peer socket.

EAGAIN The **O_NONBLOCK** flag is set on this file and the process would be delayed in the write operation.

EAGAIN An enforcement mode record lock is outstanding in the portion of the file that is to be written.

EFBIG An attempt was made to write a file that exceeds the maximum file size.

ENOSPC No free space is left on the file system containing the file.

EINTR A signal was caught during the write operation, and the signal handler was installed with an indication that subroutines are not to be restarted.

EIO An I/O error occurred while writing to the file system.

Implementation Specifics

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

Related Information

The **fcntl** subroutine, **ioctl** subroutine, **lockfx** subroutine, **lseek** subroutine, **open** subroutine, **pipe** subroutine, **poll** subroutine, **select** subroutine.

The **fcntl.h** header file, **sys/uio.h** header file.

wsprintf Subroutine

Purpose

Prints formatted output.

Library

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

Syntax

#include <stdio.h>

int wsprintf (*String,Format***[,** *Value*, **...])**
**wchar_t ****String*;
**char ****Format*;

Description

The **wsprintf** subroutine converts, formats, and stores its *Value* parameters, 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 '\0' (null character) at the end. It is your responsibility to 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 **sprintf** subroutine, except that **wsprintf** uses a **wchar_t** string *String.*

Parameters

String Specifies a **wchar_t** string.

Format Specifies a character string that contains plain characters, which are copied to the output stream, and conversion specifications, each of which causes zero or more items to be fetched from the *Value* parameter list. If there are not enough items for *Format* in the *Value* parameter list, the results are unpredictable. If more *Value*s remain after the entire *Format* has been processed, they are ignored.

Value Specifies the input to the *Format* parameter.

Return Values

Upon successful completion, the **wsprintf** subroutine returns the number of display characters in the output string rather than the number of bytes in the string. The **wsprintf** subroutine uses strings that can contain 2–byte **wchars**. The value returned by **wsprintf** does not include the final '\0' character. If an output error occurs, a negative value is returned.

Implementation Specifics

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

Related Information

The **conv** subroutine, **ecvt**, **fcvt**, **gcvt** subroutines, **printf**, **fprintf**, **sprintf**, **NLprintf**, **NLfprintf**, **NLsprintf** subroutines, **putc**, **putchar**, **fputc**, **putw**, **putwc**, **putwchar**, **fputwc** subroutines, **scanf**, **fscanf**, **sscanf**, **NLscanf**, **NLfscanf**, **NLsscanf** subroutines.

National Language Support Overview in *General Programming Concepts*.

wsscanf Subroutine

Purpose

Converts formatted input.

Library

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

Syntax

#include <stdio.h>

int wsscanf (*String*, *Format* **[**, *Pointer* **...])**
**wchar_t ***String*;
**char ***Format*;

Description

The **wsscanf** subroutine reads character data, interprets it according to a format, and stores the converted results into specified memory locations. If there are insufficient arguments for the format, the behavior is undefined. If the format is exhausted while arguments remain, the excess arguments are evaluated but otherwise ignored.

This subroutine is the same as the **scanf** subroutine, except that the **wsscanf** subroutine reads its input from the **wchar_t** string *String*.

Parameters

String — Specifies a **wchar_t** string.

Pointer — Specifies where to store the interpreted data.

Format — Contains conversion specifications used to interpret the input. If there are insufficient arguments for the *Format*, the behavior is undefined. If the *Format* is exhausted while arguments remain, the excess arguments are evaluated as always but are otherwise ignored.

Return Values

The **wsscanf** subroutine returns the number of successfully matched and assigned input items. This number can be 0 if there was an early conflict between an input character and the control string. If the input ends before the first conflict or conversion, only **EOF** is returned.

Implementation Specifics

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

Related Information

The **atof**, **atoff**, **strtod**, **strtof** subroutines, **getc**, **getchar**, **getw**, **getwc**, **fgetwc**, **getwchar** subroutines, **printf**, **fprintf**, **sprintf**, **NLprintf**, **NLfprintf**, **NLsprintf** subroutines, **scanf**, **fscanf**, **sscanf**, **NLscanf**, **NLfscanf**, **NLsscanf** subroutines, **strtol**, **strtoul**, **atol**, **atoi** subroutines.

National Language Support Overview in *General Programming Concepts*.

wstring Subroutines

Purpose

Perform operations on wide character strings.

Library

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

Syntax

```
#include <wstring.h>
wchar_t *wstrcat (Xstring1, Xstring2)
wchar_t *Xstring1, *Xstring2;

wchar_t *wstrncat (Xstring1, Xstring2, Number)
wchar_t *Xstring1, *Xstring2;
int Number;

int wstrcmp (Xstring1, Xstring2)
wchar_t *Xstring1, *Xstring2;

int wstrncmp (Xstring1, Xstring2, Number)
wchar_t *Xstring1, *Xstring2;
int Number;

wchar_t *wstrcpy (Xstring1, Xstring2)
wchar_t *Xstring1, *Xstring2;

wchar_t *wstrncpy (Xstring1, Xstring2, Number)
wchar_t *Xstring1, *Xstring2;
int Number;

int wstrlen (Xstring)
wchar_t *Xstring;

wchar_t *wstrchr (Xstring, Number)
wchar_t *Xstring;
int Number;

wchar_t *wstrrchr (Xstring, Number)
wchar_t *Xstring;
int Number;

wchar_t *wstrpbrk (Xstring1, Xstring2)
wchar_t *Xstring1, Xstring2;

int wstrspn (Xstring1, Xstring2)
wchar_t *Xstring1, Xstring2;

int wstrcspn (Xstring1, Xstring2)
wchar_t *Xstring1, Xstring2;

wchar_t *wstrtok (Xstring1, Xstring2)
wchar_t *Xstring1, Xstring2;
```

wchar_t *wstrdup (*Xstring1*)
wchar_t **Xstring1*;

Description

The **wstring** subroutines copy, compare, and append strings in memory, and determine location, size, and existence of strings in memory. For these subroutines, a string is an array of **wchar_ts**, terminated by a null character. The **wstring** subroutines parallel the **string** subroutines, but operate on strings of type **wchar_t** rather than on type **char**, except as specifically noted below.

The parameters *Xstring1*, *Xstring2* and *Xstring* point to strings of type **wchar_t** (arrays of **wchars** terminated by a **wchar_t** null character).

The subroutines **wstrcat**, **wstrncat**, **wstrcpy**, and **wstrncpy** all alter the *Xstring1* parameter. They do not check for overflow of the array pointed to by *Xstring1*. All string movement is performed wide character by wide character. Overlapping moves toward the left work as expected, but overlapping moves to the right may give unexpected results. All of these subroutines are declared in the **wstring.h** header file.

The **wstrcat** subroutine appends a copy of the **wchar_t** string pointed to by the *Xstring2* parameter to the end of the **wchar_t** string pointed to by the *Xstring1* parameter. The **wstrcat** subroutine returns a pointer to the null–terminated result.

The **wstrncat** subroutine copies, at most, the value of the *Number* parameter of **wchar_ts** in the *Xstring2* parameter to the end of the **wchar_t** string pointed to by the *Xstring1* parameter. Copying stops before *Number* **wchar_ts** if a null character is encountered in the string pointed to by the *Xstring2* parameter. The **wstrncat** subroutine returns a pointer to the null–terminated result.

The **wstrcmp** subroutine lexicographically compares the **wchar_t** string pointed to by the *Xstring1* parameter to the **wchar_t** string pointed to by the *Xstring2* parameter. The **wstrcmp** subroutine returns a value that is:

Less than 0 if *Xstring1* is less than *Xstring2*

Equal to 0 if *Xstring1* is equal to *Xstring2*

Greater than 0 if *Xstring1* is greater than *Xstring2*.

The **wstrncmp** subroutine makes the same comparison as **wstrcmp**, but it compares, at most, the value of the *Number* parameter of pairs of **wchars**. The comparisons are based on collation values as determined by the locale category **LC_COLLATE** and the **LANG** variable.

The **wstrcpy** subroutine copies the string pointed to by the *Xstring2* parameter to the array pointed to by the *Xstring1* parameter. Copying stops when the **wchar_t** nul is copied. The **wstrcpy** subroutine returns the value of the *Xstring1* parameter.

The **wstrncpy** subroutine copies the value of the *Number* parameter of **wchar_ts** from the string pointed to by the *Xstring2* parameter to the **wchar_t** array pointed to by the *Xstring1* parameter. If *Xstring2* is less than *Number* **wchar_ts** long, then **wstrncpy** pads *Xstring1* with trailing null characters to fill *Number* **wchar_ts**. If *Xstring2* is *Number* or more **wchar_ts** long, then only the first *Number* **wchar_ts** are copied; the result is not terminated with a null character. The **wstrncpy** subroutine returns the value of the *Xstring1* parameter.

The **wstrlen** subroutine returns the number of **wchar_ts** in the string pointed to by the *Xstring* parameter, not including the terminating **wchar_t** nul.

The **wstrchr** subroutine returns a pointer to the first occurrence of the **wchar_t** specified by the *Number* parameter in the **wchar_t** string pointed to by the *Xstring* parameter. A **NULL** pointer is returned if the **wchar_t** does not occur in the **wchar_t** string. The **wchar_t** nul that terminates a string is considered to be part of the **wchar_t** string.

The **wstrrchr** subroutine returns a pointer to the last occurrence of the character specified by the *Number* parameter in the **wchar_t** string pointed to by the *Xstring* parameter. A **NULL** pointer is returned if the **wchar_t** does not occur in the **wchar_t** string. The **wchar_t** null that terminates a string is considered to be part of the **wchar_t** string.

The **wstrpbrk** subroutine returns a pointer to the first occurrence in the **wchar_t** string pointed to by the *Xstring1* parameter of any code point from the string pointed to by the *Xstring2* parameter. A **NULL** pointer is returned if no character matches.

The **wstrspn** subroutine returns the length of the initial segment of the string pointed to by the *Xstring1* parameter that consists entirely of code points from the **wchar_t** string pointed to by the *Xstring2* parameter.

The **wstrcspn** subroutine returns the length of the initial segment of the **wchar_t** string pointed to by the *Xstring1* parameter that consists entirely of code points **not** from the **wchar_t** string pointed to by the *Xstring2* parameter.

The **wstrtok** subroutine returns a pointer to an occurrence of a text token in the string pointed to by the *Xstring1* parameter. The *Xstring2* parameter specifies a set of code points as token delimiters. If the *Xstring1* parameter is anything other than **NULL**, then the **wstrtok** subroutine reads the string pointed to by the *Xstring1* parameter until it finds one of the delimiter code points specified by the *Xstring2* parameter. It then stores a **wchar_t** null into the **wchar_t** string, replacing the delimiter code point, and returns a pointer to the first **wchar_t** of the text token. The **wstrtok** subroutine keeps track of its position in the **wchar_t** string so that subsequent calls with a **NULL** *Xstring1* parameter step through the **wchar_t** string. The delimiters specified by the *Xstring2* parameter can be changed for subsequent calls to **wstrtok**. When no tokens remain in the **wchar_t** string pointed to by the *Xstring1* parameter, the **wstrtok** subroutine returns a **NULL** pointer.

The **wstrdup** subroutine returns a pointer to a **wchar_t** string that is a duplicate of the **wchar_t** string to which the *Xstring1* parameter points. Space for the new string is allocated using the **malloc** subroutine. When a new string cannot be created, a **NULL** pointer is returned.

Implementation Specifics

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

Related Information

The **NCchar** subroutines, **NLstring** subroutines, **NLstrtime** subroutines, **NCstring** subroutines, **malloc** subroutine and **string** subroutines.

National Language Support Overview in *General Programming Concepts.*

wstrtod or watof Subroutine

Purpose

Converts an **NLchar** string to a double-precision floating-point in Japanese Language Support.

Library

Standard C Library

Japanese Language Support Syntax

#include <wstring.h>

double wstrtod(*String*, *Pointer***)**
NLchar **String, **Pointer*;

double watof(*String***)**
NLchar **String*

Description

The **wstrtod** subroutine recognizes a string that starts with any number of white-space characters (defined by the **ctype** subroutine **isspace**), followed by an optional sign, a string of decimal digits that may include a decimal point, e or E, an optional sign or space, and an integer. You can use either ASCII characters or SJIS kanji characters, but you cannot mix them in the same string.

When the value of *Pointer* is not (NLchar **) NULL, a pointer to the search terminating character is returned to the address indicated by *Pointer*. When the resulting number cannot be created, **Pointer* is set to *String* and 0 is returned.

The **watof** (*String*) subroutine functions like the **wstrtod** (*String* (NLchar **) NULL).

Parameters

String — Specifies the address of the string to scan.

Pointer — Specifies the address at which the pointer to the terminating character is stored.

For Japanese Language Support

When Japanese Language Support is installed on your system, the **wstrtod** subroutine returns a double-precision floating-point number that is converted from an NLchar string pointed to by the *String* parameter. The system searches the *String* until it finds the first unrecognized character. You can use either ASCII characters or SJIS kanji characters, but you cannot mix them in the same string.

Error Codes

When the value causes overflow, HUGE_VAL (defined in the **math.h** file) is returned with the appropriate sign, and the global variable **errno** is set to ERANGE. When the value causes underflow, 0 is returned and the global variable **errno** is set to ERANGE.

Implementation Specifics

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

Related Information

The **scanf, fscanf**, **sscanf**, **NLscanf**, **NLfscanf**, **NLsscanf** subroutines, **atof, atoff, strtod, strtof** subroutines, **strtol, strtoul, atol, atoi** subroutines, **wstrtol, watol, watoi** subroutines.

wstrtol, watol, or watoi Subroutine

Purpose

Converts an **NLchar** string to an integer in Japanese Language Support.

Library

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

Syntax

#include <wstring.h>

long wstrtol(*String*, *Pointer*, *Base*)
NLchar **String*, ***Pointer*;
int *Base*;

long watol(*String*)
NLchar **String*;

int watoi(*String*)
NLchar **String*;

Description

When Japanese Language Support is installed on your system, the **wstrtol** subroutine returns a long integer that is converted from the string pointed to by the *String* parameter. The string is searched until a character is found that is inconsistent with *Base*. Leading white-space characters defined by the ctype subroutine **isspace** are ignored.

You can use either ASCII characters or SJIS kanji characters, but you cannot mix them in the same string.

When the value of *Pointer* is not (NLchar **) NULL, a pointer to the terminating character is returned to the address indicated by *Pointer*. When an integer cannot be created, the address indicated by *Pointer* is set to *String*, and 0 is returned.

When the value of *Base* is positive and not greater than 36, that value is used as the base during conversion. Leading zeros that follow an optional leading sign are ignored. When the value of *Base* is 16, 0x and 0X are ignored.

When the value of *Base* is 0, the system chooses an appropriate base after examining the actual string. An optional sign followed by a leading zero signifies octal, and a leading 0x or 0X signifies hexadecimal. In all other cases, the subroutines assume a decimal base.

Truncation from **long** to **int** occurs by assignment, and also by explicit casting.

The **watol(***String*) subroutine functions like **wstrtol(***String*, (NLchar **) NULL, 10).

The **watoi(***String*) subroutine functions like (**int**) **wstrtol** (*String*, (NLchar **) NULL, 10).

Note: Even if overflow occurs, it is ignored.

Parameters

String	Specifies the address of the string to scan.
Pointer	Specifies the address at which the pointer to the terminating character is stored.

Base	Specifies an integer value used as the base during conversion.

Implementation Specifics

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

Related Information

The **scanf, fscanf**, **sscanf**, **NLscanf**, **NLfscanf**, **NLsscanf** subroutines, **atof, atoff, strtod, strtof** subroutines, **strtol, strtoul, atol, atoi** subroutines, **wstrtod, watof** subroutines.

FORTRAN Basic Linear Algebra Subroutines (BLAS)

SDOT or DDOT Function

Purpose

Returns the dot product of two vectors.

Library

BLAS Library (**libblas.a**)

FORTRAN Syntax

REAL FUNCTION	**SDOT(***N,X,INCX,Y,INCY***)**
INTEGER	*INCX,INCY,N*
REAL	*X*(*),*Y*(*)
DOUBLE PRECISION FUNCTION	**DDOT(***N,X,INCX,Y,INCY***)**
INTEGER	*INCX,INCY,N*
DOUBLE PRECISION	*X*(*),*Y*(*)

Description

The **SDOT** or **DDOT** function returns the dot product of vectors *X* and *Y*.

Parameters

N	On entry, *N* specifies the number of elements in *X* and *Y*. Unchanged on exit.
X	Vector of dimension at least (1 + (*N*–1) * abs(*INCX*)). Unchanged on exit.
INCX	On entry, *INCX* specifies the increment for the elements of *X*. Unchanged on exit.
Y	Vector of dimension at least (1 + (*N*–1) * abs(*INCY*)). Unchanged on exit.
INCY	On entry, *INCY* specifies the increment for the elements of *Y*. Unchanged on exit.

Error Codes

For values of *N* <= 0, a value of 0 is returned.

CDOTC or ZDOTC Function

Purpose

Returns the complex dot product of two vectors, conjugating the first.

Library

BLAS Library (**libblas.a**)

FORTRAN Syntax

COMPLEX FUNCTION	**CDOTC(***N,X,INCX,Y,INCY***)**
INTEGER	*INCX,INCY,N*
COMPLEX	*X*(*),*Y*(*)

DOUBLE COMPLEX FUNCTION	**ZDOTC**(*N,X,INCX,Y,INCY*)
INTEGER	*INCX,INCY,N*
COMPLEX*16	*X*(*),*Y*(*)

Description

The **CDOTC** or **ZDOTC** function returns the complex dot product of two vectors, conjugating the first.

Parameters

N — On entry, *N* specifies the number of elements in *X* and *Y*. Unchanged on exit.

X — Vector of dimension at least (1 + (*N*–1) * abs(*INCX*)). Unchanged on exit.

INCX — On entry, *INCX* specifies the increment for the elements of *X*. Unchanged on exit.

Y — Vector of dimension at least (1 + (*N*–1) * abs(*INCY*)). Unchanged on exit.

INCY — On entry, *INCY* specifies the increment for the elements of *Y*. Unchanged on exit.

Error Codes

For values of *N* <= 0, a value of 0 is returned.

CDOTU or ZDOTU Function

Purpose

Returns the complex dot product of two vectors.

Library

BLAS Library (**libblas.a**)

FORTRAN Syntax

COMPLEX FUNCTION	**CDOTU**(*N,X,INCX,Y,INCY*)
INTEGER	*INCX,INCY,N*
COMPLEX	*X*(*),*Y*(*)

DOUBLE COMPLEX FUNCTION	**ZDOTU**(*N,X,INCX,Y,INCY*)
INTEGER	*INCX,INCY,N*
COMPLEX*16	*X*(*),*Y*(*)

Description

The **CDOTU** or **ZDOTU** function returns the complex dot product of two vectors.

Parameters

N — On entry, *N* specifies the number of elements in *X* and *Y*. Unchanged on exit.

X — Vector of dimension at least (1 + (*N*–1) * abs(*INCX*)). Unchanged on exit.

INCX	On entry, *INCX* specifies the increment for the elements of *X*. Unchanged on exit.
Y	Vector of dimension at least (1 + (*N*–1) * abs(*INCY*)). Unchanged on exit.
INCY	On entry, *INCY* specifies the increment for the elements of *Y*. Unchanged on exit.

Error Codes

For values of *N* <= 0, a value of 0 is returned.

SAXPY, DAXPY, CAXPY or ZAXPY Subroutine

Purpose

Computes a constant times a vector plus a vector.

Library

BLAS Library (**libblas.a**)

FORTRAN Syntax

```
SUBROUTINE        SAXPY(N,A,X,INCX,Y,INCY)
INTEGER           INCX,INCY,N
REAL              A
REAL              X (*),Y(*)

SUBROUTINE        DAXPY(N,A,X,INCX,Y,INCY)
INTEGER           INCX,INCY,N
DOUBLE PRECISION  A
DOUBLE PRECISION  X (*),Y(*)

SUBROUTINE        CAXPY(N,A,X,INCX,Y,INCY)
INTEGER           INCX,INCY,N
COMPLEX           A
COMPLEX           X (*),Y(*)

SUBROUTINE        ZAXPY(N,A,X,INCX,Y,INCY)
INTEGER           INCX,INCY,N
COMPLEX*16        A
COMPLEX*16        X (*),Y(*)
```

Description

The **SAXPY**, **DAXPY**, **CAXPY** or **ZAXPY** subroutine computes a constant times a vector plus a vector:

$Y = A * X + Y$

Parameters

N	On entry, *N* specifies the number of elements in *X* and *Y*. Unchanged on exit.
A	On entry, *A* contains a constant to be multiplied by the *X* vector. Unchanged on exit.

X	Vector of dimension at least (1 + (*N*–1) * abs(*INCX*)). Unchanged on exit.
INCX	On entry, *INCX* specifies the increment for the elements of *X*. Unchanged on exit.
Y	Vector of dimension at least (1 + (*N*–1) * abs(*INCY*)). The result is returned in vector *Y*.
INCY	On entry, *INCY* specifies the increment for the elements of *Y*. Unchanged on exit.

Error Codes

If *SA* = 0 or *N* <= 0, the subroutine returns immediately.

SROTG, DROTG, CROTG or ZROTG Subroutine

Purpose

Constructs Givens plane rotation.

Library

BLAS Library (**libblas.a**)

FORTRAN Syntax

SUBROUTINE	**SROTG**(*A,B,C,S*)
REAL	*A,B,C,S*
SUBROUTINE	**DROTG**(*A,B,C,S*)
DOUBLE PRECISION	*A,B,C,S*
SUBROUTINE	**CROTG**(*A,B,C,S*)
REAL	*C*
COMPLEX	*A,B,S*
SUBROUTINE	**ZROTG**(*A,B,C,S*)
DOUBLE PRECISION	*C*
COMPLEX*16	*A,B,S*

Description

Given vectors *A* and *B*, the **SROTG**, **DROTG**, **CROTG** or **ZROTG** subroutine computes:

$$a = \frac{A}{|A| + |B|}, \quad b = \frac{B}{|A| + |B|}$$

$$roe = \begin{cases} a & \text{if } |A| > |B| \\ b & \text{if } |B| >= |A| \end{cases} \quad r = roe\,(a^2 + b^2)^{1/2},$$

$$C = \begin{cases} A/r & \text{if } r \text{ not } = 0 \\ 1 & \text{if } r = 0 \end{cases} \qquad S = \begin{cases} B/r & \text{if } r \text{ not } = 0 \\ 0 & \text{if } r = 0 \end{cases}$$

The numbers *C*, *S*, and *r* then satisfy the matrix equation:

```
 __     __     __    __     __    __
| C    S  |   | A    |   | r    |
|         | . |      | = |      |
| -S   C  |   | B    |   | 0    |
 __     __     __    __     __    __
```

The subroutines also compute:

```
    { S   if |A| > |B|,
z = { 1/C if |B| >= |A| and C not = 0,
    { 1   if C = 0.
```

The subroutines return *r* overwriting *A* and *z* overwriting *B*, as well as returning *C* and *S*.

Parameters

A	On entry, contains a scalar constant. On exit, contains the value *r*.
B	On entry, contains a scalar constant. On exit, contains the value *z*.
C	Can contain any value on entry. Value of *C* returned on exit.
S	Can contain any value on entry. Value of *S* returned on exit.

SROT, DROT, CSROT or ZDROT Subroutine

Purpose

Applies a plane rotation.

Library

BLAS Library (**libblas.a**)

FORTRAN Syntax

SUBROUTINE	**SROT**(*N,X,INCX,Y,INCY,C,S*)
INTEGER	*INCX,INCY,N*
REAL	*C,S*
REAL	*X(*),Y(*)*

SUBROUTINE	**DROT**(*N,X,INCX,Y,INCY,C,S*)
INTEGER	*INCX,INCY,N*
DOUBLE PRECISION	*C,S*
DOUBLE PRECISION	*X(*),Y(*)*

SUBROUTINE	**SROT**(*N,X,INCX,Y,INCY,C,S*)
INTEGER	*INCX,INCY,N*
REAL	*C,S*
COMPLEX	*X(*),Y(*)*

SUBROUTINE	**ZDROT**(*N,X,INCX,Y,INCY,C,S*)
INTEGER	*INCX,INCY,N*
DOUBLE PRECISION	*C,S*
COMPLEX*16	*X(*),Y(*)*

Description

The **SROT, DROT, CSROT** or **ZDROT** subroutine computes:

```
 ___   ___         ___     ___        ___   ___
|  X    |         |  C   S   |       |  X    |
|   i   |         |          |       |   i   |
|       |   :=    |          |   .   |       |   for i = 1, ..., N.
|  Y    |         |          |       |  Y    |
|   i   |         | –S   C   |       |   i   |
 ___   ___         ___     ___        ___   ___
```

The subroutines return the modified *X* and *Y*.

Parameters

N	On entry, *N* specifies the number of elements in *X* and *Y*. Unchanged on exit.
X	Vector of dimension at least (1 + (*N*–1) * abs (*INCX*)). Unchanged on exit.
INCX	On entry, *INCX* specifies the increment for the elements of *X*. Unchanged on exit.
Y	Vector of dimension at least (1 + (*N*–1) * abs(*INCY*)). Modified on exit.
INCY	On entry, *INCY* specifies the increment for the elements of *Y*. Unchanged on exit.
C	Scalar constant. Unchanged on exit.
S	Scalar constant. Unchanged on exit.

Error Codes

If *N* <= 0, or if *C* = 1 and *S* = 0, the subroutines return immediately.

SCOPY, DCOPY, CCOPY or ZCOPY Subroutine

Purpose

Copies vector *X* to *Y*.

Library

BLAS Library (**libblas.a**)

FORTRAN Syntax

SUBROUTINE	**SCOPY**(*N,X,INCX,Y,INCY*)
INTEGER	*INCX,INCY,N*
REAL	*X*(*), *Y*(*)
SUBROUTINE	**DCOPY**(*N,X,INCX,Y,INCY*)
INTEGER	*INCX,INCY,N*
DOUBLE PRECISION	*X*(*), *Y*(*)
SUBROUTINE	**CCOPY**(*N,X,INCX,Y,INCY*)
INTEGER	*INCX,INCY,N*
COMPLEX	*X*(*), *Y*(*)

SUBROUTINE	**ZCOPY**(*N,X,INCX,Y,INCY*)
INTEGER	*INCX,INCY,N*
COMPLEX*16	*X(*),Y(*)*

Description

The **SCOPY, DCOPY, CCOPY** or **ZCOPY** subroutine copies vector *X* to vector *Y*.

Parameters

N	On entry, *N* specifies the number of elements in *X* and *Y*. Unchanged on exit.
X	Vector of dimension at least (1 + (*N*–1) * abs(*INCX*)). Unchanged on exit.
INCX	On entry, *INCX* specifies the increment for the elements of *X*. Unchanged on exit.
Y	Vector of dimension at least (1 + (*N*–1) * abs(*INCY*)) or greater. Can contain any values on entry. On exit, contains the same values as *X*.
INCY	On entry, *INCY* specifies the increment for the elements of *Y*. Unchanged on exit.

Error Codes

For values of *N* <= 0, the subroutines return immediately.

SSWAP, DSWAP, CSWAP or ZSWAP Subroutine

Purpose

Interchanges vectors *X* and *Y*.

Library

BLAS Library (**libblas.a**)

FORTRAN Syntax

SUBROUTINE	**SSWAP**(*N,X,INCX,Y,INCY*)
INTEGER	*INCX,INCY,N*
REAL	*X(*),Y(*)*
SUBROUTINE	**DSWAP**(*N,X,INCX,Y,INCY*)
INTEGER	*INCX,INCY,N*
DOUBLE PRECISION	*X(*),Y(*)*
SUBROUTINE	**CSWAP**(*N,X,INCX,Y,INCY*)
INTEGER	*INCX,INCY,N*
COMPLEX	*X(*),Y(*)*
SUBROUTINE	**ZSWAP**(*N,X,INCX,Y,INCY*)
INTEGER	*INCX,INCY,N*
COMPLEX*16	*X(*),Y(*)*

Description

The **SSWAP**, **DSWAP**, **CSWAP** or **ZSWAP** subroutine interchanges vector *X* and vector *Y*.

Parameters

N	On entry, *N* specifies the number of elements in *X* and *Y*. Unchanged on exit.
X	Vector of dimension at least (1 + (*N*–1) * abs(*INCX*)). On exit, contains the elements of vector *Y*.
INCX	On entry, *INCX* specifies the increment for the elements of *X*. Unchanged on exit.
Y	Vector of dimension at least (1 + (*N*–1) * abs(*INCY*)). On exit, contains the elements of vector *X*.
INCY	On entry, *INCY* specifies the increment for the elements of *Y*. Unchanged on exit.

Error Codes

For values of *N* <= 0, the subroutines return immediately.

SNRM2, DNRM2, SCNRM2 or DZNRM2 Function

Purpose

Computes the Euclidean length of the *N*–vector stored in *X*() with storage increment *INCX*.

Library

BLAS Library (**libblas.a**)

FORTRAN Syntax

REAL FUNCTION	**SNRM2**(*N,X,INCX*)
INTEGER	*INCX,N*
REAL	*X*(*)
DOUBLE PRECISION FUNCTION	**DNRM2**(*N,X,INCX*)
INTEGER	*INCX,N*
DOUBLE PRECISION	*X*(*)
REAL FUNCTION	**SCNRM2**(*N,X,INCX*)
INTEGER	*INCX,N*
COMPLEX	*X*(*)
DOUBLE PRECISION FUNCTION	**DZNRM2**(*N,X,INCX*)
INTEGER	*INCX,N*
COMPLEX*16	*X*(*)

Description

The **SNRM2, DNRM2, SCNRM2** or **DZNRM2** function returns the Euclidean norm of the *N*–vector stored in *X*() with storage increment *INCX*.

Parameters

N	On entry, *N* specifies the number of elements in *X* and *Y*. Unchanged on exit.

X	Vector of dimension at least (1 + (*N*–1) * abs(*INCX*)). Unchanged on exit.
INCX	On entry, *INCX* specifies the increment for the elements of *X*. *INCX* must be greater than zero. Unchanged on exit.

Error Codes

For values of *N* <= 0, a value of 0 is returned.

SASUM, DASUM, SCASUM or DZASUM Function

Purpose

Returns the sum of absolute values of vector components.

Library

BLAS Library (**libblas.a**)

FORTRAN Syntax

REAL FUNCTION	**SASUM**(*N*,*X*,*INCX*)
INTEGER	*INCX*,*N*
REAL	*X*(*)
DOUBLE PRECISION FUNCTION	**DASUM**(*N*,*X*,*INCX*)
INTEGER	*INCX*,*N*
DOUBLE PRECISION	*X*(*)
REAL FUNCTION	**SCASUM**(*N*,*X*,*INCX*)
INTEGER	*INCX*,*N*
COMPLEX	*X*(*)
DOUBLE PRECISION FUNCTION	**DZASUM**(*N*,*X*,*INCX*)
INTEGER	*INCX*,*N*
COMPLEX*16	*X*(*)

Description

The **SASUM**, **DASUM**, **SCASUM** or **DZASUM** function returns the sum of absolute values of vector components.

Parameters

N	On entry, *N* specifies the number of elements in *X* and *Y*. Unchanged on exit.
X	Vector of dimension at least (1 + (*N*–1) * abs(*INCX*)). Unchanged on exit.
INCX	On entry, *INCX* specifies the increment for the elements of *X*. *INCX* must be greater than zero. Unchanged on exit.

Error Codes

For values of *N* <= 0, a value of 0 is returned.

SSCAL, DSCAL, CSSCAL, CSCAL, ZDSCAL or ZSCAL Subroutine

Purpose

Scales a vector by a constant.

Library

BLAS Library (**libblas.a**)

FORTRAN Syntax

SUBROUTINE	**SSCAL(***N,A,X,INCX***)**
INTEGER	*INCX,N*
REAL	*A*
REAL	*X*(*)

SUBROUTINE	**DSCAL(***N,A,X,INCX***)**
INTEGER	*INCX,N*
DOUBLE PRECISION	*A*
DOUBLE PRECISION	*X*(*)

SUBROUTINE	**CSSCAL(***N,A,X,INCX***)**
INTEGER	*INCX,N*
REAL	*A*
COMPLEX	*X*(*)

SUBROUTINE	**CSCAL**
INTEGER	*INCX,N*
COMPLEX	*A*
COMPLEX	*X*(*)

SUBROUTINE	**ZDSCAL**
INTEGER	*INCX,N*
DOUBLE PRECISION	*A*
COMPLEX*16	*X*(*)

SUBROUTINE	**ZSCAL(**
INTEGER	*INCX,N*
COMPLEX*16	*A*
COMPLEX*16	*X*(*)

Description

The **SSCAL**, **DSCAL**, **CSSCAL**, **CSCAL**, **ZDSCAL** or **ZSCAL** subroutine scales a vector by a constant:

```
X := X * A
```

Parameters

N On entry, *N* specifies the number of elements in *X* and *Y*. Unchanged on exit.

A Scaling constant. Unchanged on exit.

X Vector of dimension at least (1 + (*N*–1) * abs(*INCX*)). On exit, contains the scaled vector.

INCX On entry, *INCX* specifies the increment for the elements of *X*. *INCX* must be greater than zero. Unchanged on exit.

Error Codes

For values of *N* <= 0, the subroutines return immediately.

ISAMAX, IDAMAX, ICAMAX or IZAMAX Function

Purpose

Finds the index of element having maximum absolute value.

Library

BLAS Library (**libblas.a**)

FORTRAN Syntax

INTEGER FUNCTION	**ISAMAX**(*N,X,INCX*)
INTEGER	*INCX,N*
REAL	*X*(*)
INTEGER FUNCTION	**IDAMAX**(*N,X,INCX*)
INTEGER	*INCX,N*
DOUBLE PRECISION	*X*(*)
INTEGER FUNCTION	**ICAMAX**(*N,X,INCX*)
INTEGER	*INCX,N*
COMPLEX	*X*(*)
INTEGER FUNCTION	**IZAMAX**(*N,X,INCX*)
INTEGER	*INCX,N*
COMPLEX*16	*X*(*)

Description

The **ISAMAX, IDAMAX, ICAMAX** or **IZAMAX** function returns the index of element having maximum absolute value.

Parameters

N On entry, *N* specifies the number of elements in *X* and *Y*. Unchanged on exit.

X Vector of dimension at least (1 + (*N*–1) * abs(*INCX*)). Unchanged on exit.

INCX On entry, *INCX* specifies the increment for the elements of *X*. *INCX* must be greater than zero. Unchanged on exit.

Error Codes

For values of *N* <= 0, a value of 0 is returned.

SDSDOT Function

Purpose

Returns the dot product of two vectors plus a constant.

Library

BLAS Library (**libblas.a**)

FORTRAN Syntax

REAL FUNCTION	**SDSDOT**(*N,B,X,INCX,Y,INCY*)
INTEGER	*N,INCX,INCY*
REAL	*B,X*(*),*Y*(*)

Description

The **SDSDOT** function computes the sum of constant *B* and dot product of vectors *X* and *Y*.

Parameters

N — On entry, *N* specifies the number of elements in *X* and *Y*. Unchanged on exit.

B — Scalar. Unchanged on exit.

X — Vector of dimension at least (1 + (*N*–1) * abs(*INCX*)). Unchanged on exit.

INCX — On entry, *INCX* specifies the increment for the elements of *X*. *INCX* must be greater than zero. Unchanged on exit.

Y — Vector of dimension at least (1 + (*N*–1) * abs(*INCY*)). Unchanged on exit.

INCY — On entry, *INCY* specifies the increment for the elements of *Y*. *INCY* must be greater than zero. Unchanged on exit.

Error Codes

For values of *N* <= 0, the subroutine returns immediately.

Implementation Specifics

Computation is performed in double precision.

SROTM or DROTM Subroutine

Purpose

Applies the modified Givens transformation.

Library

BLAS Library (**libblas.a**)

FORTRAN Syntax

SUBROUTINE	**SROTM**(*N,X,INCX,Y,INCY,PARAM*)
INTEGER	*N,INCX,INCY*
REAL	*X*(*), *Y*(*), *PARAM*(**5**)

SUBROUTINE	**DROTM**(*N,X,INCX,Y,INCY,PARAM*)
INTEGER	*N,INCX,INCY*
DOUBLE PRECISION	*X*(*), *Y*(*), *PARAM*(**5**)

Description

Let H denote the modified Givens transformation defined by the parameter array *PARAM*. The **SROTM** or **DROTM** subroutine computes:

$$\begin{bmatrix} x \\ y \end{bmatrix} := H * \begin{bmatrix} x \\ y \end{bmatrix}$$

where H is is a 2 x 2 matrix with the components defined by the elements of the array *PARAM* as follows:

if *PARAM*(1) == 0.0
H(1,1) = H(2,2) = 1.0
H(2,1) = *PARAM*(3)
H(1,2) = *PARAM*(4)

if *PARAM*(1) == 1.0
H(1,2) = H(2,1) = –1.0
H(1,1) = *PARAM*(2)
H(2,2) = *PARAM*(5)

if *PARAM*(1) == –1.0
H(1,1) = *PARAM*(2)
H(2,1) = *PARAM*(3)
H(1,2) = *PARAM*(4)
H(2,2) = *PARAM*(5)

Parameters

N — On entry, *N* specifies the number of elements in *X* and *Y*. Unchanged on exit.

X — Vector of dimension at least (1 + (*N*–1) * abs(*INCX*)). On exit, modified as described above.

INCX — On entry, *INCX* specifies the increment for the elements of *X*. *INCX* must be greater than zero. Unchanged on exit.

Y — Vector of dimension at least (1 + (*N*–1) * abs(*INCY*)). On exit, modified as described above.

INCY — On entry, *INCY* specifies the increment for the elements of *Y*. *INCY* must be greater than zero. Unchanged on exit.

PARAM — Vector of dimension (5). On entry, must be set as described above. Specifically, *PARAM*(1) is a flag and must have value of either 0.0, –1.0, 1.0, or 2.0. Unchaged on exit.

Implementation Specifics

If *N* <= 0 or H is an identity matrix, the subroutines return immediately.

Related information

The **SROTMG** or **DROTMG** subroutine builds the *PARAM* array prior to use by the **SROTM** or **DROTM** subroutine.

SROTMG or DROTMG Subroutine

Purpose

Constructs a modified Givens transformation.

Library

BLAS Library (**libblas.a**)

FORTRAN Syntax

SUBROUTINE	**SROTMG(***D1,D2,X1,X2,PARAM***)**
REAL	*D1,D2,X1,X2,PARAM***(5)**
SUBROUTINE	**DROTMG(***D1,D2,X1,X2,PARAM***)**
DOUBLE PRECISION	*D1,D2,X1,X2,PARAM***(5)**

Description

The **SROTMG** or **DROTMG** subroutine constructs a modified Givens transformation. The input quantities *D1*, *D2*, *X1*, and *X2* define a 2–vector in partitioned form:

```
 ___  ___     ___           ___   ___   ___
|  a1  |     |  sqrt(D1)   0   | |   X1   |
|      |  =  |                 | |        |
|  a2  |     |   0   sqrt(D2)  | |   X2   |
 ___  ___     ___           ___   ___   ___
```

The subroutines determine the modified Givens rotation matrix H that transforms *X2*, and thus a2, to zero. A representation of this matrix is stored in the array *PARAM* as follows. Locations in *PARAM* not listed are left unchanged.

Case 1: *PARAM*(1) = 1.0
PARAM(2) = H(1,1)
PARAM(5) = H(2,2)

Case 2: *PARAM*(1) = 0.0
PARAM(3) = H(2,1)
PARAM(4) = H(1,2)

Case 3: *PARAM*(1) = 1.0
H(1,1) = *PARAM*(2)
H(2,1) = *PARAM*(3)
H(1,2) = *PARAM*(4)
H(2,2) = *PARAM*(5)

Case 4: *PARAM*(1) = –2.0
H = I (Identity matrix)

Parameters

D1 — Non–negative scalar. Modified on exit to reflect the results of the transformation.

D2 — Scalar. Can be negative on entry. Modified on exit to reflect the results of the transformation.

X1 — Scalar. Modified on exit to reflect the results of the transformation.

X2 — Scalar. Unchanged on exit.

PARAM — Vector of dimension (5). Values on entry are unused. Modified on exit as described above.

Related Information

The **SROTM** and **DROTM** subroutines apply the Modified Givens Transformation.

SGEMV, DGEMV, CGEMV or ZGEMV Subroutine

Purpose

Performs matrix–vector operation with general matrices.

Library

BLAS Library (**libblas.a**)

FORTRAN Syntax

SUBROUTINE	**SGEMV(***TRANS,M,N,ALPHA,A,LDA,X,INCX,BETA,Y,INCY***)**
REAL	*ALPHA,BETA*
INTEGER	*INCX,INCY,LDA,M,N*
CHARACTER*1	*TRANS*
REAL	*A(LDA,*),X(*),Y(*)*
SUBROUTINE	**DGEMV(***TRANS,M,N,ALPHA,A,LDA,X,INCX,BETA,Y,INCY***)**
DOUBLE PRECISION	*ALPHA,BETA*
INTEGER	*INCX,INCY,LDA,M,N*
CHARACTER*1	*TRANS*
DOUBLE PRECISION	*A(LDA,*),X(*),Y(*)*
SUBROUTINE	**CGEMV(***TRANS,M,N,ALPHA,A,LDA,X,INCX,BETA,Y,INCY***)**
COMPLEX	*ALPHA,BETA*
INTEGER	*INCX,INCY,LDA,M,N*
CHARACTER*1	*TRANS*
COMPLEX	*A(LDA,*),X(*),Y(*)*
SUBROUTINE	**ZGEMV(***TRANS,M,N,ALPHA,A,LDA,X,INCX,BETA,Y,INCY***)**
COMPLEX*16	*ALPHA,BETA*
INTEGER	*INCX,INCY,LDA,M,N*
CHARACTER*1	*TRANS*
COMPLEX*16	*A(LDA,*),X(*),Y(*)*

Description

The **SGEMV, DGEMV, CGEMV** or **ZGEMV** subroutine performs one of the matrix–vector operations:

y := alpha * *A* * x + beta * y

or

y := alpha * *A'* * x + beta * y

where alpha and beta are scalars, x and y are vectors and *A* is an *M* by *N* matrix.

Parameters

TRANS On entry, *TRANS* specifies the operation to be performed as follows:

TRANS = 'N' or 'n'
y := alpha * *A* * x + beta * y

TRANS = 'T' or 't'
y := alpha * *A'* * x + beta * y

TRANS = 'C' or 'c'
y := alpha * *A*' * x + beta * y

Unchanged on exit.

M	On entry, *M* specifies the number of rows of the matrix *A*. *M* must be at least zero. Unchanged on exit.
N	On entry, *N* specifies the number of columns of the matrix *A*. *N* must be at least zero. Unchanged on exit.
ALPHA	On entry, *ALPHA* specifies the scalar alpha. Unchanged on exit.
A	An array of dimension (*LDA*, *N*). On entry, the leading *M* by *N* part of the array *A* must contain the matrix of coefficients. Unchanged on exit.
LDA	On entry, *LDA* specifies the first dimension of *A* as declared in the calling (sub) program. *LDA* must be at least max(1, *M*). Unchanged on exit.
X	A vector of dimension at least (1 + (*N*–1) * abs(*INCX*)) when *TRANS* = 'N' or 'n' and at least (1 + (*M*–1) * abs(*INCX*)) otherwise. On entry, the incremented array *X* must contain the vector x. Unchanged on exit.
INCX	On entry, *INCX* specifies the increment for the elements of *X*. *INCX* must not be zero. Unchanged on exit.
BETA	On entry, *BETA* specifies the scalar beta. When *BETA* is supplied as zero then *Y* need not be set on input. Unchanged on exit.
Y	A vector of dimension at least 1 + (*M*–1) * abs(*INCY*)) when *TRANS* = 'N' or 'n' and at least (1 + (*N*–1) * abs(*INCY*)) otherwise. On entry, with *BETA* nonzero, the incremented array *Y* must contain the vector y. On exit, *Y* is overwritten by the updated vector y.
INCY	On entry, *INCY* specifies the increment for the elements of *Y*. *INCY* must not be zero. Unchanged on exit.

SGBMV, DGBMV, CGBMV or ZGBMV Subroutine

Purpose

Performs matrix–vector operations with general banded matrices.

Library

BLAS Library (**libblas.a**)

FORTRAN Syntax

SUBROUTINE	**SGBMV(***TRANS,M,N,KL,KU,ALPHA,A,LDA,X,INCX,BETA, Y,INCY***)**
REAL	*ALPHA,BETA*
INTEGER	*INCX,INCY,KL,KU,LDA,M,N*
CHARACTER*1	*TRANS*
REAL	*A(LDA,*),X(*),Y(*)*

Level 2: matrix–vector operations

SUBROUTINE	**DGBMV**(*TRANS,M,N,KL,KU,ALPHA,A,LDA,X,INCX,BETA, Y,INCY*)
DOUBLE PRECISION	*ALPHA,BETA*
INTEGER	*INCX,INCY,KL,KU,LDA,M,N*
CHARACTER*1	*TRANS*
DOUBLE PRECISION	*A(LDA,*),X(*),Y(*)*
SUBROUTINE	**CGBMV**(*TRANS,M,N,KL,KU,ALPHA,A,LDA,X,INCX,BETA, Y,INCY*)
COMPLEX	*ALPHA,BETA*
INTEGER	*INCX,INCY,KL,KU,LDA,M,N*
CHARACTER*1	*TRANS*
COMPLEX	*A(LDA,*),X(*),Y(*)*
SUBROUTINE	**ZGBMV**(*TRANS,M,N,KL,KU,ALPHA,A,LDA,X,INCX,BETA, Y,INCY*)
COMPLEX*16	*ALPHA,BETA*
INTEGER	*INCX,INCY,KL,KU,LDA,M,N*
CHARACTER*1	*TRANS*
COMPLEX*16	*A(LDA,*),X(*),Y(*)*

Description

The **SGBMV, DGBMV, CGBMV** or **ZGBMV** subroutine performs one of the matrix–vector operations:

y := alpha * *A* * x + beta * y

or

y := alpha * *A'* * x + beta * y

where alpha and beta are scalars, x and y are vectors and *A* is an *M* by *N* band matrix, with *KL* sub–diagonals and *KU* super–diagonals.

Parameters

TRANS
: On entry, *TRANS* specifies the operation to be performed as follows:

  TRANS = 'N' or 'n'
  y := alpha * *A* * x + beta * y

  TRANS = 'T' or 't'
  y := alpha * *A'* * x + beta * y

  TRANS = 'C' or 'c'
  y := alpha * *A'* * x + beta * y

  Unchanged on exit.

M
: On entry, *M* specifies the number of rows of the matrix *A*. *M* must be at least zero. Unchanged on exit.

N
: On entry, *N* specifies the number of columns of the matrix *A*. *N* must be at least zero. Unchanged on exit.

KL
: On entry, *KL* specifies the number of sub–diagonals of the matrix *A*. *KL* must satisfy 0 .le. *KL*. Unchanged on exit.

KU — On entry, *KU* specifies the number of super–diagonals of the matrix *A*. *KU* must satisfy 0 .le. *KU*. Unchanged on exit.

ALPHA — On entry, *ALPHA* specifies the scalar alpha. Unchanged on exit.

A — A vector of dimension (*LDA*, *N*). On entry, the leading (*KL* + *KU* + 1) by n part of the array *A* must contain the matrix of coefficients, supplied column by column, with the leading diagonal of the matrix in row (*KU* + 1) of the array, the first super–diagonal starting at position 2 in row *KU*, the first sub–diagonal starting at position 1 in row (*KU* + 2), and so on. Elements in the array *A* that do not correspond to elements in the band matrix (such as the top left *KU* by *KU* triangle) are not referenced. The following program segment will transfer a band matrix from conventional full matrix storage to band storage:

```
DO 20, J = 1, N
   K = KU + 1 - J
   DO 10, I = MAX( 1, J - KU ), MIN( M, J + KL )
      A( K + I, J ) = matrix( I, J )
10 CONTINUE
20 CONTINUE
```

Unchanged on exit.

LDA — On entry, *LDA* specifies the first dimension of *A* as declared in the calling (sub) program. *LDA* must be at least (*KL* + *KU* + 1). Unchanged on exit.

X — A vector of dimension at least (1 + (*N*–1) * abs(*INCX*)) when *TRANS* = 'N' or 'n' and at least (1 + (*M*–1) * abs(*INCX*)) otherwise. On entry, the incremented array *X* must contain the vector x. Unchanged on exit.

INCX — On entry, *INCX* specifies the increment for the elements of *X*. *INCX* must not be zero. Unchanged on exit.

BETA — On entry, *BETA* specifies the scalar beta. When *BETA* is supplied as zero then *Y* need not be set on input. Unchanged on exit.

Y — A vector of dimension at least (1 + (*M*–1) * abs(*INCY*)) when *TRANS* = 'N' or 'n' and at least (1 + (*N*–1) * abs(*INCY*)) otherwise. On entry, the incremented array *Y* must contain the vector y. On exit, *Y* is overwritten by the updated vector y.

INCY — On entry, *INCY* specifies the increment for the elements of *Y*. *INCY* must not be zero. Unchanged on exit.

CHEMV or ZHEMV Subroutine

Purpose

Performs matrix–vector operations using hermitian matrices.

Library

BLAS Library (**libblas.a**)

FORTRAN Syntax

SUBROUTINE	**CHEMV**(*UPLO,N,ALPHA,A,LDA,X,INCX,BETA,Y,INCY*)
COMPLEX	*ALPHA,BETA*
INTEGER	*INCX,INCY,LDA,N*
CHARACTER*1	*UPLO*
COMPLEX	*A(LDA,*),X(*),Y(*)*
SUBROUTINE	**ZHEMV**(*UPLO, N, ALPHA, A, LDA, X, INCX, BETA, Y, INCY*)
COMPLEX*16	*ALPHA,BETA*
INTEGER	*INCX,INCY,LDA,N*
CHARACTER*1	*UPLO*
COMPLEX*16	*A(LDA,*),X(*),Y(*)*

Description

The **CHEMV** or **ZHEMV** subroutine performs the matrix–vector operation:

y := alpha * *A* * x + beta * y

where alpha and beta are scalars, x and y are *N* element vectors and *A* is an *N* by *N* hermitian matrix.

Parameters

UPLO
: On entry, *UPLO* specifies whether the upper or lower triangular part of the array *A* is to be referenced as follows:

  UPLO = 'U' or 'u'
  Only the upper triangular part of *A* is to be referenced.

  UPLO = 'L' or 'l'
  Only the lower triangular part of *A* is to be referenced.

  Unchanged on exit.

N
: On entry, *N* specifies the order of the matrix *A*. *N* must be at least zero. Unchanged on exit.

ALPHA
: On entry, *ALPHA* specifies the scalar alpha. Unchanged on exit.

A
: An array of dimension (*LDA*, *N*). On entry with *UPLO* = 'U' or 'u', the leading *N* by *N* upper triangular part of the array *A* must contain the upper triangular part of the hermitian matrix and the strictly lower triangular part of *A* is not referenced. On entry with *UPLO* = 'L' or 'l', the leading *N* by *N* lower triangular part of the array *A* must contain the lower triangular part of the hermitian matrix and the strictly upper triangular part of *A* is not referenced. Note that the imaginary parts of the diagonal elements need not be set and are assumed to be zero. Unchanged on exit.

LDA
: On entry, *LDA* specifies the first dimension of *A* as declared in the calling (sub) program. *LDA* must be at least max(1, *N*). Unchanged on exit.

X
: A vector of dimension at least (1 + (*N*–1) * abs(*INCX*)). On entry, the incremented array *X* must contain the *N* element vector x. Unchanged on exit.

INCX
: On entry, *INCX* specifies the increment for the elements of *X*. *INCX* must not be zero. Unchanged on exit.

BETA On entry, *BETA* specifies the scalar beta. When *BETA* is supplied as zero then *Y* need not be set on input. Unchanged on exit.

Y A vector of dimension at least (1 + (*N*–1) * abs(*INCY*)). On entry, the incremented array *Y* must contain the *N* element vector y. On exit, *Y* is overwritten by the updated vector y.

INCY On entry, *INCY* specifies the increment for the elements of *Y*. *INCY* must not be zero. Unchanged on exit.

CHBMV or ZHBMV Subroutine

Purpose

Performs matrix–vector operations using a hermitian band matrix.

Library

BLAS Library (**libblas.a**)

FORTRAN Syntax

SUBROUTINE	**CHBMV**(*UPLO,N,K,ALPHA,A,LDA,X,INCX,BETA,Y,INCY*)
COMPLEX	*ALPHA,BETA*
INTEGER	*INCX,INCY,K,LDA,N*
CHARACTER*1	*UPLO*
COMPLEX	*A(LDA,*),X(*),Y(*)*

SUBROUTINE	**ZHBMV**(*UPLO,N,K,ALPHA,A,LDA,X,INCX,BETA,Y,INCY*)
COMPLEX*16	*ALPHA,BETA*
INTEGER	*INCX,INCY,K,LDA,N*
CHARACTER*1	*UPLO*
COMPLEX*16	*A(LDA,*),X(*),Y(*)*

Description

The **CHBMV** or **ZHBMV** subroutine performs the matrix–vector operation:

y := alpha * *A* * x + beta * y

where alpha and beta are scalars, x and y are *N* element vectors and *A* is an *N* by *N* hermitian band matrix, with *K* super–diagonals.

Parameters

UPLO On entry, *UPLO* specifies whether the upper or lower triangular part of the band matrix *A* is being supplied as follows:

UPLO = 'U' or 'u'
The upper triangular part of *A* is being supplied.

UPLO = 'L' or 'l'
The lower triangular part of *A* is being supplied.

Unchanged on exit.

N On entry, *N* specifies the order of the matrix *A*. *N* must be at least zero. Unchanged on exit.

K — On entry, *K* specifies the number of super–diagonals of the matrix *A*. *K* must satisfy 0 .le. *K*. Unchanged on exit.

ALPHA — On entry, *ALPHA* specifies the scalar alpha. Unchanged on exit.

A — An array of dimension (*LDA*, *N*). On entry with *UPLO* = 'U' or 'u', the leading (*K* + 1) by *N* part of the array *A* must contain the upper triangular band part of the hermitian matrix, supplied column by column, with the leading diagonal of the matrix in row (*K* + 1) of the array, the first super–diagonal starting at position 2 in row *K*, and so on. The top left *K* by *K* triangle of the array *A* is not referenced. The following program segment will transfer the upper triangular part of a hermitian band matrix from conventional full matrix storage to band storage:

```
DO 20, J = 1, N
    M = K + 1 - J
    DO 10, I = MAX( 1, J - K ), J
         A( M + I, J ) = matrix( I, J )
10 CONTINUE
20 CONTINUE
```

On entry with *UPLO* = 'L' or 'l', the leading (*K* + 1) by *N* part of the array *A* must contain the lower triangular band part of the hermitian matrix, supplied column by column, with the leading diagonal of the matrix in row 1 of the array, the first sub–diagonal starting at position 1 in row 2, and so on. The bottom right *K* by *K* triangle of the array *A* is not referenced. The following program segment will transfer the lower triangular part of a hermitian band matrix from conventional full matrix storage to band storage:

```
DO 20, J = 1, N
   M = 1 - J
   DO 10, I = J, MIN( N, J + K )
         A( M + I, J ) = matrix( I, J )
10 CONTINUE
20 CONTINUE
```

Note that the imaginary parts of the diagonal elements need not be set and are assumed to be zero. Unchanged on exit.

LDA — On entry, *LDA* specifies the first dimension of *A* as declared in the calling (sub) program. *LDA* must be at least (*K* + 1). Unchanged on exit.

X — A vector of dimension at least (1 + (*N*–1) * abs(*INCX*)). On entry, the incremented array *X* must contain the vector x. Unchanged on exit.

INCX — On entry, *INCX* specifies the increment for the elements of *X*. *INCX* must not be zero. Unchanged on exit.

BETA — On entry, *BETA* specifies the scalar beta. Unchanged on exit.

Y — A vector of dimension at least (1 + (*N*–1) * abs(*INCY*)). On entry, the incremented array *Y* must contain the vector y. On exit, *Y* is overwritten by the updated vector y.

INCY — On entry, *INCY* specifies the increment for the elements of *Y*. *INCY* must not be zero. Unchanged on exit.

CHPMV or ZHPMV Subroutine

Purpose

Performs matrix–vector operations using a packed hermitian matrix.

Library

BLAS Library (**libblas.a**)

FORTRAN Syntax

SUBROUTINE	**CHPMV(***UPLO,N,ALPHA,AP,X,INCX,BETA,Y,INCY***)**
COMPLEX	*ALPHA,BETA*
INTEGER	*INCX,INCY,N*
CHARACTER*1	*UPLO*
COMPLEX	*AP(*),X(*),Y(*)*

SUBROUTINE	**ZHPMV**
COMPLEX*16	*ALPHA,BETA*
INTEGER	*INCX,INCY,N*
CHARACTER*1	*UPLO*
COMPLEX*16	*AP(*),X(*),Y(*)*

Description

The **CHPMV** or **ZHPMV** subroutine performs the matrix–vector operation:

y := alpha * *A* * x + beta * y

where alpha and beta are scalars, x and y are *N* element vectors and *A* is an *N* by *N* hermitian matrix, supplied in packed form.

Parameters

UPLO
: On entry, *UPLO* specifies whether the upper or lower triangular part of the matrix *A* is supplied in the packed array *AP* as follows:

  UPLO = 'U' or 'u'
  : The upper triangular part of *A* is supplied in *AP*.

  UPLO = 'L' or 'l'
  : The lower triangular part of *A* is supplied in *AP*.

  Unchanged on exit.

N
: On entry, *N* specifies the order of the matrix *A*. *N* must be at least zero. Unchanged on exit.

ALPHA
: On entry, *ALPHA* specifies the scalar alpha. Unchanged on exit.

AP
: A vector of dimension at least ((*N* * (*N*+1))/2). On entry with *UPLO* = 'U' or 'u', the array *AP* must contain the upper triangular part of the hermitian matrix packed sequentially, column by column, so that *AP*(1) contains *A*(1,1), *AP*(2) and *AP*(3) contain *A*(1,2) and *A*(2,2) respectively, and so on. On entry with *UPLO* = 'L' or 'l', the array *AP* must contain the lower triangular part of the hermitian matrix packed sequentially, column by column, so that *AP*(1) contains *A*(1,1), *AP*(2) and *AP*(3) contain *A*(2,1) and

A(3,1) respectively, and so on. Note that the imaginary parts of the diagonal elements need not be set and are assumed to be zero. Unchanged on exit.

X	A vector of dimension at least (1 + (*N*–1) * abs(*INCX*)). On entry, the incremented array *X* must contain the *N* element vector x. Unchanged on exit.
INCX	On entry, *INCX* specifies the increment for the elements of *X*. *INCX* must not be zero. Unchanged on exit.
BETA	On entry, *BETA* specifies the scalar beta. When *BETA* is supplied as zero then *Y* need not be set on input. Unchanged on exit.
Y	A vector of dimension at least (1 + (*N*–1) * abs(*INCY*)). On entry, the incremented array *Y* must contain the *N* element vector y. On exit, *Y* is overwritten by the updated vector y.
INCY	On entry, *INCY* specifies the increment for the elements of *Y*. *INCY* must not be zero. Unchanged on exit.

SSYMV or DSYMV Subroutine

Purpose

Performs matrix–vector operations using a symmetric matrix.

Library

BLAS Library (**libblas.a**)

FORTRAN Syntax

```
SUBROUTINE      SSYMV(UPLO,N,ALPHA,A,LDA,X,INCX,BETA,Y,INCY)
REAL            ALPHA,BETA
INTEGER         INCX,INCY,LDA,N
CHARACTER*1     UPLO
REAL            A(LDA,*),X(*),Y(*)
```

```
SUBROUTINE           DSYMV(UPLO,N,ALPHA,A,LDA,X,INCX,BETA,Y,INCY)
DOUBLE PRECISION     ALPHA,BETA
INTEGER              INCX,INCY,LDA,N
CHARACTER*1          UPLO
DOUBLE PRECISION     A(LDA,*),X(*),Y(*)
```

Description

The **SSYMV** or **DSYMV** subroutine performs the matrix–vector operation:

y := alpha * *A* * x + beta * y

where alpha and beta are scalars, x and y are *N* element vectors and *A* is an *N* by *N* symmetric matrix.

Parameters

UPLO
: On entry, *UPLO* specifies whether the upper or lower triangular part of the array *A* is to be referenced as follows:

    UPLO = 'U' or 'u'
    Only the upper triangular part of *A* is to be referenced.

    UPLO = 'L' or 'l'
    Only the lower triangular part of *A* is to be referenced.

    Unchanged on exit.

N
: On entry, *N* specifies the order of the matrix *A*. *N* must be at least zero. Unchanged on exit.

ALPHA
: On entry, *ALPHA* specifies the scalar alpha. Unchanged on exit.

A
: An array of dimension (*LDA*, *N*). On entry with *UPLO* = 'U' or 'u', the leading *N* by *N* upper triangular part of the array *A* must contain the upper triangular part of the symmetric matrix and the strictly lower triangular part of *A* is not referenced. On entry with *UPLO* = 'L' or 'l', the leading *N* by *N* lower triangular part of the array *A* must contain the lower triangular part of the symmetric matrix and the strictly upper triangular part of *A* is not referenced. Unchanged on exit.

LDA
: On entry, *LDA* specifies the first dimension of *A* as declared in the calling (sub) program. *LDA* must be at least max(1, *N*). Unchanged on exit.

X
: A vector of dimension at least (1 + (*N*–1) * abs(*INCX*)). On entry, the incremented array *X* must contain the *N* element vector x. Unchanged on exit.

INCX
: On entry, *INCX* specifies the increment for the elements of *X*. *INCX* must not be zero. Unchanged on exit.

BETA
: On entry, *BETA* specifies the scalar beta. When *BETA* is supplied as zero then *Y* need not be set on input. Unchanged on exit.

Y
: A vector of dimension at least (1 + (*N*–1) * abs(*INCY*)). On entry, the incremented array *Y* must contain the *N* element vector y. On exit, *Y* is overwritten by the updated vector y.

INCY
: On entry, *INCY* specifies the increment for the elements of *Y*. *INCY* must not be zero. Unchanged on exit.

SSBMV or DSBMV Subroutine

Purpose

Performs matrix–vector operations using symmetric band matrix.

Library

BLAS Library (**libblas.a**)

FORTRAN Syntax

SUBROUTINE	**SSBMV**(*UPLO,N,K,ALPHA,A,LDA,X,INCX,BETA,Y,INCY*)
REAL	*ALPHA,BETA*
INTEGER	*INCX,INCY,K,LDA,N*
CHARACTER*1	*UPLO*
REAL	*A(LDA,*),X(*),Y(*)*
SUBROUTINE	**DSBMV**(*UPLO,N,K,ALPHA,A,LDA,X,INCX,BETA,Y,INCY*)
DOUBLE PRECISION	*ALPHA,BETA*
INTEGER	*INCX,INCY,K,LDA,N*
CHARACTER*1	*UPLO*
DOUBLE PRECISION	*A(LDA,*),X(*),Y(*)*

Description

The **SSBMV** or **DSBMV** subroutine performs the matrix–vector operation:

y := alpha * *A* * x + beta * y

where alpha and beta are scalars, x and y are *N* element vectors and *A* is an *N* by *N* symmetric band matrix, with *K* super–diagonals.

Parameters

UPLO
: On entry, *UPLO* specifies whether the upper or lower triangular part of the band matrix *A* is being supplied as follows:

  UPLO = 'U' or 'u'
  The upper triangular part of *A* is being supplied.

  UPLO = 'L' or 'l'
  The lower triangular part of *A* is being supplied.

  Unchanged on exit.

N
: On entry, *N* specifies the order of the matrix *A*. *N* must be at least zero. Unchanged on exit.

K
: On entry, *K* specifies the number of super–diagonals of the matrix *A*. *K* must satisfy 0 .le. *K*. Unchanged on exit.

ALPHA
: On entry, *ALPHA* specifies the scalar alpha. Unchanged on exit.

A
: An array of dimension (*LDA*, *N*). On entry with *UPLO* = 'U' or 'u', the leading (*K* + 1) by *N* part of the array *A* must contain the upper triangular band part of the symmetric matrix, supplied column by column, with the leading diagonal of the matrix in row (*K* + 1) of the array, the first super–diagonal starting at position 2 in row *K*, and so on. The top left *K* by *K* triangle of the array *A* is not referenced. The following program segment will transfer the upper triangular part of a symmetric band matrix from conventional full matrix storage to band storage:

```
DO 20, J = 1, N
      M = K + 1 – J
      DO 10, I = MAX( 1, J – K ), J
            A( M + I, J ) = matrix( I, J )
10 CONTINUE
20 CONTINUE
```

On entry with *UPLO* = 'L' or 'l', the leading (*K* + 1) by *N* part of the array *A* must contain the lower triangular band part of the symmetric matrix, supplied column by column, with the leading diagonal of the matrix in row 1 of the array, the first sub–diagonal starting at position 1 in row 2, and so on. The bottom right *K* by *K* triangle of the array *A* is not referenced. The following program segment will transfer the lower triangular part of a symmetric band matrix from conventional full matrix storage to band storage:

```
DO 20, J = 1, N
       M = 1 - J
       DO 10, I = J, MIN( N, J + K )
            A( M + I, J ) = matrix( I, J )
10 CONTINUE
20 CONTINUE
```

Unchanged on exit.

LDA On entry, *LDA* specifies the first dimension of *A* as declared in the calling (sub) program. *LDA* must be at least (*K* + 1). Unchanged on exit.

X A vector of dimension at least (1 + (*N*–1) * abs(*INCX*)). On entry, the incremented array *X* must contain the vector x. Unchanged on exit.

INCX On entry, *INCX* specifies the increment for the elements of *X*. *INCX* must not be zero. Unchanged on exit.

BETA On entry, *BETA* specifies the scalar beta. Unchanged on exit.

Y A vector of dimension at least (1 + (*N*–1) * abs(*INCY*)). On entry, the incremented array *Y* must contain the vector y. On exit, *Y* is overwritten by the updated vector y.

INCY On entry, *INCY* specifies the increment for the elements of *Y*. *INCY* must not be zero. Unchanged on exit.

SSPMV or DSPMV Subroutine

Purpose

Performs matrix–vector operations using a packed symmetric matrix.

Library

BLAS Library (**libblas.a**)

FORTRAN Syntax

SUBROUTINE	**SSPMV**(*UPLO,N,ALPHA,AP,X,INCX,BETA,Y,INCY*)
REAL	*ALPHA,BETA*
INTEGER	*INCX,INCY,N*
CHARACTER*1	*UPLO*
REAL	*AP(*),X(*),Y(*)*

SUBROUTINE	**DSPMV**(*UPLO,N,ALPHA,AP,X,INCX,BETA,Y,INCY*)
DOUBLE PRECISION	*ALPHA,BETA*
INTEGER	*INCX,INCY,N*

CHARACTER*1	*UPLO*
DOUBLE PRECISION	*AP*(*),*X*(*),*Y*(*)

Description

The **SSPMV** or **DSPMV** subroutine performs the matrix–vector operation:

y := alpha * *A* * x + beta * y

where alpha and beta are scalars, x and y are *N* element vectors and *A* is an *N* by *N* symmetric matrix, supplied in packed form.

Parameters

UPLO
: On entry, *UPLO* specifies whether the upper or lower triangular part of the matrix *A* is supplied in the packed array *AP* as follows:

UPLO = 'U' or 'u'
The upper triangular part of *A* is supplied in *AP*.

UPLO = 'L' or 'l'
The lower triangular part of *A* is supplied in *AP*.

Unchanged on exit.

N
: On entry, *N* specifies the order of the matrix *A*. *N* must be at least zero. Unchanged on exit.

ALPHA
: On entry, *ALPHA* specifies the scalar alpha. Unchanged on exit.

AP
: A vector of dimension at least ((*N* * (*N*+1))/2). On entry with *UPLO* = 'U' or 'u', the array *AP* must contain the upper triangular part of the symmetric matrix packed sequentially, column by column, so that *AP*(1) contains *A*(1,1), *AP*(2) and *AP*(3) contain *A*(1,2) and *A*(2,2) respectively, and so on. On entry with *UPLO* = 'L' or 'l', the array *AP* must contain the lower triangular part of the symmetric matrix packed sequentially, column by column, so that *AP*(1) contains *A*(1,1), *AP*(2) and *AP*(3) contain *A*(2,1) and *A*(3,1) respectively, and so on. Unchanged on exit.

X
: A vector of dimension at least (1 + (*N*–1) * abs(*INCX*)). On entry, the incremented array *X* must contain the *N* element vector x. Unchanged on exit.

INCX
: On entry, *INCX* specifies the increment for the elements of *X*. *INCX* must not be zero. Unchanged on exit.

BETA
: On entry, *BETA* specifies the scalar beta. When *BETA* is supplied as zero then *Y* need not be set on input. Unchanged on exit.

Y
: A vector of dimension at least (1 + (*N*–1) * abs(*INCY*)). On entry, the incremented array *Y* must contain the *N* element vector y. On exit, *Y* is overwritten by the updated vector y.

INCY
: On entry, *INCY* specifies the increment for the elements of *Y*. *INCY* must not be zero. Unchanged on exit.

STRMV, DTRMV, CTRMV or ZTRMV Subroutine

Purpose

Performs matrix–vector operations using a triangular matrix.

Library

BLAS Library (**libblas.a**)

FORTRAN Syntax

SUBROUTINE	**STRMV**(*UPLO,TRANS,DIAG,N,A,LDA,X,INCX*)
INTEGER	*INCX,LDA,N*
CHARACTER*1	*DIAG,TRANS,UPLO*
REAL	*A(LDA,*),X(*)*
SUBROUTINE	**DTRMV**(*UPLO,TRANS,DIAG,N,A,LDA,X,INCX*)
INTEGER	*INCX,LDA,N*
CHARACTER*1	*DIAG,TRANS,UPLO*
DOUBLE PRECISION	*A(LDA,*),X(*)*
SUBROUTINE	**CTRMV**(*UPLO,TRANS,DIAG,N,A,LDA,X,INCX*)
INTEGER	*INCX,LDA,N*
CHARACTER*1	*DIAG,TRANS,UPLO*
COMPLEX	*A(LDA,*),X(*)*
SUBROUTINE	**ZTRMV**(*UPLO,TRANS,DIAG,N,A,LDA,X,INCX*)
INTEGER	*INCX,LDA,N*
CHARACTER*1	*DIAG,TRANS,UPLO*
COMPLEX*16	*A(LDA,*),X(*)*

Description

The **STRMV**, **DTRMV**, **CTRMV** or **ZTRMV** subroutine performs one of the matrix–vector operations:

$$x := A * x$$

or

$$x := A' * x$$

where x is an *N* element vector and *A* is an *N* by *N* unit, or non–unit, upper or lower triangular matrix.

Parameters

UPLO
: On entry, *UPLO* specifies whether the matrix is an upper or lower triangular matrix as follows:

  UPLO = 'U' or 'u'
  : *A* is an upper triangular matrix.

  UPLO = 'L' or 'l'
  : *A* is a lower triangular matrix.

  Unchanged on exit.

TRANS
: On entry, *TRANS* specifies the operation to be performed as follows:

TRANS = 'N' or 'n'
x := *A* * x

TRANS = 'T' or 't'
x := *A*' * x

TRANS = 'C' or 'c'
x := *A*' * x

Unchanged on exit.

DIAG On entry, *DIAG* specifies whether or not *A* is unit triangular as follows:

DIAG = 'U' or 'u'
A is assumed to be unit triangular.

DIAG = 'N' or 'n'
A is not assumed to be unit triangular.

Unchanged on exit.

N On entry, *N* specifies the order of the matrix *A*. *N* must be at least zero. Unchanged on exit.

A An array of dimension (*LDA*, *N*). On entry with *UPLO* = 'U' or 'u', the leading *N* by *N* upper triangular part of the array *A* must contain the upper triangular matrix and the strictly lower triangular part of *A* is not referenced. On entry with *UPLO* = 'L' or 'l', the leading *N* by *N* lower triangular part of the array *A* must contain the lower triangular matrix and the strictly upper triangular part of *A* is not referenced. Note that when *DIAG* = 'U' or 'u', the diagonal elements of *A* are not referenced, but are assumed to be unity. Unchanged on exit.

LDA On entry, *LDA* specifies the first dimension of *A* as declared in the calling (sub) program. *LDA* must be at least max(1, *N*). Unchanged on exit.

X A vector of dimension at least (1 + (*N*–1) * abs(*INCX*)). On entry, the incremented array *X* must contain the *N* element vector x. On exit, *X* is overwritten with the tranformed vector x.

INCX On entry, *INCX* specifies the increment for the elements of *X*. *INCX* must not be zero. Unchanged on exit.

STBMV, DTBMV, CTBMV or ZTBMV Subroutine

Purpose

Performs matrix–vector operations using a triangular band matrix.

Library

BLAS Library (**libblas.a**)

FORTRAN Syntax

SUBROUTINE **STBMV(***UPLO,TRANS,DIAG,N,K,A,LDA,X,INCX***)**
INTEGER *INCX,K,LDA,N*

CHARACTER*1	*DIAG,TRANS,UPLO*
REAL	*A(LDA,*),X(*)*
SUBROUTINE	**DTBMV**(*UPLO,TRANS,DIAG,N,K,A,LDA,X,INCX*)
INTEGER	*INCX,K,LDA,N*
CHARACTER*1	*DIAG,TRANS,UPLO*
DOUBLE PRECISION	*A(LDA,*),X(*)*
SUBROUTINE	**CTBMV**(*UPLO,TRANS,DIAG,N,K,A,LDA,X,INCX*)
INTEGER	*INCX,K,LDA,N*
CHARACTER*1	*DIAG,TRANS,UPLO*
COMPLEX	*A(LDA,*),X(*)*
SUBROUTINE	**ZTBMV**(*UPLO,TRANS,DIAG,N,K,A,LDA,X,INCX*)
INTEGER	*INCX,K,LDA,N*
CHARACTER*1	*DIAG,TRANS,UPLO*
COMPLEX*16	*A(LDA,*),X(*)*

Description

The **STBMV**, **DTBMV**, **CTBMV** or **ZTBMV** subroutine performs one of the matrix–vector operations:

x := *A* * x

or

x := *A*' * x

where x is an *N* element vector and *A* is an *N* by *N* unit, or non–unit, upper or lower triangular band matrix, with (*K* + 1) diagonals.

Parameters

UPLO

On entry, *UPLO* specifies whether the matrix is an upper or lower triangular matrix as follows:

UPLO = 'U' or 'u'
A is an upper triangular matrix.

UPLO = 'L' or 'l'
A is a lower triangular matrix.

Unchanged on exit.

TRANS

On entry, *TRANS* specifies the operation to be performed as follows:

TRANS = 'N' or 'n'
x := *A* * x

TRANS = 'T' or 't'
x := *A*' * x

TRANS = 'C' or 'c'
x := *A*' * x

Unchanged on exit.

DIAG

On entry, *DIAG* specifies whether or not *A* is unit triangular as follows:

DIAG = 'U' or 'u'
A is assumed to be unit triangular.

DIAG = 'N' or 'n'
A is not assumed to be unit triangular.

Unchanged on exit.

N
On entry, *N* specifies the order of the matrix *A*. *N* must be at least zero. Unchanged on exit.

K
On entry with *UPLO* = 'U' or 'u', *K* specifies the number of super–diagonals of the matrix *A*. On entry with *UPLO* = 'L' or 'l', *K* specifies the number of sub–diagonals of the matrix *A*. *K* must satisfy 0 .le. *K*. Unchanged on exit.

A
An array of dimension (*LDA*, *N*). On entry with *UPLO* = 'U' or 'u', the leading (*K* + 1) by *N* part of the array *A* must contain the upper triangular band part of the matrix of coefficients, supplied column by column, with the leading diagonal of the matrix in row (*K* + 1) of the array, the first super–diagonal starting at position 2 in row *K*, and so on. The top left *K* by *K* triangle of the array *A* is not referenced. The following program segment will transfer an upper triangular band matrix from conventional full matrix storage to band storage:

```
DO 20, J = 1, N
       M = K + 1 - J
       DO 10, I = MAX( 1, J - K ), J
             A( M + I, J ) = matrix( I, J )
10 CONTINUE
20 CONTINUE
```

On entry with *UPLO* = 'L' or 'l', the leading (*K* + 1) by *N* part of the array *A* must contain the lower triangular band part of the matrix of coefficients, supplied column by column, with the leading diagonal of the matrix in row 1 of the array, the first sub–diagonal starting at position 1 in row 2, and so on. The bottom right *K* by *K* triangle of the array *A* is not referenced. The following program segment will transfer a lower triangular band matrix from conventional full matrix storage to band storage:

```
DO 20, J = 1, N
       M = 1 - J
       DO 10, I = J, MIN( N, J + K )
            A( M + I, J ) = matrix( I, J )
10 CONTINUE
20 CONTINUE
```

Note that when *DIAG* = 'U' or 'u' the elements of the array *A* corresponding to the diagonal elements of the matrix are not referenced, but are assumed to be unity. Unchanged on exit.

LDA
On entry, *LDA* specifies the first dimension of *A* as declared in the calling (sub) program. *LDA* must be at least (*K* + 1). Unchanged on exit.

X
A vector of dimension at least (1 + (*N*–1) * abs(*INCX*)). On entry, the incremented array *X* must contain the *N* element vector x. On exit, *X* is overwritten with the tranformed vector x.

INCX On entry, *INCX* specifies the increment for the elements of *X*. *INCX* must not be zero. Unchanged on exit.

STPMV, DTPMV, CTPVM or ZTPMV Subroutine

Purpose

Performs matrix–vector operations on a packed triangular matrix.

Library

BLAS Library (**libblas.a**)

FORTRAN Syntax

SUBROUTINE	**STPMV**(*UPLO,TRANS,DIAG,N,AP,X,INCX*)
INTEGER	*INCX,N*
CHARACTER*1	*DIAG,TRANS,UPLO*
REAL	*AP(*),X(*)*
SUBROUTINE	**DTPMV**(*UPLO,TRANS,DIAG,N,AP,X,INCX*)
INTEGER	*INCX,N*
CHARACTER*1	*DIAG,TRANS,UPLO*
DOUBLE PRECISION	*AP(*),X(*)*
SUBROUTINE	**CTPMV**(*UPLO,TRANS,DIAG,N,AP,X,INCX*)
INTEGER	*INCX,N*
CHARACTER*1	*DIAG,TRANS,UPLO*
COMPLEX	*AP(*),X(*)*
SUBROUTINE	**ZTPMV**(*UPLO,TRANS,DIAG,N,AP,X,INCX*)
INTEGER	*INCX,N*
CHARACTER*1	*DIAG,TRANS,UPLO*
COMPLEX*16	*AP(*),X(*)*

Description

The **STPMV**, **DTPMV**, **CTPMV** or **ZTPMV** subroutine performs one of the matrix–vector operations:

$$x := A * x$$

or

$$x := A' * x$$

where x is an *N* element vector and *A* is an *N* by *N* unit, or non–unit, upper or lower triangular matrix, supplied in packed form.

Parameters

UPLO On entry, *UPLO* specifies whether the matrix is an upper or lower triangular matrix as follows:

UPLO = 'U' or 'u'
: *A* is an upper triangular matrix.

UPLO = 'L' or 'l'
: *A* is a lower triangular matrix.

Unchanged on exit.

TRANS

On entry, *TRANS* specifies the operation to be performed as follows:

TRANS = 'N' or 'n'
x := *A* * x

TRANS = 'T' or 't'
x := *A*' * x

TRANS = 'C' or 'c'
x := *A*' * x

Unchanged on exit.

DIAG

On entry, *DIAG* specifies whether or not *A* is unit triangular as follows:

DIAG = 'U' or 'u'
A is assumed to be unit triangular.

DIAG = 'N' or 'n'
A is not assumed to be unit triangular.

Unchanged on exit.

N

On entry, *N* specifies the order of the matrix *A*. *N* must be at least zero. Unchanged on exit.

AP

A vector of dimension at least ((*N* * (*N*+1))/2). On entry with *UPLO* = 'U' or 'u', the array *AP* must contain the upper triangular matrix packed sequentially, column by column, so that *AP*(1) contains *A*(1,1), *AP*(2) and *AP*(3) contain *A*(1,2) and *A*(2,2) respectively, and so on. On entry with *UPLO* = 'L' or 'l', the array *AP* must contain the lower triangular matrix packed sequentially, column by column, so that *AP*(1) contains *A*(1,1), *AP*(2) and *AP*(3) contain *A*(2,1) and *A*(3,1) respectively, and so on. Note that when *DIAG* = 'U' or 'u', the diagonal elements of *A* are not referenced, but are assumed to be unity. Unchanged on exit.

X

A vector of dimension at least (1 + (*N*–1) * abs(*INCX*)). On entry, the incremented array *X* must contain the *N* element vector x. On exit, *X* is overwritten with the tranformed vector x.

INCX

On entry, *INCX* specifies the increment for the elements of *X*. *INCX* must not be zero. Unchanged on exit.

STRSV, DTRSV, CTRSV or ZTRSV Subroutine

Purpose

Solves system of equations.

Library

BLAS Library (**libblas.a**)

FORTRAN Syntax

SUBROUTINE	**STRSV**(*UPLO,TRANS,DIAG,N,A,LDA,X,INCX*)
INTEGER	*INCX,LDA,N*
CHARACTER*1	*DIAG,TRANS,UPLO*
REAL	*A(LDA,*),X(*)*
SUBROUTINE	**DTRSV**(*UPLO,TRANS,DIAG,N,A,LDA,X,INCX*)
INTEGER	*INCX,LDA,N*
CHARACTER*1	*DIAG,TRANS,UPLO*
DOUBLE PRECISION	*A(LDA,*),X(*)*
SUBROUTINE	**CTRSV**(*UPLO,TRANS,DIAG,N,A,LDA,X,INCX*)
INTEGER	*INCX,LDA,N*
CHARACTER*1	*DIAG,TRANS,UPLO*
COMPLEX	*A(LDA,*),X(*)*
SUBROUTINE	**ZTRSV**(*UPLO,TRANS,DIAG,N,A,LDA,X,INCX*)
INTEGER	*INCX,LDA,N*
CHARACTER*1	*DIAG,TRANS,UPLO*
COMPLEX*16	*A(LDA,*),X(*)*

Description

The **STRSV**, **DTRSV**, **CTRSV** or **ZTRSV** subroutine solves one of the systems of equations:

$A * x = b$

or

$A' * x = b$

where b and x are *N* element vectors and *A* is an *N* by *N* unit, or non–unit, upper or lower triangular matrix.

Parameters

UPLO

On entry, *UPLO* specifies whether the matrix is an upper or lower triangular matrix as follows:

UPLO = 'U' or 'u'
A is an upper triangular matrix.

UPLO = 'L' or 'l'
A is a lower triangular matrix.

Unchanged on exit.

TRANS

On entry, *TRANS* specifies the equations to be solved as follows:

TRANS = 'N' or 'n'
$A * x = b$

TRANS = 'T' or 't'
$A' * x = b$

TRANS = 'C' or 'c'
$A' * x = b$

Unchanged on exit.

DIAG
On entry, *DIAG* specifies whether or not *A* is unit triangular as follows:

DIAG = 'U' or 'u'
A is assumed to be unit triangular.

DIAG = 'N' or 'n'
A is not assumed to be unit triangular.

Unchanged on exit.

N
On entry, *N* specifies the order of the matrix *A*. *N* must be at least zero. Unchanged on exit.

A
An array of dimension (*LDA*, *N*). On entry with *UPLO* = 'U' or 'u', the leading *N* by *N* upper triangular part of the array *A* must contain the upper triangular matrix and the strictly lower triangular part of *A* is not referenced. On entry with *UPLO* = 'L' or 'l', the leading *N* by *N* lower triangular part of the array *A* must contain the lower triangular matrix and the strictly upper triangular part of *A* is not referenced. Note that when *DIAG* = 'U' or 'u', the diagonal elements of *A* are not referenced, but are assumed to be unity. Unchanged on exit.

LDA
On entry, *LDA* specifies the first dimension of *A* as declared in the calling (sub) program. *LDA* must be at least max(1, N). Unchanged on exit.

X
A vector of dimension at least (1 + (*N*–1) * abs(*INCX*)). On entry, the incremented array *X* must contain the *N* element right–hand side vector b. On exit, *X* is overwritten with the solution vector x.

INCX
On entry, *INCX* specifies the increment for the elements of *X*. *INCX* must not be zero. Unchanged on exit.

Implementation Specifics

No test for singularity or near–singularity is included in this routine. Such tests must be performed before calling this routine.

STBSV, DTBSV, CTBSV or ZTBSV Subroutine

Purpose

Solves system of equations.

Library

BLAS Library (**libblas.a**)

FORTRAN Syntax

SUBROUTINE	**STBSV**(*UPLO,TRANS,DIAG,N,K,A,LDA,X,INCX*)
INTEGER	*INCX,K,LDA,N*
CHARACTER*1	*DIAG,TRANS,UPLO*
REAL	*A(LDA,*),X(*)*
SUBROUTINE	**DTBSV**(*UPLO,TRANS,DIAG,N,K,A,LDA,X,INCX*)
INTEGER	*INCX,K,LDA,N*
CHARACTER*1	*DIAG,TRANS,UPLO*
DOUBLE PRECISION	*A(LDA,*),X(*)*
SUBROUTINE	**CTBSV**(*UPLO,TRANS,DIAG,N,K,A,LDA,X,INCX*)
INTEGER	*INCX,K,LDA,N*
CHARACTER*1	*DIAG,TRANS,UPLO*
COMPLEX	*A(LDA,*),X(*)*
SUBROUTINE	**ZTBSV**(*UPLO,TRANS,DIAG,N,K,A,LDA,X,INCX*)
INTEGER	*INCX,K,LDA,N*
CHARACTER*1	*DIAG,TRANS,UPLO*
COMPLEX*16	*A(LDA,*),X(*)*

Description

The **STBSV**, **DTBSV**, **CTBSV** or **ZTBSV** subroutine solves one of the systems of equations:

A * x = b

or

A' * x = b

where b and x are *N* element vectors and *A* is an *N* by *N* unit, or non–unit, upper or lower triangular band matrix, with (*K* + 1) diagonals.

Parameters

UPLO
: On entry, *UPLO* specifies whether the matrix is an upper or lower triangular matrix as follows:

  UPLO = 'U' or 'u'
  : *A* is an upper triangular matrix.

  UPLO = 'L' or 'l'
  : *A* is a lower triangular matrix.

  Unchanged on exit.

TRANS
: On entry, *TRANS* specifies the equations to be solved as follows:

  TRANS = 'N' or 'n'
  : *A* * x = b

  TRANS = 'T' or 't'
  : *A'* * x = b

  TRANS = 'C' or 'c'
  : *A'* * x = b

  Unchanged on exit.

DIAG
: On entry, *DIAG* specifies whether *A* is unit triangular as follows:

DIAG = 'U' or 'u'
A is assumed to be unit triangular.

DIAG = 'N' or 'n'
A is not assumed to be unit triangular.

Unchanged on exit.

N
On entry, *N* specifies the order of the matrix *A*. *N* must be at least zero. Unchanged on exit.

K
On entry with *UPLO* = 'U' or 'u', *K* specifies the number of super–diagonals of the matrix *A*. On entry with *UPLO* = 'L' or 'l', *K* specifies the number of sub–diagonals of the matrix *A*. *K* must satisfy 0 .le. *K*. Unchanged on exit.

A
An array of dimension (*LDA*, N). On entry with *UPLO* = 'U' or 'u', the leading (*K* + 1) by *N* part of the array *A* must contain the upper triangular band part of the matrix of coefficients, supplied column by column, with the leading diagonal of the matrix in row (*K* + 1) of the array, the first super–diagonal starting at position 2 in row *K*, and so on. The top left *K* by *K* triangle of the array *A* is not referenced. The following program segment will transfer an upper triangular band matrix from conventional full matrix storage to band storage:

```
DO 20, J = 1, N
       M = K + 1 - J
       DO 10, I = MAX( 1, J - K ), J
             A( M + I, J ) = matrix( I, J )
10 CONTINUE
20 CONTINUE
```

On entry with *UPLO* = 'L' or 'l', the leading (*K* + 1) by N part of the array *A* must contain the lower triangular band part of the matrix of coefficients, supplied column by column, with the leading diagonal of the matrix in row 1 of the array, the first sub–diagonal starting at position 1 in row 2, and so on. The bottom right *K* by *K* triangle of the array *A* is not referenced. The following program segment will transfer a lower triangular band matrix from conventional full matrix storage to band storage:

```
DO 20, J = 1, N
       M = 1 - J
       DO 10, I = J, MIN( N, J + K )
             A( M + I, J ) = matrix( I, J )
10 CONTINUE
20 CONTINUE
```

Note that when *DIAG* = 'U' or 'u' the elements of the array *A* corresponding to the diagonal elements of the matrix are not referenced, but are assumed to be unity. Unchanged on exit.

LDA
On entry, *LDA* specifies the first dimension of *A* as declared in the calling (sub) program. *LDA* must be at least (*K* + 1). Unchanged on exit.

X
A vector of dimension at least (1 + (*N*–1) * abs(*INCX*)). On entry, the incremented array *X* must contain the *N* element right–hand side vector b. On exit, *X* is overwritten with the solution vector x.

INCX On entry, *INCX* specifies the increment for the elements of *X*. *INCX* must not be zero. Unchanged on exit.

Implementation Specifics

No test for singularity or near–singularity is included in this routine. Such tests must be performed before calling this routine.

STPSV, DTPSV, CTPSV or ZTPSV Subroutine

Purpose

Solves systems of equations.

Library

BLAS Library (**libblas.a**)

FORTRAN Syntax

SUBROUTINE	**STPSV**(*UPLO,TRANS,DIAG,N,AP,X,INCX*)
INTEGER	*INCX,N*
CHARACTER*1	*DIAG,TRANS,UPLO*
REAL	*AP(*),X(*)*
SUBROUTINE	**DTPSV**(*UPLO,TRANS,DIAG,N,AP,X,INCX*)
INTEGER	*INCX,N*
CHARACTER*1	*DIAG,TRANS,UPLO*
DOUBLE PRECISION	*AP(*),X(*)*
SUBROUTINE	**CTPSV**(*UPLO,TRANS,DIAG,N,AP,X,INCX*)
INTEGER	*INCX,N*
CHARACTER*1	*DIAG,TRANS,UPLO*
COMPLEX	*AP(*),X(*)*
SUBROUTINE	**ZTPSV**(*UPLO,TRANS,DIAG,N,AP,X,INCX*)
INTEGER	*INCX,N*
CHARACTER*1	*DIAG,TRANS,UPLO*
COMPLEX*16	*AP(*),X(*)*

Description

The **STPSV**, **DTPSV**, **DTPSV** or **ZTPSV** subroutine solves one of the systems of equations:

$A * x = b$

or

$A' * x = b$

where b and x are *N* element vectors and *A* is an *N* by *N* unit, or non–unit, upper or lower triangular matrix, supplied in packed form.

Parameters

UPLO On entry, *UPLO* specifies whether the matrix is an upper or lower triangular matrix as follows:

UPLO = 'U' or 'u'
A is an upper triangular matrix.

UPLO = 'L' or 'l'
A is a lower triangular matrix.

Unchanged on exit.

TRANS On entry, *TRANS* specifies the equations to be solved as follows:

TRANS = 'N' or 'n'
A * x = b

TRANS = 'T' or 't'
A' * x = b

TRANS = 'C' or 'c'
A' * x = b

Unchanged on exit.

DIAG On entry, *DIAG* specifies whether or not *A* is unit triangular as follows:

DIAG = 'U' or 'u'
A is assumed to be unit triangular.

DIAG = 'N' or 'n'
A is not assumed to be unit triangular.

Unchanged on exit.

N On entry, *N* specifies the order of the matrix *A*. *N* must be at least zero. Unchanged on exit.

AP A vector of dimension at least ((*N* * (*N*+1))/2). On entry with *UPLO* = 'U' or 'u', the array *AP* must contain the upper triangular matrix packed sequentially, column by column, so that *AP*(1) contains *A*(1,1), *AP*(2) and *AP*(3) contain *A*(1,2) and *A*(2,2) respectively, and so on. Before entry with *UPLO* = 'L' or 'l', the array *AP* must contain the lower triangular matrix packed sequentially, column by column, so that *AP*(1) contains *A*(1,1), *AP*(2) and *AP*(3) contain *A*(2,1) and *A*(3,1) respectively, and so on. Note that when *DIAG* = 'U' or 'u', the diagonal elements of *A* are not referenced, but are assumed to be unity. Unchanged on exit.

X A vector of dimension at least (1 + (*N*–1) * abs(*INCX*)). On entry, the incremented array *X* must contain the *N* element right–hand side vector b. On exit, *X* is overwritten with the solution vector x.

INCX On entry, *INCX* specifies the increment for the elements of *X*. *INCX* must not be zero. Unchanged on exit.

Implementation Specifics

No test for singularity or near–singularity is included in this routine. Such tests must be performed before calling this routine.

SGER or DGER Subroutine

Purpose

Performs the rank 1 operation.

Library

BLAS Library (**libblas.a**)

FORTRAN Syntax

SUBROUTINE	**SGER**(*M,N,ALPHA,X,INCX,Y,INCY,A,LDA*)
REAL	*ALPHA*
INTEGER	*INCX,INCY,LDA,M,N*
REAL	*A(LDA,*),X(*),Y(*)*

SUBROUTINE	**DGER**(*M,N,ALPHA,X,INCX,Y,INCY,A,LDA*)
DOUBLE PRECISION	*ALPHA*
INTEGER	*INCX,INCY,LDA,M,N*
DOUBLE PRECISION	*A(LDA,*),X(*),Y(*)*

Description

The **SGER** or **DGER** subroutine performs the rank 1 operation:

A := alpha * x * y' + A

where alpha is a scalar, x is an *M* element vector, y is an *N* element vector and *A* is an *M* by *N* matrix.

Parameters

M — On entry, *M* specifies the number of rows of the matrix *A*. *M* must be at least zero. Unchanged on exit.

N — On entry, *N* specifies the number of columns of the matrix *A*. *N* must be at least zero. Unchanged on exit.

ALPHA — On entry, *ALPHA* specifies the scalar alpha. Unchanged on exit.

X — A vector of dimension at least (1 + (*M*–1) * abs(*INCX*)). On entry, the incremented array *X* must contain the *M* element vector x. Unchanged on exit.

INCX — On entry, *INCX* specifies the increment for the elements of *X*. *INCX* must not be zero. Unchanged on exit.

Y — A vector of dimension at least (1 + (*N*–1) * abs(*INCY*)). On entry, the incremented array *Y* must contain the *N* element vector y. Unchanged on exit.

INCY — On entry, *INCY* specifies the increment for the elements of *Y*. *INCY* must not be zero. Unchanged on exit.

A — An array of dimension (*LDA*, *N*). On entry, the leading *M* by *N* part of the array *A* must contain the matrix of coefficients. On exit, *A* is overwritten by the updated matrix.

LDA	On entry, *LDA* specifies the first dimension of *A* as declared in the calling (sub) program. *LDA* must be at least max(1, *M*). Unchanged on exit.

CGERU or ZGERU Subroutine

Purpose

Performs the rank 1 operation.

Library

BLAS Library (**libblas.a**)

FORTRAN Syntax

SUBROUTINE	**CGERU**(*M,N,ALPHA,X,INCX,Y,INCY,A,LDA*)
COMPLEX	*ALPHA*
INTEGER	*INCX,INCY,LDA,M,N*
COMPLEX	*A(LDA,*),X(*),Y(*)*

SUBROUTINE	**ZGERU**
COMPLEX*16	*ALPHA*
INTEGER	*INCX,INCY,LDA,M,N*
COMPLEX*16	*A(LDA,*),X(*),Y(*)*

Description

The **CGERU** or **ZGERU** subroutine performs the rank 1 operation:

A := alpha * x * y' + *A*

where alpha is a scalar, x is an *M* element vector, y is an *N* element vector and *A* is an *M* by *N* matrix.

Parameters

M	On entry, *M* specifies the number of rows of the matrix *A*. *M* must be at least zero. Unchanged on exit.
N	On entry, *N* specifies the number of columns of the matrix *A*. *N* must be at least zero. Unchanged on exit.
ALPHA	On entry, *ALPHA* specifies the scalar alpha. Unchanged on exit.
X	A vector of dimension at least (1 + (*M*–1) * abs(*INCX*)). On entry, the incremented array *X* must contain the *M* element vector x. Unchanged on exit.
INCX	On entry, *INCX* specifies the increment for the elements of *X*. *INCX* must not be zero. Unchanged on exit.
Y	A vector of dimension at least (1 + (*N*–1) * abs(*INCY*)). On entry, the incremented array *Y* must contain the *N* element vector y. Unchanged on exit.
INCY	On entry, *INCY* specifies the increment for the elements of *Y*. *INCY* must not be zero. Unchanged on exit.

A An array of dimension (*LDA*, *N*). On entry, the leading *M* by *N* part of the array *A* must contain the matrix of coefficients. On exit, *A* is overwritten by the updated matrix.

LDA On entry, *LDA* specifies the first dimension of *A* as declared in the calling (sub) program. *LDA* must be at least max(1, *M*). Unchanged on exit.

CGERC or ZGERC Subroutine

Purpose

Performs the rank 1 operation.

Library

BLAS Library (**libblas.a**)

FORTRAN Syntax

SUBROUTINE	**CGERC(***M,N,ALPHA,X,INCX,Y,INCY,A,LDA***)**
COMPLEX	*ALPHA*
INTEGER	*INCX,INCY,LDA,M,N*
COMPLEX	*A(LDA,*),X(*),Y(*)*

SUBROUTINE	**ZGERC**
COMPLEX*16	*ALPHA*
INTEGER	*INCX,INCY,LDA,M,N*
COMPLEX*16	*A(LDA,*),X(*),Y(*)*

Description

The **CGERC** or **ZGERC** subroutine performs the rank 1 operation:

A := alpha * x * conjg(y') + *A*

where alpha is a scalar, x is an *M* element vector, y is an *N* element vector and *A* is an *M* by *N* matrix.

Parameters

M On entry, *M* specifies the number of rows of the matrix *A*. *M* must be at least zero. Unchanged on exit.

N On entry, *N* specifies the number of columns of the matrix *A*. *N* must be at least zero. Unchanged on exit.

ALPHA On entry, *ALPHA* specifies the scalar alpha. Unchanged on exit.

X A vector of dimension at least (1 + (*M*–1) * abs(*INCX*)). On entry, the incremented array *X* must contain the *M* element vector x. Unchanged on exit.

INCX On entry, *INCX* specifies the increment for the elements of *X*. *INCX* must not be zero. Unchanged on exit.

Y A vector of dimension at least (1 + (*N*–1) * abs(*INCY*)). On entry, the incremented array *Y* must contain the *N* element vector y. Unchanged on exit.

INCY	On entry, *INCY* specifies the increment for the elements of *Y*. *INCY* must not be zero. Unchanged on exit.
A	An array of dimension (*LDA*, *N*). On entry, the leading *M* by *N* part of the array *A* must contain the matrix of coefficients. On exit, *A* is overwritten by the updated matrix.
LDA	On entry, *LDA* specifies the first dimension of *A* as declared in the calling (sub) program. *LDA* must be at least max(1, *M*). Unchanged on exit.

CHER or ZHER Subroutine

Purpose

Performs the hermitian rank 1 operation.

Library

BLAS Library (**libblas.a**)

FORTRAN Syntax

SUBROUTINE	**CHER(***UPLO,N,ALPHA,X,INCX,A,LDA***)**
REAL	*ALPHA*
INTEGER	*INCX,LDA,N*
CHARACTER*1	*UPLO*
COMPLEX	*A(LDA,*),X(*)*

SUBROUTINE	**ZHER(***UPLO,N,ALPHA,X,INCX,A,LDA***)**
DOUBLE PRECISION	*ALPHA*
INTEGER	*INCX,LDA,N*
CHARACTER*1	*UPLO*
COMPLEX*16	*A(LDA,*),X(*)*

Description

The **CHER** or **ZHER** subroutine performs the hermitian rank 1 operation:

A := alpha * x * conjg(x') + *A*

where alpha is a real scalar, x is an *N* element vector and *A* is an *N* by *N* hermitian matrix.

Parameters

UPLO
On entry, *UPLO* specifies whether the upper or lower triangular part of the array *A* is to be referenced as follows:

UPLO = 'U' or 'u'
Only the upper triangular part of *A* is to be referenced.

UPLO = 'L' or 'l'
Only the lower triangular part of *A* is to be referenced.

Unchanged on exit.

N
On entry, *N* specifies the order of the matrix *A*. *N* must be at least zero. Unchanged on exit.

ALPHA
On entry, *ALPHA* specifies the scalar alpha. Unchanged on exit.

X	A vector of dimension at least (1 + (*N*–1) * abs(*INCX*)). On entry, the incremented array *X* must contain the *N* element vector x. Unchanged on exit.
INCX	On entry, *INCX* specifies the increment for the elements of *X*. *INCX* must not be zero. Unchanged on exit.
A	An array of dimension (*LDA*, *N*). On entry with *UPLO* = 'U' or 'u', the leading *N* by *N* upper triangular part of the array *A* must contain the upper triangular part of the hermitian matrix and the strictly lower triangular part of *A* is not referenced. On exit, the upper triangular part of the array *A* is overwritten by the upper triangular part of the updated matrix. On entry with *UPLO* = 'L' or 'l', the leading *N* by *N* lower triangular part of the array *A* must contain the lower triangular part of the hermitian matrix and the strictly upper triangular part of *A* is not referenced. On exit, the lower triangular part of the array *A* is overwritten by the lower triangular part of the updated matrix. Note that the imaginary parts of the diagonal elements need not be set, they are assumed to be zero, and on exit they are set to zero.
LDA	On entry, *LDA* specifies the first dimension of *A* as declared in the calling (sub) program. *LDA* must be at least max(1, *N*). Unchanged on exit.

CHPR or ZHPR Subroutine

Purpose

Performs the hermitian rank 1 operation.

Library

BLAS Library (**libblas.a**)

FORTRAN Syntax

SUBROUTINE	**CHPR**(*UPLO,N,ALPHA,X,INCX,AP*)
REAL	*ALPHA*
INTEGER	*INCX,N*
CHARACTER*1	*UPLO*
COMPLEX	*AP(*),X(*)*

SUBROUTINE	**ZHPR**(*UPLO,N,ALPHA,X,INCX,AP*)
DOUBLE PRECISION	*ALPHA*
INTEGER	*INCX,N*
CHARACTER*1	*UPLO*
COMPLEX*16	*AP(*),X(*)*

Description

The **CHPR** or **ZHPR** subroutine performs the hermitian rank 1 operation:

A := alpha * x * conjg(x') + *A*

where alpha is a real scalar, x is an *N* element vector and *A* is an *N* by *N* hermitian matrix, supplied in packed form.

Parameters

UPLO
On entry, *UPLO* specifies whether the upper or lower triangular part of the matrix *A* is supplied in the packed array *AP* as follows:

UPLO = 'U' or 'u'
The upper triangular part of *A* is supplied in *AP*.

UPLO = 'L' or 'l'
The lower triangular part of *A* is supplied in *AP*.

Unchanged on exit.

N
On entry, *N* specifies the order of the matrix *A*. *N* must be at least zero. Unchanged on exit.

ALPHA
On entry, *ALPHA* specifies the scalar alpha. Unchanged on exit.

X
A vector of dimension at least (1 + (*N*–1) * abs(*INCX*)). On entry, the incremented array *X* must contain the *N* element vector x. Unchanged on exit.

INCX
On entry, *INCX* specifies the increment for the elements of *X*. *INCX* must not be zero. Unchanged on exit.

AP
A vector of dimension at least ((*N* * (*N*+1))/2). On entry with *UPLO* = 'U' or 'u', the array *AP* must contain the upper triangular part of the hermitian matrix packed sequentially, column by column, so that *AP*(1) contains *A*(1,1), *AP*(2) and *AP*(3) contain *A*(1,2) and A(2,2) respectively, and so on. On exit, the array *AP* is overwritten by the upper triangular part of the updated matrix. On entry with *UPLO* = 'L' or 'l', the array *AP* must contain the lower triangular part of the hermitian matrix packed sequentially, column by column, so that *AP*(1) contains *A*(1,1), *AP*(2) and *AP*(3) contain *A*(2,1) and *A*(3,1) respectively, and so on. On exit, the array *AP* is overwritten by the lower triangular part of the updated matrix. Note that the imaginary parts of the diagonal elements need not be set, they are assumed to be zero, and on exit they are set to zero.

CHER2 or ZHER2 Subroutine

Purpose

Performs the hermitian rank 2 operation.

Library

BLAS Library (**libblas.a**)

FORTRAN Syntax

SUBROUTINE	**CHER2(***UPLO,N,ALPHA,X,INCX,Y,INCY,A,LDA***)**
COMPLEX	*ALPHA*
INTEGER	*INCX,INCY,LDA,N*
CHARACTER*1	*UPLO*
COMPLEX	*A*(*LDA*,*),*X*(*),*Y*(*)

SUBROUTINE	**ZHER2**(*UPLO,N,ALPHA,X,INCX,Y,INCY,A,LDA*)
COMPLEX*16	*ALPHA*
INTEGER	*INCX,INCY,LDA,N*
CHARACTER*1	*UPLO*
COMPLEX*16	*A(LDA,*),X(*),Y(*)*

Description

The **CHER2** or **ZHER2** subroutine performs the hermitian rank 2 operation:

A := alpha * x * conjg(y') + conjg(alpha) * y * conjy(x') + *A*

where alpha is a scalar, x and y are *N* element vectors and *A* is an *N* by *N* hermitian matrix.

Parameters

UPLO
: On entry, *UPLO* specifies whether the upper or lower triangular part of the array *A* is to be referenced as follows:

  UPLO = 'U' or 'u'
  Only the upper triangular part of *A* is to be referenced.

  UPLO = 'L' or 'l'
  Only the lower triangular part of *A* is to be referenced.

  Unchanged on exit.

N
: On entry, *N* specifies the order of the matrix *A*. *N* must be at least zero. Unchanged on exit.

ALPHA
: On entry, *ALPHA* specifies the scalar alpha. Unchanged on exit.

X
: A vector of dimension at least (1 + (*N*–1) * abs(*INCX*)). On entry, the incremented vector *X* must contain the *N* element vector x. Unchanged on exit.

INCX
: On entry, *INCX* specifies the increment for the elements of *X*. *INCX* must not be zero. Unchanged on exit.

Y
: A vector of dimension at least (1 + (*N*–1) * abs(*INCY*)). On entry, the incremented vector *Y* must contain the *N* element vector y. Unchanged on exit.

INCY
: On entry, *INCY* specifies the increment for the elements of *Y*. *INCY* must not be zero. Unchanged on exit.

A
: An array of dimension (*LDA*, *N*). On entry with *UPLO* = 'U' or 'u', the leading *N* by *N* upper triangular part of the array *A* must contain the upper triangular part of the hermitian matrix and the strictly lower triangular part of *A* is not referenced. On exit, the upper triangular part of the array *A* is overwritten by the upper triangular part of the updated matrix. On entry with *UPLO* = 'L' or 'l', the leading *N* by *N* lower triangular part of the array *A* must contain the lower triangular part of the hermitian matrix and the strictly upper triangular part of *A* is not referenced. On exit, the lower triangular part of the array *A* is overwritten by the lower triangular part of the updated matrix. Note that the imaginary parts of the diagonal elements need not be set; they are assumed to be zero, and on exit they are set to zero.

LDA On entry, *LDA* specifies the first dimension of *A* as declared in the calling (sub) program. *LDA* must be at least max(1, *N*). Unchanged on exit.

CHPR2 or ZHPR2 Subroutine

Purpose

Performs the hermitian rank 2 operation.

Library

BLAS Library (**libblas.a**)

FORTRAN Syntax

SUBROUTINE	**CHPR2** (*UPLO,N,ALPHA,X,INCX,Y,INCY,AP*)
COMPLEX	*ALPHA*
INTEGER	*INCX,INCY,N*
CHARACTER*1	*UPLO*
COMPLEX	*AP(*),X(*),Y(*)*

SUBROUTINE	**ZHPR2**
COMPLEX*16	*ALPHA*
INTEGER	*INCX,INCY,N*
CHARACTER*1	*UPLO*
COMPLEX*16	*AP(*),X(*),Y(*)*

Description

The **CHPR2** or **ZHPR2** subroutine performs the hermitian rank 2 operation:

A := alpha * x * conjg(y') + conjg(alpha) * y * conjg(x') + *A*

where alpha is a scalar, x and y are *N* element vectors and *A* is an *N* by *N* hermitian matrix, supplied in packed form.

Parameters

UPLO On entry, *UPLO* specifies whether the upper or lower triangular part of the matrix *A* is supplied in the packed array *AP* as follows:

UPLO = 'U' or 'u'
The upper triangular part of *A* is supplied in *AP*.

UPLO = 'L' or 'l'
The lower triangular part of *A* is supplied in *AP*.

Unchanged on exit.

N On entry, *N* specifies the order of the matrix *A*. *N* must be at least zero. Unchanged on exit.

ALPHA On entry, *ALPHA* specifies the scalar alpha. Unchanged on exit.

X A vector of dimension at least (1 + (*N*–1) * abs(*INCX*)). On entry, the incremented array *X* must contain the *N* element vector x. Unchanged on exit.

INCX — On entry, *INCX* specifies the increment for the elements of *X*. *INCX* must not be zero. Unchanged on exit.

Y — A vector of dimension at least (1 + (*N*–1) * abs(*INCY*)). On entry, the incremented array *Y* must contain the *N* element vector y. Unchanged on exit.

INCY — On entry, *INCY* specifies the increment for the elements of *Y*. *INCY* must not be zero. Unchanged on exit.

AP — A vector of dimension at least ((*N* * (*N*+1))/2). On entry with *UPLO* = 'U' or 'u', the array *AP* must contain the upper triangular part of the hermitian matrix packed sequentially, column by column, so that *AP*(1) contains *A*(1,1), *AP*(2) and *AP*(3) contain *A*(1,2) and *A*(2,2) respectively, and so on. On exit, the array *AP* is overwritten by the upper triangular part of the updated matrix. On entry with *UPLO* = 'L' or 'l', the array *AP* must contain the lower triangular part of the hermitian matrix packed sequentially, column by column, so that *AP*(1) contains *A*(1,1), *AP*(2) and *AP*(3) contain *A*(2,1) and *A*(3,1) respectively, and so on. On exit, the array *AP* is overwritten by the lower triangular part of the updated matrix. Note that the imaginary parts of the diagonal elements need not be set, they are assumed to be zero, and on exit they are set to zero.

SSYR or DSYR Subroutine

Purpose

Performs the symmetric rank 1 opertation.

Library

BLAS Library (**libblas.a**)

FORTRAN Syntax

SUBROUTINE	**SSYR**(*UPLO,N,ALPHA,X,INCX,A,LDA*)
REAL	*ALPHA*
INTEGER	*INCX,LDA,N*
CHARACTER*1	*UPLO*
REAL	*A(LDA,*),X(*)*

SUBROUTINE	**DSYR**(*UPLO,N,ALPHA,X,INCX,A,LDA*)
DOUBLE PRECISION	*ALPHA*
INTEGER	*INCX,LDA,N*
CHARACTER*1	*UPLO*
DOUBLE PRECISION	*A(LDA,*),X(*)*

Description

The **SSYR** or **DSYR** subroutine performs the symmetric rank 1 operation:

A := alpha * x * x' + *A*

where alpha is a real scalar, x is an *N* element vector and *A* is an *N* by *N* symmetric matrix.

Parameters

UPLO
: On entry, *UPLO* specifies whether the upper or lower triangular part of the array *A* is to be referenced as follows:

    UPLO = 'U' or 'u'
    Only the upper triangular part of *A* is to be referenced.

    UPLO = 'L' or 'l'
    Only the lower triangular part of *A* is to be referenced.

    Unchanged on exit.

N
: On entry, *N* specifies the order of the matrix *A*. *N* must be at least zero. Unchanged on exit.

ALPHA
: On entry, *ALPHA* specifies the scalar alpha. Unchanged on exit.

X
: A vector of dimension at least (1 + (*N*–1) * abs(*INCX*)). On entry, the incremented array *X* must contain the *N* element vector x. Unchanged on exit.

INCX
: On entry, *INCX* specifies the increment for the elements of *X*. *INCX* must not be zero. Unchanged on exit.

A
: An array of dimension (*LDA*, *N*). On entry with *UPLO* = 'U' or 'u', the leading *N* by *N* upper triangular part of the array *A* must contain the upper triangular part of the symmetric matrix and the strictly lower triangular part of *A* is not referenced. On exit, the upper triangular part of the array *A* is overwritten by the upper triangular part of the updated matrix. On entry with *UPLO* = 'L' or 'l', the leading *N* by *N* lower triangular part of the array *A* must contain the lower triangular part of the symmetric matrix and the strictly upper triangular part of *A* is not referenced. On exit, the lower triangular part of the array *A* is overwritten by the lower triangular part of the updated matrix.

LDA
: On entry, *LDA* specifies the first dimension of *A* as declared in the calling (sub) program. *LDA* must be at least max(1, *N*). Unchanged on exit.

SSPR or DSPR Subroutine

Purpose

Performs the symmetric rank 1 operation.

Library

BLAS Library (**libblas.a**)

FORTRAN Syntax

SUBROUTINE	**SSPR(***UPLO,N,ALPHA,X,INCX,AP***)**
REAL	*ALPHA*
INTEGER	*INCX,N*
CHARACTER*1	*UPLO*
REAL	*AP(*),X(*)*

SUBROUTINE	**DSPR(***UPLO,N,ALPHA,X,INCX,AP***)**
DOUBLE PRECISION	*ALPHA*
INTEGER	*INCX,N*
CHARACTER*1	*UPLO*
DOUBLE PRECISION	*AP*(*),*X*(*)

Description

The **SSPR** or **DSPR** subroutine performs the symmetric rank 1 operation:

A := alpha * x * x' + *A*

where alpha is a real scalar, x is an *N* element vector and *A* is an *N* by *N* symmetric matrix, supplied in packed form.

Parameters

UPLO
: On entry, *UPLO* specifies whether the upper or lower triangular part of the matrix *A* is supplied in the packed array *AP* as follows:

  UPLO = 'U' or 'u'
  The upper triangular part of *A* is supplied in *AP*.

  UPLO = 'L' or 'l'
  The lower triangular part of *A* is supplied in *AP*.

  Unchanged on exit.

N
: On entry, *N* specifies the order of the matrix *A*. *N* must be at least zero. Unchanged on exit.

ALPHA
: On entry, *ALPHA* specifies the scalar alpha. Unchanged on exit.

X
: A vector of dimension at least (1 + (*N*–1) * abs(*INCX*)). On entry, the incremented array *X* must contain the *N* element vector x. Unchanged on exit.

INCX
: On entry, *INCX* specifies the increment for the elements of *X*. *INCX* must not be zero. Unchanged on exit.

AP
: A vector of dimension at least ((*N* * (*N*+1))/2). On entry with *UPLO* = 'U' or 'u', the array *AP* must contain the upper triangular part of the symmetric matrix packed sequentially, column by column, so that *AP*(1) contains *A*(1,1), *AP*(2) and *AP*(3) contain *A*(1,2) and *A*(2,2) respectively, and so on. On exit, the array *AP* is overwritten by the upper triangular part of the updated matrix. On entry with *UPLO* = 'L' or 'l', the array *AP* must contain the lower triangular part of the symmetric matrix packed sequentially, column by column, so that *AP*(1) contains *A*(1,1), *AP*(2) and *AP*(3) contain *A*(2,1) and *A*(3,1) respectively, and so on. On exit, the array *AP* is overwritten by the lower triangular part of the updated matrix.

SSYR2 or DSYR2 Subroutine

Purpose

Performs the symmetric rank 2 operation.

Level 2: matrix–vector operations

Library

BLAS Library (**libblas.a**)

FORTRAN Syntax

SUBROUTINE	**SSYR2**(*UPLO,N,ALPHA,X,INCX,Y,INCY,A,LDA*)
REAL	*ALPHA*
INTEGER	*INCX,INCY,LDA,N*
CHARACTER*1	*UPLO*
REAL	*A(LDA,*),X(*),Y(*)*

SUBROUTINE	**DSYR2**(*UPLO,N,ALPHA,X,INCX,Y,INCY,A,LDA*)
DOUBLE PRECISION	*ALPHA*
INTEGER	*INCX,INCY,LDA,N*
CHARACTER*1	*UPLO*
DOUBLE PRECISION	*A(LDA,*),X(*),Y(*)*

Description

The **SSYR2** or **DSYR2** subroutine performs the symmetric rank 2 operation:

A := alpha * x * y' + alpha * y * x' + *A*

where alpha is a scalar, x and y are *N* element vectors and *A* is an *N* by *N* symmetric matrix.

Parameters

UPLO
: On entry, *UPLO* specifies whether the upper or lower triangular part of the array *A* is to be referenced as follows:

  UPLO = 'U' or 'u'
  Only the upper triangular part of *A* is to be referenced.

  UPLO = 'L' or 'l'
  Only the lower triangular part of *A* is to be referenced.

  Unchanged on exit.

N
: On entry, *N* specifies the order of the matrix *A*. *N* must be at least zero. Unchanged on exit.

ALPHA
: On entry, *ALPHA* specifies the scalar alpha. Unchanged on exit.

X
: A vector of dimension at least (1 + (*N*–1) * abs(*INCX*)). On entry, the incremented array *X* must contain the *N* element vector x. Unchanged on exit.

INCX
: On entry, *INCX* specifies the increment for the elements of *X*. *INCX* must not be zero. Unchanged on exit.

Y
: A vector of dimension at least (1 + (*N*–1) * abs(*INCY*)). On entry, the incremented array *Y* must contain the *N* element vector y. Unchanged on exit.

INCY
: On entry, *INCY* specifies the increment for the elements of *Y*. *INCY* must not be zero. Unchanged on exit.

A
: An array of dimension (*LDA*, *N*). On entry with *UPLO* = 'U' or 'u', the leading *N* by *N* upper triangular part of the array *A* must contain the upper

triangular part of the symmetric matrix and the strictly lower triangular part of *A* is not referenced. On exit, the upper triangular part of the array *A* is overwritten by the upper triangular part of the updated matrix. On entry with *UPLO* = 'L' or 'l', the leading *N* by *N* lower triangular part of the array *A* must contain the lower triangular part of the symmetric matrix and the strictly upper triangular part of *A* is not referenced. On exit, the lower triangular part of the array *A* is overwritten by the lower triangular part of the updated matrix.

LDA On entry, *LDA* specifies the first dimension of *A* as declared in the calling (sub) program. *LDA* must be at least max(1, *N*). Unchanged on exit.

SSPR2 or DSPR2 Subroutine

Purpose

Performs the symmetric rank 2 operation.

Library

BLAS Library (**libblas.a**)

FORTRAN Syntax

SUBROUTINE	**SSPR2(***UPLO,N,ALPHA,X,INCX,Y,INCY,AP***)**
REAL	*ALPHA*
INTEGER	*INCX,INCY,N*
CHARACTER*1	*UPLO*
REAL	*AP(*),X(*),Y(*)*

SUBROUTINE	**DSPR2(***UPLO,N,ALPHA,X,INCX,Y,INCY,AP***)**
DOUBLE PRECISION	*ALPHA*
INTEGER	*INCX,INCY,N*
CHARACTER*1	*UPLO*
DOUBLE PRECISION	*AP(*),X(*),Y(*)*

Description

The **SSPR2** or **DSPR2** subroutine performs the symmetric rank 2 operation:

A := alpha * x * y' + alpha * y * x' + A

where alpha is a scalar, x and y are *N* element vectors and *A* is an *N* by *N* symmetric matrix, supplied in packed form.

Parameters

UPLO On entry, *UPLO* specifies whether the upper or lower triangular part of the matrix *A* is supplied in the packed array *AP* as follows:

UPLO = 'U' or 'u'
The upper triangular part of *A* is supplied in *AP*.

UPLO = 'L' or 'l'
The lower triangular part of *A* is supplied in *AP*.

Unchanged on exit.

N	On entry, *N* specifies the order of the matrix *A*. *N* must be at least zero. Unchanged on exit.
ALPHA	On entry, *ALPHA* specifies the scalar alpha. Unchanged on exit.
X	A vector of dimension at least (1 + (*N*–1) * abs(*INCX*)). On entry, the incremented array *X* must contain the *N* element vector x. Unchanged on exit.
INCX	On entry, *INCX* specifies the increment for the elements of *X*. *INCX* must not be zero. Unchanged on exit.
Y	A vector of dimension at least (1 + (*N*–1) * abs(*INCY*)). On entry, the incremented array *Y* must contain the *N* element vector y. Unchanged on exit.
INCY	On entry, *INCY* specifies the increment for the elements of *Y*. *INCY* must not be zero. Unchanged on exit.
AP	A vector of dimension at least ((*N* * (*N*+1))/2). On entry with *UPLO* = 'U' or 'u', the array *AP* must contain the upper triangular part of the symmetric matrix packed sequentially, column by column, so that *AP*(1) contains *A*(1,1), *AP*(2) and *AP*(3) contain *A*(1,2) and *A*(2,2) respectively, and so on. On exit, the array *AP* is overwritten by the upper triangular part of the updated matrix. On entry with *UPLO* = 'L' or 'l', the array *AP* must contain the lower triangular part of the symmetric matrix packed sequentially, column by column, so that *AP*(1) contains *A*(1,1), *AP*(2) and *AP*(3) contain *A*(2,1) and *A*(3,1) respectively, and so on. On exit, the array *AP* is overwritten by the lower triangular part of the updated matrix.

SGEMM, DGEMM, CGEMM or ZGEMM Subroutine

Purpose

Performs matrix–matrix operations on general matrices.

Library

BLAS Library (**libblas.a**)

FORTRAN Syntax

SUBROUTINE	**SGEMM(***TRANSA,TRANSB,M,N,K,ALPHA,A,LDA,B,LDB, BETA,C,LDC***)**
CHARACTER*1	*TRANSA,TRANSB*
INTEGER	*M,N,K,LDA,LDB,LDC*
REAL	*ALPHA,BETA*
REAL	*A(LDA,*),B(LDB,*),C(LDC,*)*

SUBROUTINE	**DGEMM(***TRANSA,TRANSB,M,N,K,ALPHA,A,LDA,B,LDB, BETA,C,LDC***)**
CHARACTER*1	*TRANSA,TRANSB*
INTEGER	*M,N,K,LDA,LDB,LDC*
DOUBLE PRECISION	*ALPHA,BETA*
DOUBLE PRECISION	*A(LDA,*),B(LDB,*),C(LDC,*)*

SUBROUTINE	**CGEMM(***TRANSA,TRANSB,M,N,K,ALPHA,A,LDA,B,LDB, BETA,C,LDC***)**
CHARACTER*1	*TRANSA,TRANSB*
INTEGER	*M,N,K,LDA,LDB,LDC*
COMPLEX	*ALPHA,BETA*
COMPLEX	*A(LDA,*),B(LDB,*),C(LDC,*)*

SUBROUTINE	**ZGEMM(***TRANSA,TRANSB,M,N,K,ALPHA,A,LDA,B,LDB, BETA,C,LDC***)**
CHARACTER*1	*TRANSA,TRANSB*
INTEGER	*M,N,K,LDA,LDB,LDC*
COMPLEX*16	*ALPHA,BETA*
COMPLEX*16	*A(LDA,*),B(LDB,*),C(LDC,*)*

Description

The **SGEMM**, **DGEMM**, **CGEMM** or **ZGEMM** subroutine performs one of the matrix–matrix operations:

C := alpha * op(*A*) * op(*B*) + beta * *C*

where op(X) is one of op(X) = X or op(X) = X',alpha and beta are scalars, and *A*, *B* and *C* are matrices, with op(*A*) an *M* by *K* matrix, op(*B*) a *K* by *N* matrix and *C* an *M* by *N* matrix.

Parameters

TRANSA — On entry, *TRANSA* specifies the form of op(*A*) to be used in the matrix multiplication as follows:

TRANSA = 'N' or 'n'
op(*A*) = *A*

TRANSA = 'T' or 't'
op(*A*) = *A'*

TRANSA = 'C' or 'c'
op(*A*) = *A'*

Unchanged on exit.

TRANSB — On entry, *TRANSB* specifies the form of op(*B*) to be used in the matrix multiplication as follows:

TRANSB = 'N' or 'n'
op(*B*) = *B*

TRANSB = 'T' or 't'
op(B) = B'

TRANSB = 'C' or 'c'
op(*B*) = *B'*

Unchanged on exit.

M — On entry, *M* specifies the number of rows of the matrix op(*A*) and of the matrix *C*. *M* must be at least zero. Unchanged on exit.

N — On entry, *N* specifies the number of columns of the matrix op(*B*) and the number of columns of the matrix *C*. *N* must be at least zero. Unchanged on exit.

K — On entry, *K* specifies the number of columns of the matrix op(*A*) and the number of rows of the matrix op(*B*). *K* must be at least zero. Unchanged on exit.

ALPHA — On entry, *ALPHA* specifies the scalar alpha. Unchanged on exit.

A — An array of dimension (*LDA*, *KA*), where *KA* is *K* when *TRANSA* = 'N' or 'n', and is *M* otherwise. On entry with *TRANSA* = 'N' or 'n', the leading *M* by *K* part of the array *A* must contain the matrix *A*, otherwise the leading *K* by *M* part of the array *A* must contain the matrix *A*. Unchanged on exit.

LDA — On entry, *LDA* specifies the first dimension of *A* as declared in the calling (sub) program. When *TRANSA* = 'N' or 'n' then *LDA* must be at least max(1, *M*), otherwise *LDA* must be at least max(1, *K*). Unchanged on exit.

B — An array of dimension (*LDB*, *KB*) where *KB* is *N* when *TRANSB* = 'N' or 'n', and is *K* otherwise. On entry with *TRANSB* = 'N' or 'n', the leading *K* by *N* part of the array *B* must contain the matrix *B*, otherwise the leading *N* by *K* part of the array *B* must contain the matrix *B*. Unchanged on exit.

LDB — On entry, *LDB* specifies the first dimension of *B* as declared in the calling (sub) program. When *TRANSB* = 'N' or 'n' then *LDB* must be at least max(1, *K*), otherwise *LDB* must be at least max(1, *N*). Unchanged on exit.

BETA — On entry, *BETA* specifies the scalar beta. When *BETA* is supplied as zero then *C* need not be set on input. Unchanged on exit.

C — An array of dimension (*LDC*, *N*). On entry, the leading *M* by *N* part of the array *C* must contain the matrix *C*, except when beta is zero, in which case

C need not be set on entry. On exit, the array *C* is overwritten by the *M* by *N* matrix (alpha * op(*A*) * op(*B*) + beta * *C*).

LDC — On entry, *LDC* specifies the first dimension of *C* as declared in the calling (sub) program. *LDC* must be at least max(1, *M*). Unchanged on exit.

SSYMM, DSYMM, CSYMM or ZSYMM Subroutine

Purpose

Performs matrix–matrix matrix operations on symmertric matrices.

Library

BLAS Library (**libblas.a**)

FORTRAN Syntax

```
SUBROUTINE        SSYMM(SIDE,UPLO,M,N,ALPHA,A,LDA,B,LDB,BETA,C,
                  LDC)
CHARACTER*1       SIDE,UPLO
INTEGER           M,N,LDA,LDB,LDC
REAL              ALPHA,BETA
REAL              A(LDA,*),B(LDB,*),C(LDC,*)

SUBROUTINE        DSYMM(SIDE,UPLO,M,N,ALPHA,A,LDA,B,LDB,BETA,C,
                  LDC)
CHARACTER*1       SIDE,UPLO
INTEGER           M,N,LDA,LDB,LDC
DOUBLE PRECISION  ALPHA,BETA
DOUBLE PRECISION  A(LDA,*),B(LDB,*),C(LDC,*)

SUBROUTINE        CSYMM(SIDE,UPLO,M,N,ALPHA,A,LDA,B,LDB,BETA,C,
                  LDC)
CHARACTER*1       SIDE,UPLO
INTEGER           M,N,LDA,LDB,LDC
COMPLEX           ALPHA,BETA
COMPLEX           A(LDA,*),B(LDB,*),C(LDC,*)

SUBROUTINE        ZSYMM(SIDE,UPLO,M,N,ALPHA,A,LDA,B,LDB,BETA,C,
                  LDC)
CHARACTER*1       SIDE,UPLO
INTEGER           M,N,LDA,LDB,LDC
COMPLEX*16        ALPHA,BETA
COMPLEX*16        A(LDA,*),B(LDB,*),C(LDC,*)
```

Description

The **SSYMM, DSYMM, CSYMM** or **ZSYMM** subroutine performs one of the matrix–matrix operations:

C := alpha * *A* * *B* + beta * *C*

or

C := alpha * *B* * *A* + beta * *C*

where alpha and beta are scalars, *A* is a symmetric matrix and *B* and *C* are *M* by *N* matrices.

Parameters

SIDE

On entry, *SIDE* specifies whether the symmetric matrix *A* appears on the left or right in the operation as follows:

SIDE = 'L' or 'l'
$C := \text{alpha} * A * B + \text{beta} * C$

SIDE = 'R' or 'r'
$C := \text{alpha} * B * A + \text{beta} * C$

Unchanged on exit.

UPLO

On entry, *UPLO* specifies whether the upper or lower triangular part of the symmetric matrix *A* is to be referenced as follows:

UPLO = 'U' or 'u'
Only the upper triangular part of the symmetric matrix is to be referenced.

UPLO = 'L' or 'l'
Only the lower triangular part of the symmetric matrix is to be referenced.

Unchanged on exit.

M

On entry, *M* specifies the number of rows of the matrix *C*. *M* must be at least zero. Unchanged on exit.

N

On entry, *N* specifies the number of columns of the matrix *C*. *N* must be at least zero. Unchanged on exit.

ALPHA

On entry, *ALPHA* specifies the scalar alpha. Unchanged on exit.

A

An array of dimension (*LDA*, *KA*), where *KA* is *M* when *SIDE* = 'L' or 'l' and is *N* otherwise. On entry with *SIDE* = 'L' or 'l', the *M* by *M* part of the array *A* must contain the symmetric matrix, such that when *UPLO* = 'U' or 'u', the leading *M* by *M* upper triangular part of the array *A* must contain the upper triangular part of the symmetric matrix and the strictly lower triangular part of *A* is not referenced, and when *UPLO* = 'L' or 'l', the leading *M* by *M* lower triangular part of the array *A* must contain the lower triangular part of the symmetric matrix and the strictly upper triangular part of *A* is not referenced. On entry with *SIDE* = 'R' or 'r', the *N* by *N* part of the array *A* must contain the symmetric matrix, such that when *UPLO* = 'U' or 'u', the leading *N* by *N* upper triangular part of the array *A* must contain the upper triangular part of the symmetric matrix and the strictly lower triangular part of *A* is not referenced, and when *UPLO* = 'L' or 'l', the leading *N* by *N* lower triangular part of the array *A* must contain the lower triangular part of the symmetric matrix and the strictly upper triangular part of *A* is not referenced. Unchanged on exit.

LDA

On entry, *LDA* specifies the first dimension of *A* as declared in the calling (sub) program. When *SIDE* = 'L' or 'l' then *LDA* must be at least max(1, *M*), otherwise *LDA* must be at least max(1, *N*). Unchanged on exit.

B

An array of dimension (*LDB*, *N*). On entry, the leading *M* by *N* part of the array *B* must contain the matrix *B*. Unchanged on exit.

LDB On entry, *LDB* specifies the first dimension of *B* as declared in the calling (sub) program. *LDB* must be at least max(1, *M*). Unchanged on exit.

BETA On entry, *BETA* specifies the scalar beta. When *BETA* is supplied as zero then *C* need not be set on input. Unchanged on exit.

C An array of dimension (*LDC*, *N*). On entry, the leading *M* by *N* part of the array *C* must contain the matrix *C*, except when beta is zero, in which case *C* need not be set on entry. On exit, the array *C* is overwritten by the *M* by *N* updated matrix.

LDC On entry, *LDC* specifies the first dimension of *C* as declared in the calling (sub) program. *LDC* must be at least max(1, *M*). Unchanged on exit.

CHEMM or ZHEMM Subroutine

Purpose

Performs matrix–matrix operations on hermitian matrices.

Library

BLAS Library (**libblas.a**)

FORTRAN Syntax

SUBROUTINE	**CHEMM**(*SIDE,UPLO,M,N,ALPHA,A,LDA,B,LDB,BETA,C,LDC*)
CHARACTER*1	*SIDE,UPLO*
INTEGER	*M,N,LDA,LDB,LDC*
COMPLEX	*ALPHA,BETA*
COMPLEX	*A*(*LDA,**),*B*(*LDB,**),*C*(*LDC,**)

SUBROUTINE	**ZHEMM**(*SIDE,UPLO,M,N,ALPHA,A,LDA,B,LDB,BETA,C,LDC*)
CHARACTER*1	*SIDE,UPLO*
INTEGER	*M,N,LDA,LDB,LDC*
COMPLEX*16	*ALPHA,BETA*
COMPLEX*16	*A*(*LDA,**),*B*(*LDB,**),*C*(*LDC,**)

Purpose

The **CHEMM** or **ZHEMM** subroutine performs one of the matrix–matrix operations:

C := alpha * *A* * *B* + beta * *C*

or

C := alpha * *B* * *A* + beta * *C*

where alpha and beta are scalars, *A* is an hermitian matrix, and *B* and *C* are *M* by *N* matrices.

Parameters

SIDE On entry, *SIDE* specifies whether the hermitian matrix *A* appears on the left or right in the operation as follows:

SIDE = 'L' or 'l'
C := alpha * *A* * *B* + beta * *C*

SIDE = 'R' or 'r'
$C := \text{alpha} * B * A + \text{beta} * C$

Unchanged on exit.

UPLO — On entry, *UPLO* specifies whether the upper or lower triangular part of the hermitian matrix *A* is to be referenced as follows:

UPLO = 'U' or 'u'
Only the upper triangular part of the hermitian matrix is to be referenced.

UPLO = 'L' or 'l'
Only the lower triangular part of the hermitian matrix is to be referenced.

Unchanged on exit.

M — On entry, *M* specifies the number of rows of the matrix *C*. *M* must be at least zero. Unchanged on exit.

N — On entry, *N* specifies the number of columns of the matrix *C*. *N* must be at least zero. Unchanged on exit.

ALPHA — On entry, *ALPHA* specifies the scalar alpha. Unchanged on exit.

A — An array of dimension (*LDA*, *KA*), where *KA* is *M* when *SIDE* = 'L' or 'l' and is *N* otherwise. On entry with *SIDE* = 'L' or 'l', the *M* by *M* part of the array *A* must contain the hermitian matrix, such that when *UPLO* = 'U' or 'u', the leading *M* by *M* upper triangular part of the array *A* must contain the upper triangular part of the hermitian matrix and the strictly lower triangular part of *A* is not referenced, and when *UPLO* = 'L' or 'l', the leading *M* by *M* lower triangular part of the array *A* must contain the lower triangular part of the hermitian matrix and the strictly upper triangular part of *A* is not referenced. On entry with *SIDE* = 'R' or 'r', the *N* by *N* part of the array *A* must contain the hermitian matrix, such that when *UPLO* = 'U' or 'u', the leading *N* by *N* upper triangular part of the array *A* must contain the upper triangular part of the hermitian matrix and the strictly lower triangular part of *A* is not referenced, and when *UPLO* = 'L' or 'l', the leading *N* by *N* lower triangular part of the array *A* must contain the lower triangular part of the hermitian matrix and the strictly upper triangular part of *A* is not referenced. Note that the imaginary parts of the diagonal elements need not be set, they are assumed to be zero. Unchanged on exit.

LDA — On entry, *LDA* specifies the first dimension of *A* as declared in the calling (sub) program. When *SIDE* = 'L' or 'l' then *LDA* must be at least max(1, *M*), otherwise *LDA* must be at least max(1, *N*). Unchanged on exit.

B — An array of dimension (*LDB*, *N*). On entry, the leading *M* by *N* part of the array *B* must contain the matrix *B*. Unchanged on exit.

LDB — On entry, *LDB* specifies the first dimension of *B* as declared in the calling (sub) program. *LDB* must be at least max(1, *M*). Unchanged on exit.

BETA — On entry, *BETA* specifies the scalar beta. When *BETA* is supplied as zero then *C* need not be set on input. Unchanged on exit.

C — An array of dimension (*LDC*, *N*). On entry, the leading *M* by *N* part of the array *C* must contain the matrix *C*, except when beta is zero, in which case *C* need not be set on entry. On exit, the array *C* is overwritten by the *M* by *N* updated matrix.

LDC — On entry, *LDC* specifies the first dimension of *C* as declared in the calling (sub) program. *LDC* must be at least max(1, *M*). Unchanged on exit.

SSYRK, DSYRK, CSYRK or ZSYRK Subroutine

Purpose

Perform symmetric rank k operations.

Library

BLAS Library (**libblas.a**)

FORTRAN Syntax

```
SUBROUTINE          SSYRK(UPLO,TRANS,N,K,ALPHA,A,LDA,BETA,C,LDC)
CHARACTER*1         UPLO,TRANS
INTEGER             N,K,LDA,LDC
REAL                ALPHA,BETA
REAL                A(LDA,*),C(LDC,*)

SUBROUTINE          DSYRK(UPLO,TRANS,N,K,ALPHA,A,LDA,BETA,C,LDC)
CHARACTER*1         UPLO,TRANS
INTEGER             N,K,LDA,LDC
DOUBLE PRECISION    ALPHA,BETA
DOUBLE PRECISION    A(LDA,*),C(LDC,*)

SUBROUTINE          CSYRK(UPLO,TRANS,N,K,ALPHA,A,LDA,BETA,C,LDC)
CHARACTER*1         UPLO,TRANS
INTEGER             N,K,LDA,LDC
COMPLEX             ALPHA,BETA
COMPLEX             A(LDA,*),C(LDC,*)

SUBROUTINE          ZSYRK(UPLO,TRANS,N,K,ALPHA,A,LDA,BETA,C,LDC)
CHARACTER*1         UPLO,TRANS
INTEGER             N,K,LDA,LDC
COMPLEX*16          ALPHA,BETA
COMPLEX*16          A(LDA,*),C(LDC,*)
```

Description

The **SSYRK**, **DSYRK**, **CSYRK** or **ZSYRK** subroutine performs one of the symmetric rank k operations:

C := alpha * *A* * *A*' + beta * *C*

or

C := alpha * *A*' * *A* + beta * *C*

where alpha and beta are scalars, *C* is an *N* by *N* symmetric matrix, and *A* is an *N* by *K* matrix in the first case and a *K* by *N* matrix in the second case.

Parameters

UPLO

On entry, *UPLO* specifies whether the upper or lower triangular part of the array *C* is to be referenced as follows:

UPLO = 'U' or 'u'
Only the upper triangular part of *C* is to be referenced.

UPLO = 'L' or 'l'
Only the lower triangular part of *C* is to be referenced.

Unchanged on exit.

TRANS

On entry, *TRANS* specifies the operation to be performed as follows:

TRANS = 'N' or 'n'
C := alpha * *A* * *A*' + beta * *C*

TRANS = 'T' or 't'
C := alpha * *A*' * *A* + beta * *C*

TRANS = 'C' or 'c'
C := alpha * *A*' * *A* + beta * *C*

Unchanged on exit.

N

On entry, *N* specifies the order of the matrix *C*. *N* must be at least zero. Unchanged on exit.

K

On entry with *TRANS* = 'N' or 'n', *K* specifies the number of columns of the matrix *A*, and on entry with *TRANS* = 'T' or 't' or 'C' or 'c', *K* specifies the number of rows of the matrix *A*. *K* must be at least zero. Unchanged on exit.

ALPHA

On entry, *ALPHA* specifies the scalar alpha. Unchanged on exit.

A

An array of dimension (*LDA*, KA), where *KA* is *K* when *TRANS* = 'N' or 'n', and is *N* otherwise. On entry with *TRANS* = 'N' or 'n', the leading *N* by *K* part of the array *A* must contain the matrix *A*, otherwise the leading *K* by *N* part of the array *A* must contain the matrix *A*. Unchanged on exit.

LDA

On entry, *LDA* specifies the first dimension of *A* as declared in the calling (sub) program. When *TRANS* = 'N' or 'n', *LDA* must be at least max(1, *N*); otherwise *LDA* must be at least max(1, *K*). Unchanged on exit.

BETA

On entry, *BETA* specifies the scalar beta. Unchanged on exit.

C

An array of dimension (*LDC*, *N*). On entry with *UPLO* = 'U' or 'u', the leading *N* by *N* upper triangular part of the array *C* must contain the upper triangular part of the symmetric matrix and the strictly lower triangular part of *C* is not referenced. On exit, the upper triangular part of the array *C* is overwritten by the upper triangular part of the updated matrix. On entry with *UPLO* = 'L' or 'l', the leading N by N lower triangular part of the array *C* must contain the lower triangular part of the symmetric matrix and the strictly upper triangular part of *C* is not referenced. On exit, the lower triangular part of the array *C* is overwritten by the lower triangular part of the updated matrix.

LDC

On entry, *LDC* specifies the first dimension of *C* as declared in the calling (sub) program. *LDC* must be at least max(1, *N*). Unchanged on exit.

CHERK or ZHERK Subroutine

Purpose

Performs hermitian rank k operations.

Library

BLAS Library (**libblas.a**)

FORTRAN Syntax

SUBROUTINE	**CHERK(***UPLO,TRANS,N,K,ALPHA,A,LDA,BETA,C,LDC***)**
CHARACTER*1	*UPLO,TRANS*
INTEGER	*N,K,LDA,LDC*
REAL	*ALPHA,BETA*
COMPLEX	*A(LDA,*),C(LDC,*)*

SUBROUTINE	**ZHERK(***UPLO,TRANS,N,K,ALPHA,A,LDA,BETA,C,LDC***)**
CHARACTER*1	*UPLO,TRANS*
INTEGER	*N,K,LDA,LDC*
DOUBLE PRECISION	*ALPHA,BETA*
COMPLEX*16	*A(LDA,*),C(LDC,*)*

Description

The **CHERK** or **ZHERK** subroutine performs one of the hermitian rank k operations:

C := alpha * *A* * conjg(*A*') + beta * *C*

or

C := alpha * conjg(*A*') * *A* + beta * *C*

where alpha and beta are real scalars, *C* is an *N* by *N* hermitian matrix, and *A* is an *N* by *K* matrix in the first case and a *K* by *N* matrix in the second case.

Parameters

UPLO

On entry, *UPLO* specifies whether the upper or lower triangular part of the array *C* is to be referenced as follows:

UPLO = 'U' or 'u'
Only the upper triangular part of *C* is to be referenced.

UPLO = 'L' or 'l'
Only the lower triangular part of *C* is to be referenced.

Unchanged on exit.

TRANS

On entry, *TRANS* specifies the operation to be performed as follows:

TRANS = 'N' or 'n'
C := alpha * *A* * conjg(*A'*) + beta * *C*

TRANS = 'C' or 'c'
C := alpha * conjg(*A'*) * *A* + beta * *C*

Unchanged on exit.

N	On entry, *N* specifies the order of the matrix *C*. *N* must be at least zero. Unchanged on exit.
K	On entry with *TRANS* = 'N' or 'n', *K* specifies the number of columns of the matrix *A*, and on entry with *TRANS* = 'C' or 'c', *K* specifies the number of rows of the matrix *A*. *K* must be at least zero. Unchanged on exit.
ALPHA	On entry, *ALPHA* specifies the scalar alpha. Unchanged on exit.
A	An array of dimension (*LDA*, *KA*), where *KA* is *K* when *TRANS* = 'N' or 'n', and is *N* otherwise. On entry with *TRANS* = 'N' or 'n', the leading *N* by *K* part of the array *A* must contain the matrix *A*, otherwise the leading *K* by *N* part of the array *A* must contain the matrix *A*. Unchanged on exit.
LDA	On entry, *LDA* specifies the first dimension of *A* as declared in the calling (sub) program. When *TRANS* = 'N' or 'n', *LDA* must be at least max(1, *N*), otherwise *LDA* must be at least max(1, *K*). Unchanged on exit.
BETA	On entry, *BETA* specifies the scalar beta. Unchanged on exit.
C	An array of dimension (*LDC*, *N*). On entry with *UPLO* = 'U' or 'u', the leading *N* by *N* upper triangular part of the array *C* must contain the upper triangular part of the hermitian matrix and the strictly lower triangular part of *C* is not referenced. On exit, the upper triangular part of the array *C* is overwritten by the upper triangular part of the updated matrix. On entry with *UPLO* = 'L' or 'l', the leading *N* by *N* lower triangular part of the array *C* must contain the lower triangular part of the hermitian matrix and the strictly upper triangular part of *C* is not referenced. On exit, the lower triangular part of the array *C* is overwritten by the lower triangular part of the updated matrix. Note that the imaginary parts of the diagonal elements need not be set, they are assumed to be zero, and on exit they are set to zero.
LDC	On entry, *LDC* specifies the first dimension of *C* as declared in the calling (sub) program. *LDC* must be at least max(1, *N*). Unchanged on exit.

SSYR2K, DSYR2K, CSYR2K or ZSYR2K Subroutine

Purpose

Performs symmetric rank 2k operations.

Library

BLAS Library (**libblas.a**)

FORTRAN Syntax

SUBROUTINE	**SSYR2K(***UPLO,TRANS,N,K,ALPHA,A,LDA,B,LDB,BETA, C,LDC***)**
CHARACTER*1	*UPLO,TRANS*
INTEGER	*N,K,LDA,LDB,LDC*
REAL	*ALPHA,BETA*
REAL	*A(LDA,*),B(LDB,*),C(LDC,*)*
SUBROUTINE	**DSYR2K(***UPLO,TRANS,N,K,ALPHA,A,LDA,B,LDB,BETA, C,LDC***)**

CHARACTER*1	*UPLO,TRANS*
INTEGER	*N,K,LDA,LDB,LDC*
DOUBLE PRECISION	*ALPHA,BETA*
DOUBLE PRECISION	*A(LDA,*),B(LDB,*),C(LDC,*)*
SUBROUTINE	**CSYR2K(***UPLO,TRANS,N,K,ALPHA,A,LDA,B,LDB,BETA, C,LDC*)
CHARACTER*1	*UPLO,TRANS*
INTEGER	*N,K,LDA,LDB,LDC*
COMPLEX	*ALPHA,BETA*
COMPLEX	*A(LDA,*),B(LDB,*),C(LDC,*)*
SUBROUTINE	**ZSYR2K(***UPLO,TRANS,N,K,ALPHA,A,LDA,B,LDB,BETA, C,LDC*)
CHARACTER*1	*UPLO,TRANS*
INTEGER	*N,K,LDA,LDB,LDC*
COMPLEX*16	*ALPHA,BETA*
COMPLEX*16	*A(LDA,*),B(LDB,*),C(LDC,*)*

Description

The **SSYR2K**, **DSYR2K**, **CSYR2K** or **ZSYR2K** subroutine performs one of the symmetric rank 2k operations:

C := alpha * A * B' + alpha * B * A' + beta * C

or

C := alpha * A' * B + alpha * B' * A + beta * C

where alpha and beta are scalars, C is an N by N symmetric matrix, and A and B are N by K matrices in the first case and K by N matrices in the second case.

Parameters

UPLO
: On entry, *UPLO* specifies whether the upper or lower triangular part of the array *C* is to be referenced as follows:

UPLO = 'U' or 'u'
Only the upper triangular part of *C* is to be referenced.

UPLO = 'L' or 'l'
Only the lower triangular part of *C* is to be referenced.

Unchanged on exit.

TRANS
: On entry, *TRANS* specifies the operation to be performed as follows:

TRANS = 'N' or 'n'
C := alpha * A * B' + alpha * B * A' + beta * C

TRANS = 'T' or 't'
C := alpha * A' * B + alpha * B' * A + beta * C

Unchanged on exit.

N
: On entry, *N* specifies the order of the matrix *C*. *N* must be at least zero. Unchanged on exit.

K
: On entry with *TRANS* = 'N' or 'n', *K* specifies the number of columns of the matrices *A* and *B*, and on entry with *TRANS* = 'T' or 't', *K* specifies the

number of rows of the matrices *A* and *B*. *K* must be at least zero. Unchanged on exit.

ALPHA On entry, *ALPHA* specifies the scalar alpha. Unchanged on exit.

A An array of dimension (*LDA*, *KA*), where *KA* is *K* when *TRANS* = 'N' or 'n', and is *N* otherwise. On entry with *TRANS* = 'N' or 'n', the leading *N* by *K* part of the array *A* must contain the matrix *A*, otherwise the leading *K* by *N* part of the array *A* must contain the matrix *A*. Unchanged on exit.

LDA On entry, *LDA* specifies the first dimension of *A* as declared in the calling (sub) program. When *TRANS* = 'N' or 'n', *LDA* must be at least max(1, *N*); otherwise *LDA* must be at least max(1, *K*). Unchanged on exit.

B An array of dimension (*LDB*, *KB*), where *KB* is *K* when *TRANS* = 'N' or 'n', and is *N* otherwise. On entry with *TRANS* = 'N' or 'n', the leading *N* by *K* part of the array *B* must contain the matrix *B*, otherwise the leading *K* by *N* part of the array *B* must contain the matrix *B*. Unchanged on exit.

LDB On entry, *LDB* specifies the first dimension of *B* as declared in the calling (sub) program. When *TRANS* = 'N' or 'n', *LDB* must be at least max(1, *N*); otherwise *LDB* must be at least max(1, *K*). Unchanged on exit.

BETA On entry, *BETA* specifies the scalar beta. Unchanged on exit.

C An array of dimension (*LDC*, *N*). On entry with *UPLO* = 'U' or 'u', the leading *N* by *N* upper triangular part of the array *C* must contain the upper triangular part of the symmetric matrix and the strictly lower triangular part of *C* is not referenced. On exit, the upper triangular part of the array *C* is overwritten by the upper triangular part of the updated matrix. On entry with *UPLO* = 'L' or 'l', the leading *N* by *N* lower triangular part of the array *C* must contain the lower triangular part of the symmetric matrix and the strictly upper triangular part of *C* is not referenced. On exit, the lower triangular part of the array *C* is overwritten by the lower triangular part of the updated matrix.

LDC On entry, *LDC* specifies the first dimension of *C* as declared in the calling (sub) program. *LDC* must be at least max(1, *N*). Unchanged on exit.

CHER2K or ZHER2K Subroutine

Purpose

Performs hermitian rank 2k operations.

Library

BLAS Library (**libblas.a**)

FORTRAN Syntax

SUBROUTINE	**CHER2K(***UPLO,TRANS,N,K,ALPHA,A,LDA,B,LDB,C,LDC***)**
CHARACTER*1	*UPLO,TRANS*
INTEGER	*N,K,LDA,LDB,LDC*
REAL	*BETA*

COMPLEX	*ALPHA*
COMPLEX	*A(LDA,*),B(LDB,*),C(LDC,*)*
SUBROUTINE	**ZHER2K**(*UPLO,TRANS,N,K,ALPHA,A,LDA,B,LDB,C,LDC*)
CHARACTER*1	*UPLO,TRANS*
INTEGER	*N,K,LDA,LDB,LDC*
DOUBLE PRECISION	*BETA*
COMPLEX*16	*ALPHA*
COMPLEX*16	*A(LDA,*),B(LDB,*),C(LDC,*)*

Description

The **CHER2K** or **ZHER2K** subroutine performs one of the hermitian rank 2k operations:

C := alpha * A * conjg(B') + conjg(alpha) * B * conjg(A') + beta * C

or

C := alpha * conjg(A') * B + conjg(alpha) * conjg(B') * A + beta * C

where alpha and beta are scalars with beta real, C is an N by N hermitian matrix, and A and B are N by K matrices in the first case and K by N matrices in the second case.

Parameters

UPLO
: On entry, *UPLO* specifies whether the upper or lower triangular part of the array *C* is to be referenced as follows:

  UPLO = 'U' or 'u'
  Only the upper triangular part of *C* is to be referenced.

  UPLO = 'L' or 'l'
  Only the lower triangular part of *C* is to be referenced.

  Unchanged on exit.

TRANS
: On entry, *TRANS* specifies the operation to be performed as follows:

  TRANS = 'N' or 'n'
  C := alpha * A * conjg(B') + conjg(alpha) * B * conjg(A') + beta * C

  TRANS = 'C' or 'c'
  C := alpha * conjg(A') * B + conjg(alpha) * conjg(B') * A + beta * C

  Unchanged on exit.

N
: On entry, *N* specifies the order of the matrix *C*. *N* must be at least zero. Unchanged on exit.

K
: On entry with *TRANS* = 'N' or 'n', *K* specifies the number of columns of the matrices *A* and *B*, and on entry with *TRANS* = 'C' or 'c', *K* specifies the number of rows of the matrices *A* and *B*. *K* must be at least zero. Unchanged on exit.

ALPHA
: On entry, *ALPHA* specifies the scalar alpha. Unchanged on exit.

A
: An array of dimension (*LDA*, *KA*), where *KA* is *K* when *TRANS* = 'N' or 'n', and is *N* otherwise. On entry with *TRANS* = 'N' or 'n', the leading *N* by *K*

	part of the array *A* must contain the matrix *A*, otherwise the leading *K* by *N* part of the array *A* must contain the matrix *A*. Unchanged on exit.
LDA	On entry, *LDA* specifies the first dimension of *A* as declared in the calling (sub) program. When *TRANS* = 'N' or 'n', *LDA* must be at least max(1, *N*); otherwise *LDA* must be at least max(1, *K*). Unchanged on exit.
B	An array of dimension (*LDB*, *KB*), where *KB* is *K* when *TRANS* = 'N' or 'n', and is *N* otherwise. On entry with *TRANS* = 'N' or 'n', the leading *N* by *K* part of the array *B* must contain the matrix *B*, otherwise the leading *K* by *N* part of the array *B* must contain the matrix *B*. Unchanged on exit.
LDB	On entry, *LDB* specifies the first dimension of *B* as declared in the calling (sub) program. When *TRANS* = 'N' or 'n', *LDB* must be at least max(1, *N*); otherwise *LDB* must be at least max(1, *K*). Unchanged on exit.
BETA	On entry, *BETA* specifies the scalar beta. Unchanged on exit.
C	An array of dimension (*LDC*, *N*). On entry with *UPLO* = 'U' or 'u', the leading *N* by *N* upper triangular part of the array *C* must contain the upper triangular part of the hermitian matrix and the strictly lower triangular part of *C* is not referenced. On exit, the upper triangular part of the array *C* is overwritten by the upper triangular part of the updated matrix. On entry with *UPLO* = 'L' or 'l', the leading *N* by *N* lower triangular part of the array *C* must contain the lower triangular part of the hermitian matrix and the strictly upper triangular part of *C* is not referenced. On exit, the lower triangular part of the array *C* is overwritten by the lower triangular part of the updated matrix. Note that the imaginary parts of the diagonal elements need not be set, they are assumed to be zero, and on exit they are set to zero.
LDC	On entry, *LDC* specifies the first dimension of *C* as declared in the calling (sub) program. *LDC* must be at least max(1, *N*). Unchanged on exit.

STRMM, DTRMM, CTRMM or ZTRMM Subroutine

Purpose

Performs matrix–matrix operations on triangular matrixes.

Library

BLAS Library (**libblas.a**)

FORTRAN Syntax

SUBROUTINE	**STRMM**(*SIDE,UPLO,TRANSA,DIAG,M,N,ALPHA,A,LDA,B,LDB*)
CHARACTER*1	*SIDE,UPLO,TRANSA,DIAG*
INTEGER	*M,N,LDA,LDB*
REAL	*ALPHA*
REAL	*A*(*LDA*,*),*B*(*LDB*,*)
SUBROUTINE	**DTRMM**(*SIDE,UPLO,TRANSA,DIAG,M,N,ALPHA,A,LDA,B,LDB*)
CHARACTER*1	*SIDE,UPLO,TRANSA,DIAG*
INTEGER	*M,N,LDA,LDB*

DOUBLE PRECISION	*ALPHA*
DOUBLE PRECISION	*A(LDA,*),B(LDB,*)*
SUBROUTINE	**CTRMM**(*SIDE,UPLO,TRANSA,DIAG,M,N,ALPHA,A,LDA,B, LDB*)
CHARACTER*1	*SIDE,UPLO,TRANSA,DIAG*
INTEGER	*M,N,LDA,LDB*
COMPLEX	*ALPHA*
COMPLEX	*A(LDA,*),B(LDB,*)*
SUBROUTINE	**ZTRMM**(*SIDE,UPLO,TRANSA,DIAG,M,N,ALPHA,A,LDA,B, LDB*)
CHARACTER*1	*SIDE,UPLO,TRANSA,DIAG*
INTEGER	*M,N,LDA,LDB*
COMPLEX*16	*ALPHA*
COMPLEX*16	*A(LDA,*),B(LDB,*)*

Description

The **STRMM**, **DTRMM**, **CTRMM** or **ZTRMM** subroutine performs one of the matrix–matrix operations:

B := alpha * op(*A*) * *B*

or

B := alpha * *B* * op(*A*)

where alpha is a scalar, *B* is an *M* by *N* matrix, A is a unit, or non–unit, upper or lower triangular matrix, and op(*A*) is either op(*A*) = *A* or op(*A*) = *A*'.

Parameters

SIDE

On entry, *SIDE* specifies whether op(*A*) multiplies *B* from the left or right as follows:

SIDE = 'L' or 'l'
B := alpha * op(*A*) * *B*

SIDE = 'R' or 'r'
B := alpha * *B* * op(*A*)

Unchanged on exit.

UPLO

On entry, *UPLO* specifies whether the matrix *A* is an upper or lower triangular matrix as follows:

UPLO = 'U' or 'u'
A is an upper triangular matrix.

UPLO = 'L' or 'l'
A is a lower triangular matrix.

Unchanged on exit.

TRANSA

On entry, *TRANSA* specifies the form of op(*A*) to be used in the matrix multiplication as follows:

TRANSA = 'N' or 'n'
op(*A*) = *A*

TRANSA = 'T' or 't'
op(*A*) = *A*'

TRANSA = 'C' or 'c'
op(*A*) = *A*'

Unchanged on exit.

DIAG On entry, *DIAG* specifies whether or not *A* is unit triangular as follows:

DIAG = 'U' or 'u'
A is assumed to be unit triangular.

DIAG = 'N' or 'n'
A is not assumed to be unit triangular.

Unchanged on exit.

M On entry, *M* specifies the number of rows of *B*. *M* must be at least zero. Unchanged on exit.

N On entry, *N* specifies the number of columns of *B*. *N* must be at least zero. Unchanged on exit.

ALPHA On entry, *ALPHA* specifies the scalar alpha. When alpha is zero then *A* is not referenced and *B* need not be set before entry. Unchanged on exit.

A An array of dimension (*LDA*, k), where k is *M* when *SIDE* = 'L' or 'l' and is *N* when *SIDE* = 'R' or 'r'. On entry with *UPLO* = 'U' or 'u', the leading k by k upper triangular part of the array *A* must contain the upper triangular matrix and the strictly lower triangular part of *A* is not referenced. On entry with *UPLO* = 'L' or 'l', the leading k by k lower triangular part of the array *A* must contain the lower triangular matrix and the strictly upper triangular part of *A* is not referenced. Note that when *DIAG* = 'U' or 'u', the diagonal elements of *A* are not referenced either, but are assumed to be unity. Unchanged on exit.

LDA On entry, *LDA* specifies the first dimension of *A* as declared in the calling (sub) program. When *SIDE* = 'L' or 'l' then *LDA* must be at least max(1, *M*), when *SIDE* = 'R' or 'r' then *LDA* must be at least max(1, *N*). Unchanged on exit.

B An array of dimension (*LDB*, *N*). On entry, the leading *M* by *N* part of the array *B* must contain the matrix *B*, and on exit is overwritten by the transformed matrix.

LDB On entry, *LDB* specifies the first dimension of *B* as declared in the calling (sub) program. *LDB* must be at least max(1, *M*). Unchanged on exit.

STRSM, DTRSM, CTRSM or ZTRSM Subroutine

Purpose

Solves certain matrix equations.

Library

BLAS Library (**libblas.a**)

FORTRAN Syntax

SUBROUTINE	**STRSM**(*SIDE,UPLO,TRANSA,DIAG,M,N,ALPHA,A,LDA,B, LDB*)
CHARACTER*1	*SIDE,UPLO,TRANSA,DIAG*
INTEGER	*M,N,LDA,LDB*
REAL	*ALPHA*
REAL	*A(LDA,*),B(LDB,*)*
SUBROUTINE	**DTRSM**(*SIDE,UPLO,TRANSA,DIAG,M,N,ALPHA,A,LDA,B, LDB*)
CHARACTER*1	*SIDE,UPLO,TRANSA,DIAG*
INTEGER	*M,N,LDA,LDB*
DOUBLE PRECISION	*ALPHA*
DOUBLE PRECISION	*A(LDA,*),B(LDB,*)*
SUBROUTINE	**CTRSM**(*SIDE,UPLO,TRANSA,DIAG,M,N,ALPHA,A,LDA,B, LDB*)
CHARACTER*1	*SIDE,UPLO,TRANSA,DIAG*
INTEGER	*M,N,LDA,LDB*
COMPLEX	*ALPHA*
COMPLEX	*A(LDA,*),B(LDB,*)*
SUBROUTINE	**ZTRSM**(*SIDE,UPLO,TRANSA,DIAG,M,N,ALPHA,A,LDA,B, LDB*)
CHARACTER*1	*SIDE,UPLO,TRANSA,DIAG*
INTEGER	*M,N,LDA,LDB*
COMPLEX*16	*ALPHA*
COMPLEX*16	*A(LDA,*),B(LDB,*)*

Description

The **STRSM, DTRSM, CTRSM** or **ZTRSM** subroutine solves one of the matrix equations:

op(*A*) * X = alpha * *B*

or

X * op(*A*) = alpha * *B*

where alpha is a scalar, X and *B* are *M* by *N* matrices, *A* is a unit, or non–unit, upper or lower triangular matrix, and op(*A*) is either op(*A*) = *A* or op(*A*) = *A*'. The matrix X is overwritten on *B*.

Parameters

SIDE — On entry, *SIDE* specifies whether op(*A*) appears on the left or right of X as follows:

SIDE = 'L' or 'l'
op(*A*) * X = alpha * *B*

SIDE = 'R' or 'r'
X * op(*A*) = alpha * *B*

Unchanged on exit.

UPLO
On entry, *UPLO* specifies whether the matrix *A* is an upper or lower triangular matrix as follows:

UPLO = 'U' or 'u'
A is an upper triangular matrix.

UPLO = 'L' or 'l'
A is a lower triangular matrix.

Unchanged on exit.

TRANSA
On entry, *TRANSA* specifies the form of op(*A*) to be used in the matrix multiplication as follows:

TRANSA = 'N' or 'n'
op(*A*) = *A*

TRANSA = 'T' or 't'
op(*A*) = *A*'

TRANSA = 'C' or 'c'
op(*A*) = *A*'

Unchanged on exit.

DIAG
On entry, *DIAG* specifies whether or not *A* is unit triangular as follows:

DIAG = 'U' or 'u'
A is assumed to be unit triangular.

DIAG = 'N' or 'n'
A is not assumed to be unit triangular.

Unchanged on exit.

M
On entry, *M* specifies the number of rows of *B*. *M* must be at least zero. Unchanged on exit.

N
On entry, *N* specifies the number of columns of *B*. *N* must be at least zero. Unchanged on exit.

ALPHA
On entry, *ALPHA* specifies the scalar alpha. When alpha is zero then *A* is not referenced and *B* need not be set before entry. Unchanged on exit.

A
An array of dimension (*LDA*, k), where k is *M* when *SIDE* = 'L' or 'l' and is *N* when *SIDE* = 'R' or 'r'. On entry with *UPLO* = 'U' or 'u', the leading k by k upper triangular part of the array *A* must contain the upper triangular matrix and the strictly lower triangular part of *A* is not referenced. On entry with *UPLO* = 'L' or 'l', the leading k by k lower triangular part of the array *A* must contain the lower triangular matrix and the strictly upper triangular part of *A* is not referenced. Note that when *DIAG* = 'U' or 'u', the diagonal elements of *A* are not referenced, but are assumed to be unity. Unchanged on exit.

LDA
On entry, *LDA* specifies the first dimension of *A* as declared in the calling (sub) program. When *SIDE* = 'L' or 'l', *LDA* must be at least max(1, *M*); when *SIDE* = 'R' or 'r', *LDA* must be at least max(1, *N*). Unchanged on exit.

B	An array of dimension (*LDB*, *N*). On entry, the leading *M* by *N* part of the array *B* must contain the right–hand side matrix *B*, and on exit is overwritten by the solution matrix X.
LDB	On entry, *LDB* specifies the first dimension of *B* as declared in the calling (sub) program. *LDB* must be at least max(1, *M*). Unchanged on exit.

Level 3: matrix–matrix operations

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

Appendix B. ODM Error Codes

When an ODM subroutine fails, 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 *General Programming Concepts.*

ODMI_BAD_LOCK
Cannot set a lock on the file. Check path name and permissions.

ODMI_BAD_TIMEOUT
The timeout 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 linked to 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 sufficent 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. 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.

ODM Error Codes

Appendix C. List of X.25 API Error Codes

List of X.25-Specific Error Codes

For X.25-specific error conditions, **x25_errno** is set to one of the following values:

X25ACKREQ One or more packets require acknowledgement. Issue **x25_ack** before continuing.

X25AUTH The calling application does not have system permission to control the status of the link.

X25AUTHCTR The application does not have permission to remove this counter because it is not the application that issued the corresponding **x25_ctr_get**.

X25AUTHLISTEN The application cannot listen to this name, because the corresponding entry in the routing list has a user name that excludes the user running the application. Use another routing list name, or change the user name in the routing list entry.

X25BADCONNID The connection identifier is invalid.

X25BADDEVICE The X.25 port name is invalid.

X25BADID The connection identifier or listen identifier is invalid.

X25BADLISTENID The listen identifier is invalid.

X25CALLED The called address is invalid. Check that the address is correct and is a NULL-terminated string.

X25CALLING The calling address is invalid. Check that the address is correct and is a NULL-terminated string.

X25CTRUSE The counter has a non-zero value.

X25INIT X.25 is already initialized for this X.25 port, so cannot be initialized again.

X25INVCTR The specified counter does not exist. (In the case of **x25_ctr_wait**, the counter is one of an array of counters.)

X25INVFAC An optional facility requested is invalid. Check **cb_fac_struct**.

X25INVMON The monitoring mode is invalid.

X25LINKUP The X.25 port is already connected.

X25LINKUSE The X.25 port still has virtual circuits established; it may still be in use. Either free all virtual circuits or disconnect the port using the override.

X25LONG The parameter is too long. Check each of the parameters for this subroutine.

X.25 Application Error Codes

X25MAXDEVICE	Attempts have been made to connect more X.25 ports than are available. Check the **smit** configuration to see how many ports are available.
X25MONITOR	X.25 traffic on this X.25 port is already being monitored by another application. The other application must stop monitoring before any other application can start it.
X25NAMEUSED	Calls for this name are already being listened for.
X25NOACKREQ	No packets currently require acknowledgement.
X25NOCARD	The X.25 adapter is either not installed or not functioning.
X25NOCTRS	No counters are available.
X25NODATA	No data is has arrived for this connection identifier. Issue **x25_ctr_wait** to be notified when data arrives.
X25NODEVICE	The X.25 device driver is either not installed or not functioning.
X25NOLINK	The X.25 port is not connected. Issue **x25_link_connect**, or use **xmanage** to connect it.
X25NONAME	The name is not in the routing list. Add the name or use one that is already in the list.
X25NOSUCHLINK	The X.25 port does not exist. Check the **smit** configuration.
X25NOTINIT	The application has not initialized X.25 communications. Issue **x25_init**.
X25NOTPVC	This is not defined as a permanent virtual circuit (PVC). Check the **smit** configuration.
X25PROTOCOL	An X.25 protocol error occurred.
X25PVCUSED	This permanent virtual circuit (PVC) is already allocated to another application. The other application must free the PVC before it can be used.
X25RESETCLEAR	The call was reset or cleared during processing. Issue **x25_receive** to obtain the reset-indication or clear-indication packet. Then issue **x25_reset_confirm** or **x25_clear_confirm**, as necessary.
X25SYSERR	An error occurred that was not an X.25 error. Check the value of **errno**.
X25TABLE	The routing list cannot be updated because **xroute** is using it. Try again after **xroute** has completed.
X25TIMEOUT	A timeout problem occurred.
X25TOOMANYVCS	No virtual circuits are free on the listed X.25 ports.

X25TRUNCTX The packet size is too big for internal buffers, so data cannot be sent.

List of System Error Codes

For non-X.25-specific error conditions, **x25_errno** is set to `X25SYSERR` and **errno** is set to one of the following values:

EFAULT Bad address pointer.

EINTR A signal was caught during the call.

EIO An I/O error occurred.

ENOMEM Could not allocate memory for device information.

ENOSPC There are no buffers available in the pool.

EPERM Calling application does not have sufficient authorization.

Index

Special Characters

A

B

C

D

E

F

G

J

K

L

M

N

O

P

S

T

X

Y

Reader's Comment Form

AIX Calls and Subroutines Reference for IBM RISC System/6000: Volumes 1 and 2

SC23–2198–00

Please use this form only to identify publication errors or to request changes in publications. Your comments assist us in improving our publications. Direct any requests for additional publications, technical questions about IBM systems, changes in IBM programming support, and so on, to your IBM representative or to your IBM-approved remarketer. You may use this form to communicate your comments about this publication, its organization, or subject matter, with the understanding that IBM may use or distribute whatever information you supply in any way it believes appropriate without incurring any obligation to you.

☐ If your comment does not need a reply (for example, pointing out a typing error), check this box and do not include your name and address below. If your comment is applicable, we will include it in the next revision of the manual.

☐ If you would like a reply, check this box. Be sure to print your name and address below.

Page	Comments

Please contact your IBM representative or your IBM–approved remarketer to request additional publications.

Please print

Date ____________

Your Name ________________________________

Company Name ________________________________

Mailing Address ________________________________

____

____

Phone No. (   ) ____________________
Area Code

No postage necessary if mailed in the U.S.A

NO POSTAGE
NECESSARY
IF MAILED
IN THE
UNITED STATES

BUSINESS REPLY MAIL

FIRST CLASS PERMIT NO. 40 ARMONK, NEW YORK

POSTAGE WILL BE PAID BY ADDRESSEE

International Business Machines Corporation
Department 997, Building 997
11400 Burnet Rd.
Austin, Texas 78758–3493

Fold Fold

Cut or Fold Along Line

Fold and Tape Please Do Not Staple Fold and Tape

International Business Machines
Corporation
11400 Burnet Road
Austin, Texas 78758–3493

Printed in the
United States of America

SC23-2198-00

SC23-2198-00