AIX Version 3 for RISC System/6000™

Calls and Subroutines Reference: Base Operating System

Volume 2

SC23-2

AIX Version 3 for RISC System/6000™

Calls and Subroutines Reference: Base Operating System

Volume 2

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

The following trademarks and acknowledgements apply to this information:

AIX is a trademark of International Business Machines Corporation.

AIXwindows is a trademark of International Business Machines Corporation.

Apollo is a trademark of Apollo Computer, Inc.

IBM is a registered trademark of International Business Machines Corporation.

NCK is a trademark of Apollo Computer, Inc.

NCS is a trademark of Apollo Computer, Inc.

Network Computing Kernel is a trademark of Apollo Computer, Inc.

Network Computing System is a trademark of Apollo Computer, Inc.

Network File System and NFS are trademarks of Sun Microsystems, Inc.

POSIX is a trademark of the Institute of Electrical and Electronic Engineers (IEEE).

RISC System/6000 is a trademark of International Business Machines Corporation.

SNA 3270 is a trademark of International Business Machines Corporation.

UNIX was developed and licensed by AT&T and is a registered trademark of AT&T Corporation.

X/OPEN is a trademark of X/OPEN Company Limited.

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

AIX 3270 Host Connection Program (HCON)

BREAK Statement

Purpose

Interrupts a loop in a LAF script.

Syntax

BREAK;

Description

The **BREAK** statement interrupts the execution of the innermost enclosing **WHILE** or **REPEAT–UNTIL** statement. Execution continues with the statement following the **WHILE** or **REPEAT–UNTIL** statement. The **BREAK** statement is one of the script statements in the LAF language that are used to compose a LAF script.

Example

The statements below execute a loop. If a time out for the **WAIT** statement occurs, the **BREAK** statement terminates the repeat loop and executes the next statement:

```
REPEAT
  DO
  MATCHAT(1,1,'VM/370?ONLINE');
    IF (NOT MATCH) DO /* if not found */
      WAIT(2);        /* wait for update to display or timeout */
    IF(TIMEOUT)
      BREAK;
    END;
  END;
UNTIL(MATCH);
```

Implementation Specifics

The **BREAK** statement is part of the Logon Assist Feature of the AIX 3270 Host Connection Program/6000 (HCON).

Related Information

How To Use a Logon Assist Feature Script, Understanding the Logon Assist Feature (LAF) in *Communications Programming Concepts*.

HCON Overview for Programming in *Communications Programming Concepts*.

cfxfer Function

Purpose

Checks the status of the programmatic File Transfer.

Library

File Transfer Library (**libfxfer.a**)

C Syntax

#include <fxfer.h>
cfxfer(*sxfer*)
struct *fxs* *sxfer;

Pascal Syntax

%include fxfer.inc

%include fxhfile.inc

function pcfxfer(var sxfer : fxs) **: integer; external;**

FORTRAN Syntax

INTEGER FCFXFER
EXTERNAL FCFXFER
CHARACTER*XX SRC, DST, TIME*
INTEGER *BYTCNT, STAT*
INTEGER *ERRNO*
RC = **FCFXFER** (*SRC, DST, BYTCNT, STAT, ERRNO, TIME, RC*)

Description

The **cfxfer** function returns the status of the file transfer request made by the **fxfer** function. This function must be called once for each file transfer request. The **cfxfer** function places the status in the structure specified by the *sxfer* parameter for C and Pascal. For FORTRAN, status is placed in each corresponding parameter.

Each individual file transfer and file transfer status completes the requests in the order the requests are made. If multiple asynchronous requests are made:

- To a single host session, the **cfxfer** function returns the status of each request in the same order the requests are made
- To more than one host session, the **cfxfer** function returns the status of each request in the order it is completed.

If the file transfer is run asynchronously and the **cfxfer** function is immediately called, the function returns a status not available (–2) code. An application performing a file transfer should not call the **cfxfer** function until an error (–1) or ready status (0) is returned. The application program can implement the status check in a **FOR LOOP** or a **WHILE LOOP** and wait for a –1 (negative one) or 0 (zero) to occur.

C Parameter

sxfer Specifies a record of type *fxs* defined in the **fxfer.h** file.

The C struct *fxs* is defined as follows:

```
struct fxs
    int     fxs_bytcnt;
    char   *fxs_src;
    char   *fxs_dst;
    char   *fxs_ctime;
    int     fxs_stat;
    int     fxs_errno;
]c.;
```

Pascal Parameter

Sfxfer Specifies a record of type *fxs* within the **fxfer.inc** file.

The Pascal *fxs* record format is as follows:

```
fxs =  record
       fxs_bytcnt : integer;
       fxs_src : stringptr;
       fxs_dst : stringptr;
       fxs_ctime : stringptr;
       fxs_stat : integer;
       fxs_errno : integer;
end;
```

C and Pascal fxs Field Descriptions

fxc_bytcnt Indicates the number of bytes transferred.

fxc_src Points to a static buffer containing the source file name. The static buffer is overwritten by each call.

fxc_dst Points to a static buffer containing the destination file name. The static buffer is overwritten by each call.

fxs_ctime Specifies the time the destination file is created relative to Greenwich Mean Time (GMT), midnight on January 1, 1970.

fxs_stat Specifies the status of the file transfer request.

fxs_errno Specifies the error number that results from an error in a system call.

FORTRAN Parameters

SRC Specifies a character array of XX length containing the source file name

DST Specifies a character array of XX length containing the destination file name.

BYTCNT Indicates the number of bytes transferred.

STAT Specifies the status of the file transfer request.

ERRNO Specifies the error number that results from an error in a system call

TIME Specifies the time the destination file is created.

Return Value

The **cfxfer** function returns the following:

0 (zero), if status is available.

–1, if an I/O error occurs on the **fx_statxxxxxx** status file and the status cannot be obtained

–2, if status is not available or if there are no outstanding file transfer requests.

The **fx_statxxxxxx** status file contains the status of each file transfer request made by the application program. The **fxfer** function fills in the **xxxxxx** portion of the **fx_stat** file based on random letter generation and places the file in the $HOME directory.

Implementation Specifics

The **cfxfer** function is part of the AIX 3270 Host Connection Program/6000 (HCON).

Files

$HOME/fx_statxxxxxx	Temporary file used for status
/usr/lib/libfxfer.a	Library containing **C**, **FORTRAN**, and **Pascal** interface file transfer functions.
/usr/include/fxfer.h	File transfer include file with structures and definitions.
/usr/include/fxfer.inc	Pascal file transfer include file with structure.
/usr/include/fxconst.inc	Pascal file transfer function constants.
/usr/include/fxhfile.inc	Pascal file transfer invocation include file.

Related Information

The **fxfer** command, **fxfer** function.

HCON Overview for Programming, Understanding File Transfer Programming, File Transfer Program Interface Error Codes in *Communications Programming Concepts*.

DEBUG Statement

Purpose

Enables debugging messages in a Logon Assist Feature (LAF) script.

Syntax

DEBUG;

Description

The **DEBUG** statement enables debugging messages in a LAF script. The **DEBUG** statement is one of the script statements in the LAF language that are used to compose a LAF script. The **DEBUG** statement operates on successive LAF statements. Debugging occurs up to the end of the LAF script or when a **NODEBUG** statement is encountered. The messages are written to the standard error. This statement should only be used in a script linked with the **tlaf** test program. If a script containing the **DEBUG** statement is linked with the file transfer program or an application using the HCON API, unpredictable results occur.

Implementation Specifics

The **DEBUG** statement is part of the Logon Assist Feature of the AIX 3270 Host Connection Program/6000 (HCON).

Related Information

How To Use a Logon Assist Feature Script, Understanding the Logon Assist Feature (LAF) in *Communications Programming Concepts*.

HCON Overview for Programming in *Communications Programming Concepts*.

DO–END Statement

Purpose

Groups Logon Assist Feature (LAF) statements.

Syntax

DO *statementlist* **END**;

Description

The **DO–END** statement is used for grouping LAF statements. The **DO–END** statement is one of the script statements in the LAF language that are used to compose a LAF script.

Expression

statementlist	A statement or statements to be executed that are grouped by a **DO–END** statement.

Example

The statements below search for `CP READ` string on line 24 of the terminal screen. The list waits for a screen update and then looks for the string. If the string cannot be found in two seconds, an exit is performed with a return code of 2.

```
DO
  MATCH(24,1,'CP?READ');
    IF (NOT MATCH) DO /* if not found */
     WAIT(2);         /* wait for update to display or timeout */
     IF(TIMEOUT)
     EXIT(2);         /* exit with error——can't find it */
    END;
END;
```

Implementation Specifics

The **DO–END** statement is part of the Logon Assist Feature of the AIX 3270 Host Connection Program/6000 (HCON).

Related Information

How To Use a Logon Assist Feature Script, Understanding the Logon Assist Feature (LAF) in *Communications Programming Concepts.*

HCON Overview for Programming in *Communications Programming Concepts.*

EXIT Statement

Purpose

Terminates the execution of a Logon Assist Feature (LAF) script.

Syntax

EXIT(*number***);**

Description

The **EXIT** statement halts the execution of a LAF script. The **EXIT** statement is one of the script statements in the LAF language that are used to compose a LAF script. Upon termination a specified return value is passed to the program that uses the LAF script. If a LAF script exits with a successful logon or logoff the return value is zero (0). The **EXIT** statement allows for abnormal exits whose return values indicate the area in the LAF script that failed.

Expression

number Specifies the return code value.

Example

This statement terminates the script with a return code of 3 if a time out occurs:

```
IF(TIMEOUT) EXIT(3);
```

Implementation Specifics

The **EXIT** statement is part of the Logon Assist Feature of the AIX 3270 Host Connection Program/6000 (HCON).

Related Information

How To Use a Logon Assist Feature Script, Understanding the Logon Assist Feature (LAF) in *Communications Programming Concepts.*

HCON Overview for Programming in *Communications Programming Concepts.*

FINISH Statement

Purpose

Ends a Logon Assist Feature (LAF) script.

Syntax

FINISH;

Description

The **FINISH** statement ends a LAF script. The **FINISH** statement is one of the script statements in the LAF language that are used to compose a LAF script.

Each LAF script requires one **FINISH** statement, and it must be the last statement in the script. The **FINISH** statement implies that a zero (0) return value is passed back to the program using the LAF script (**fxfer** function or API functions). This return value denotes a successful logon or logoff. Any other return value is interpreted by the program using the LAF script as unsuccessful logon or logoff.

Implementation Specifics

The **FINISH** statement is part of the Logon Assist Feature of the AIX 3270 Host Connection Program/6000 (HCON).

Related Information

The **EXIT** statement.

How To Use a Logon Assist Feature Script, Understanding the Logon Assist Feature (LAF) in *Communications Programming Concepts.*

HCON Overview for Programming in *Communications Programming Concepts.*

fxfer Function

Purpose

Initiates a file transfer from within a program executing in AIX.

Library

File Transfer Library (libfxfer.a)

C Syntax

#include <fxfer.h>

fxfer (*xfer, sessionname*)

struct fxc **xfer*;

char **sessionname*;

Pascal Syntax

%include /usr/include/fxfer.inc
%include /usr/include/fxhfile.inc
%include /usr/include/fxconst.inc
function pfxfer
(var *xfer* : *fxc*; *sessionname* : *stringptr*) :
integer; external;

FORTRAN Syntax

INTEGER *FFXFER*
EXTERNAL *FFXFER*
CHARACTER*XX *SRCF, DSTF, LOGID, SESSIONNAME*
INT *FLAGS, RECL, BLKSIZE, SPACE, INCR, UNIT, RC*
RC = **FFxfer** (*SRCF, DSTF, LOGID, FLAGS, RECL, BLKSIZE, SPACE,*
+ *INCR, UNIT, SESSIONNAME*)

Description

The **fxfer** function transfers a file from a specified source to a specified destination. The file transfer is accomplished as follows:

- In the C or Pascal language, the **fxfer** or **pfxfer** function transfers a file specified by the *fxc_src* variable to the file specified by the *fxc_dst* variable. Both variables are defined in the **fxc** structure.
- In the FORTRAN language, the FFxfer function transfers a file specified by the *SRCF* variable to the file specified by the *DSTF* variable.

The file names are character strings. The RISC System/6000 file names must be in AIX format. The host file names must conform to the host naming convention, which must be one of the following formats:

VM/CMS:	filename filetype filemode
MVS/TSO:	data_set_name [(member_name)][/password]

C Parameters

xfer	Specifies a pointer to the **fxc** structure defined in the **fxfer.h** file.

sessionname Points to the name of a session, specifying the host connectivity to be used by the File Transfer Programming Interface. The session name is a single character in the range of a–z. Capital letters are interpreted as lowercase letters. Session variables are defined in a HCON session profile. If the *sessionname* is set to NULL the **fxfer** function assumes you are running in an e789 subshell.

Pascal Parameters

xfer Specifies a record of type **fxc** within the **fxfer.inc** file.

sessionname Points to the name of a session. The sessionname defines the host connectivity to be used by the File Transfer Programming Interface. The session name is a single character in the range of a–z. Capital letters are interpreted as lowercase letters. Session variables are defined in a HCON session profile. If the *sessionname* is set to char(0) the **pfxfer** function assumes you are running in an e789 subshell.

FORTRAN Parameters

SRCF Specifies a character array of XX length containing the source file name.

DSTF Specifies a character array of XX length containing the destination file name.

LOGID Specifies a character array of XX length containing the logon ID.

SESSIONAME Points to the name of a session. The sessionname defines the host connectivity to be used by the File Transfer Programming Interface. The session name is a single character in the range of a–z. Capital letters are interpreted as lowercase letters. Session variables are defined in a HCON session profile. If the *SESSIONNAME* is set to char(0) the FFxfer function assumes you are running in an e789 subshell.

FLAGS Contains the option flags value, which is the sum of the desired option values listed below:

1	Upload
2	Download
4	Translate On
8	Translate Carriage Return Line Feed
16	Replace
32	Append
64	Queue
128	Fixed Length Records

	256	Variable Length Records
	512	Undefined Length (TSO only)
	1024	Host System TSO
	2048	Host System CMS

RECL Specifies the logical record length.

BLKSIZE Specifies the block size.

SPACE Specifies the allocation space.

INCR Specifies the allocation space increment.

UNIT Specifies the unit of allocation, which is:

–1	Specifies the number of TRACKS
–2	Specifies the number of CYLINDERS

Note: All FORTRAN character array strings must be NULL–terminated. For example:

```
SRCF = 'rtfile'//CHAR(0)
```

A positive number indicates the number of bytes to be allocated.

Return Value

If the **fxfer** function is called synchronously, it returns the value zero (0) when the transfer is completed. The application program can then issue a **cfxfer** function call to obtain the status of the file transfer.

If the **fxfer** function is called asynchronously, it returns zero (0) immediately. The application program can issue a **cfxfer** function call to determine when the file transfer is completed and to obtain the status of the file transfer. If the status cannot be reported by the **cfxfer** function due to an I/O error on the **fx_statxxxxxx** status file, the **cfxfer** function returns a –1 (negative one). If the status is not ready, the **cfxfer** function returns a –2 (negative two).

The **fx_statxxxxxx** status file contains the status of each file transfer request made by the application program. The **fxfer** function fills in the **xxxxxx** portion of the **fx_stat** file based on random letter generation and places the file in the $HOME directory.

Implementation Specifics

The **fxfer** function is part of the AIX 3270 Host Connection Program/6000 (HCON).

The **fxfer** function requires one of the following network communication adapters:

- IBM 3270 Connection Adapter plus appropriate cables for attachment to an IBM 3174/3274 Control Unit, IBM 4361 Work Station Adapter, or an IBM 9370 Work Station Subsystem Controller configured for non–SNA distributed function terminal (non–SNA DFT) mode.
- IBM System/370 Host Interface Adapter plus appropriate cables for attachment to an IBM 5088 Graphics Control Unit.

This function requires one of the following IBM System/370 operating system environments be installed on the System/370: VM/SP CMS, VM/XA CMS, MVS/SP TSO/E, or MVS/XA TSO/E.

This function requires that the System/370 IBM Host–Supported File Transfer Program (IND$FILE) be installed on the System/370.

This function is not available for Japanese Language Support.

Files

$HOME/fx_statxxxxxx	Temporary file used for status
/usr/lib/libfxfer.a	Library containing **C**, **FORTRAN**, and **Pascal** interface file transfer functions.
/usr/include/fxfer.h	File transfer include file with structures and definitions.
/usr/include/fxfer.inc	Pascal file transfer include file with structure.
/usr/include/fxconst.inc	Pascal file transfer function constants.
/usr/include/fxhfile.inc	Pascal file transfer invocation include file.

Related Information

The file transfer check status function is the **cfxfer** function.

HCON Overview for Programming, Understanding the File Transfer Program Interface, How to Compile a File Transfer Program, File Transfer Program Interface Error Codes in *Communications Programming Concepts*.

G32ALLOC Function

Purpose

Starts interaction with an AIX API application running simultaneously on the RISC System/6000.

Syntax

G32ALLOC

Description

The **G32ALLOC** function starts a session with an AIX API application by sending a message to the AIX **g32_alloc** system call indicating that the allocation is complete. The **G32ALLOC** function is a HCON API function that can be called by a 370 Assembler applications program.

Return Values

This call sets register 0 (zero), to the following values:

>= 0	Normal return; successful call. The value returned indicates the maximum number of bytes that may be transferred to an AIX application via G32WRITE or received from an AIX application via G32READ.

Example

The following 370 Assembler code example illustrates the use of the host **G32ALLOC** function:

```
L R11,=v(G32DATA)
USING G32DATAD,R11
G32ALLOC                /* Allocate a session */
LTR R0,R0
BNM OK                  /* Normal completion  */
C  R0,G32ESESS  /*Session error       */
BE SESSERR
C  R0,G32ESYS           /* System error       */
BE SYSERR
.
.
.
```

Implementation Specifics

The **G32ALLOC** function is part of the AIX 3270 Host Connection Program/6000 (HCON).

The **G32ALLOC** function requires one of the following network communication adapters:

- IBM 3270 Connection Adapter and attachment cables for connection to an IBM 3174/3274 Control Unit, IBM 4361 Work Station Adapter, or an IBM 9370 Work Station Subsystem Controller configured for non–SNA distributed function terminal (non–SNA DFT) mode.
- IBM System/370 Host Interface Adapter and attachment cables for connection to an IBM 5088 Graphics Control Unit.

The **G32ALLOC** function requires one of the following IBM System/370 operating system environments be installed on the System/370: VM/SP CMS, VM/XA CMS, MVS/SP TSO/E, or MVS/XA TSO/E.

The **G32ALLOC** function is not available for Japanese Language Support.

Related Information

Additional host interface functions are the **G32DLLOC** function, **G32READ** function, and **G32WRITE** function.

AIX session control subroutines are the **g32_alloc** subroutine, **g32_close** subroutine, **g32_dealloc** subroutine, **g32_open** subroutine, and **g32_openx** subroutine.

AIX message interface subroutines are the **g32_get_status** subroutine, **g32_read** subroutine, and **g32_write** subroutine.

HCON Overview for Programming, Understanding the HCON Application Programming Interfaces, Understanding the HCON Host Interface in *Communications Programming Concepts.*

How to Compile a Host HCON API Program, Host API Errors, Sample Flows of API Programs in *Communications Programming Concepts.*

g32_alloc Function

Purpose

Initiate interaction with a host application.

Library

HCON Library

C (libg3270.a)
Pascal (libg3270p.a)
FORTRAN (libg3270f.a)

C Syntax

#include <g32_api.h>

g32_alloc (*as*, *applname*, *mode*)
struct g32_api **as*;
char **applname*;
int *mode*;

Pascal Syntax

function g32allc(*var as : g32_api;*
applname : stringptr;
mode : integer): **integer; external;**

FORTRAN Syntax

EXTERNAL G32ALLOC
INTEGER RC, *MODE*, *AS*(9), **G32ALLOC**
CHARACTER* *XX NAME*
RC = G32ALLOC (*AS, NAME, MODE*)

Description

The **g32_alloc** function initiates interaction with a host application and sets the API mode. The host application program is invoked by entering its name, using the logical terminal interface.

If invocation of the host program is successful and the mode is API/API, control of the session is passed to the AIX application. If the mode is API/3270, the emulator retains control of the session. The application communicates with the session by way of the logical terminal interface.

The **g32_alloc** function may be used only after a successful open using the **g32_open** or **g32_openx** function. The **g32_alloc** function must be issued before using any of the message or logical terminal interface functions.

HCON application programs using the Pascal language interface must include and link both the C and Pascal libraries. Applications programs using the FORTRAN language for the HCON API must include and link both the C and FORTRAN libraries.

C Parameters

as
Specifies a pointer to a **g32_api** structure. Status information is returned in this structure.

applname
Specifies a pointer to the name of the host application that is to be executed. This string should be the entire string necessary to start the application, including any necessary parameters or options. When using API/3270 mode, place the value in two double quotes (""Testload"") or specify a null string (" "). When using API/API mode, place the host application name in double quotes ("Testload")

mode
Specifies the API mode.The types of modes that can be used are contained in the **g32_api.h** file and are defined as follows:

MODE_3270
The API/3270 mode is for communicating with host applications that assume they are communicating with a 3270terminal. Applications in this mode use the logical terminal interface to communicate with the host application. In API/3270 mode, if *applname* is a null pointer, no host application is started.

MODE_API
The API/API mode is for communicating with host applications that assume they are communicating with a program. Applications in this mode use the message interface to communicate with host applications using the host API.

Note: When a session is in this mode, all activity to the screen is stopped until this mode is exited. API/3270 mode functions cannot be used while in the API/API mode.

MODE_API_T
The API_T mode is the same as **MODE_API** except this mode translates messages received from the host from EBCDIC to ASCII, and translates messages sent to the host from ASCII to EBCDIC. The translation table used is determined by the country field in the HCON session profile.

Note: A host application started in API/API or API/API_T mode must issue a **G32ALLOC** function as the API waits for an acknowledgment from the host application, when starting an API/API mode session.

Pascal Parameters

as
Specifies the **g32_api** structure.

applname
Specifies a **stringptr** containing the name of the host application to be executed. This string should be the entire string necessary to start the host application, including any necessary parameters and options. A NULL application name is valid in 3270 mode.

sessionmode
Specifies the mode desired for the session.

FORTRAN Parameters

AS Specifies the **g32_api** equivalent structure as an array of integers.

NAME Specifies the name of the application that is to execute on the host.

MODE Specifies the desired mode for the API.

Return Values

Upon successful completion:

- A value of 0 is returned.

Upon unsuccessful completion:

- A value of –1 is returned.
- The **errcode** bit is set to an error code identifying the error.
- The **xerrinfo** bit can be set to give more information about the error.

Example

C Language

1. The following example illustrates the use of the **g32_alloc** function:

```
#include <g32_api.h                /* API include file */
main ()
{
struct g32_api *as, asx;           /* asx is statically defined*/
int session_mode = MODE_API        /* api session mode.  Other modes
                                      are MODE_API_T */
char appl_name [20]                /* name of the application to
                                      run on the host*/
int return;                        /* return code */
.
.
.
strcpy (appl_name, "APITESTN"); /* name of host application*/
return = g32_alloc(as, appl_name, session_mode);
.
.
.
return = g32_dealloc(as);
.
.
.
```

Implementation Specifics

The **g32_alloc** function is part of the AIX 3270 Host Connection Program/6000 (HCON).

The **g32_alloc** function requires one of the following network communication adapters:

- IBM 3270 Connection Adapter and attachment cables for connection to an IBM 3174/3274 Control Unit, IBM 4361 Work Station Adapter, or an IBM 9370 Work Station Subsystem Controller configured for non-SNA distributed function terminal (non–SNA DFT) mode.
- IBM System/370 Host Interface Adapter and attachment cables for connection to an IBM 5088 Graphics Control Unit.

The **g32_alloc** function requires one of the following IBM System/370 operating system environments be installed on the System/370: VM/SP CMS, VM/XA CMS, MVS/SP TSO/E, or MVS/XA TSO/E.

The **g32_alloc** function is not available for Japanese Language Support.

Files

/usr/include/g32_api.h	Contains data structures and associated symbol definitions.
/usr/include/g32const.inc	Defines Pascal API constants
/usr/include/g32hfile.inc	Defines Pascal API external definitions
/usr/include/g32types.inc	Defines Pascal API data types

Related Information

Additional session control functions are the **g32_close** function, **g32_dealloc** function, **g32_open** function, and **g32_openx** function.

AIX logical terminal interface functions are the **g32_get_cursor** function, **g32_get_data** function, **g32_notify** function, **g32_search** function, and **g32_send_keys** function.

The API file transfer functions is the **g32_fxfer** function.

AIX message interface functions are the **g32_get_status** function, **g32_read** function, and **g32_write** function.

Host interface functions are the **G32ALLOC** function, **G32DLLOC** function, **G32READ** function, and **G32WRITE** function.

HCON Overview for Programming, Understanding the HCON Application Programming Interface, Understanding the AIX Interface for HCON API, API error codes, Sample Flows of API Programs in *Communications Programming Concepts*.

Understanding HCON Emulator Session Profiles in *Communication Concepts and Procedures*.

g32_close Function

Purpose

Detaches from a session.

Library

HCON Library

C (libg3270.a)
Pascal (libg3270p.a
FORTRAN (libg3270f.a)

C Syntax

#include <g32_api.h>

g32_close(*as***)**
**struct g32_api ****as*;

Pascal Syntax

function g32clse (var *as* : g32_api) **: integer; external;**

FORTRAN Syntax

EXTERNAL G32CLOSE
INTEGER *AS*(9), **G32CLOSE**

RC = G32CLOSE(*AS***)**

Description

The **g32_close** function relinquishes use of the session. If the **g32_open** or **g32_openx** created the session, the **g32_close** function will log off from the host and terminate the session. Any session must be terminated (by using the **g32_dealloc** function) before issuing the **g32_close** function.

HCON application programs using the Pascal language interface must include and link both the C and Pascal libraries. Applications programs using the FORTRAN language for the HCON API must include and link both the C and FORTRAN libraries.

C Parameter

as Specifies a pointer to a **g32_api** structure. Status is returned in this structure.

Pascal Parameter

as Specifies a **g32_api** structure.

FORTRAN Parameter

AS Specifies the **g32_api** equivalent structure as an array of integers.

Return Values

Upon successful completion:

- A value of 0 is returned.

Upon unsuccessful completion:

- A value of –1 is returned.
- The **errcode** bit is set to an error code identifying the error.
- The **xerrinfo** bit can be set to give more information about the error.

Examples

C Language

1. The following example fragment illustrates the use of the **g32_close** function:

```
#include <g32_api.h>              /* API include file */
main()
{
struct g32_api *as;               /* g32 structure */
int return;
.
.
.
return = g32_close(as);
.
.
.
```

Implementation Specifics

The **g32_close** function is part of the AIX 3270 Host Connection Program/6000 (HCON).

The **g32_close** function requires one of the following network communication adapters:

- IBM 3270 Connection Adapter and attachment cables for connection to an IBM 3174/3274 Control Unit, IBM 4361 Work Station Adapter, or an IBM 9370 Work Station Subsystem Controller configured for non–SNA distributed function terminal (non–SNA DFT) mode.
- IBM System/370 Host Interface Adapter and attachment cables for connection to an IBM 5088 Graphics Control Unit.

The **g32_close** function requires one of the following IBM System/370 operating system environments be installed on the System/370: VM/SP CMS, VM/XA CMS, MVS/SP TSO/E, or MVS/XA TSO/E.

The **g32_close** function is not available for Japanese Language Support.

Files

/usr/include/g32_api.h	Contains data structures and associated symbol definitions.
/usr/include/g32const.inc	Defines Pascal API constants
/usr/include/g32hfile.inc	Defines Pascal API external definitions
/usr/include/g32types.inc	Defines Pascal API data types

Related Information

Additional session control functions are the **g32_alloc** function, **g32_dealloc** function **g32_open** function, and **g32_openx** function.

AIX logical terminal interface functions are the **g32_get_cursor** function, **g32_get_data** function, **g32_notify** function, **g32_search** function, and **g32_send_keys** function.

The API file transfer functions is the **g32_fxfer** function.

AIX message interface functions are the **g32_get_status** function, **g32_read** function, and **g32_write** function.

Host interface functions are the **G32ALLOC** function, **G32DLLOC** function, **G32READ** function, and **G32WRITE** function.

HCON Overview for Programming, Understanding the HCON Application Programming Interfaces, Understanding the AIX Interface for HCON API, API error codes, Sample Flows of API Programs in *Communications Programming Concepts.*

g32_dealloc Function

Purpose

Ends interaction with a host application.

Library

HCON Library

C (libg3270.a)
Pascal (libg3270p.a
FORTRAN (libg3270f.a)

C Syntax

#include <g32_api.h>

g32_dealloc(*as***)**
struct g32_api **as*;

Pascal Syntax

function g32deal (var *as* : g32_api) **: integer; external;**

FORTRAN Syntax

EXTERNAL G32DEALLOC
INTEGER *AS***(9), G32DEALLOC**

RC = G32DEALLOC(*AS***)**

Description

The **g32_dealloc** function ends interaction with the AIX application and the host application. The function releases control of the session.

HCON application programs using the Pascal language interface must include and link both the C and Pascal libraries. Applications programs using the FORTRAN language for the HCON API must include and link both the C and FORTRAN libraries.

C Parameter

as Specifies a pointer to a **g32_api** structure as an array of integers.

Pascal Parameter

as Specifies the **g32_api** structure.

FORTRAN Parameters

AS Specifies the **g32_api** equivalent structure.

Return Values

Upon successful completion:

- The session is terminated.
- A value of 0 is returned.

Upon unsuccessful completion:

- A value of –1 is returned.
- The **errcode** bit is set to an error code identifying the error.
- The **xerrinfo** bit can be set to give more information about the error.

Examples

C Language

1. The following example illustrates the use of the **g32_dealloc** function:

```
#include <g32_api.h>            /* API include file */
main ()
{
struct g32_api *as, asx;        /* asx is statically defined */
int session_mode = MODE_API;    /* api session mode.  Other modes
                                   are MODE_API_T */
char appl_name [20];            /* name of the application to
                                   run on the host            */
int return;                     /* return code                */
.
.
.
strcpy (appl_name, "APITESTN"); /* name of host application  */
return = g32_alloc(as, appl_name, session_mode);
.
.
.
return = g32_dealloc(as);
.
.
.
```

Implementation Specifics

The **g32_dealloc** function is part of the AIX 3270 Host Connection Program/6000 (HCON).

The **g32_dealloc** function requires one of the following network communication adapters:

- IBM 3270 Connection Adapter and attachment cables for connection to an IBM 3174/3274 Control Unit, IBM 4361 Work Station Adapter, or an IBM 9370 Work Station Subsystem Controller configured for non–SNA distributed function terminal (non–SNA DFT) mode.
- IBM System/370 Host Interface Adapter and attachment cables for connection to an IBM 5088 Graphics Control Unit.

The **g32_dealloc** function requires one of the following IBM System/370 operating system environments be installed on the System/370: VM/SP CMS, VM/XA CMS, MVS/SP TSO/E, or MVS/XA TSO/E.

The **g32_dealloc** function is not available for Japanese Language Support.

Files

/usr/include/g32_api.h	Contains data structures and associated symbol definitions.
/usr/include/g32const.inc	Defines Pascal API constants

/usr/include/g32hfile.inc	Defines Pascal API external definitions
/usr/include/g32types.inc	Defines Pascal API data types

Related Information

Additional session control functions are the **g32_alloc** function, **g32_close** function, **g32_open** function, and **g32_openx** function.

AIX logical terminal interface functions are the **g32_get_cursor** function, **g32_get_data** function, **g32_notify** function, **g32_search** function, and **g32_send_keys** function.

The API file transfer functions is the **g32_fxfer** function.

AIX message interface functions are the **g32_get_status** function, **g32_read** function, and **g32_write** function.

Host interface functions are the **G32ALLOC** function, **G32DLLOC** function, **G32READ** function, and **G32WRITE** function.

HCON Overview for Programming, Understanding the HCON Application Programming Interfaces, Understanding the AIX Interface for HCON API, API error codes, Sample Flows of API Programs in *Communications Programming Concepts*.

G32DLLOC Function

Purpose

Terminates interaction with an AIX API application running simultaneously on the RISC System/6000.

Syntax

G32DLLOC

Description

The **G32DLLOC** function ends interaction with an AIX API application. The **G32DLLOC** function is a HCON API function that can be called by a 370 Assembler applications program.

Return Values

This call sets register 0 (zero) to the following values:

0	Zero. A normal return; call successful
< 0	Less than zero. Error condition.

Examples

The following 370 Assembler code example illustrates the use of the host **G32DLLOC** function:

```
L R11,=v(G32DATA)
USING G32DATAD,R11
G32DLLOC                /* Deallocate a session   */
C R0, G32ESESS  /* Check for G32 error    */
BE    SESSERR           /* Branch if error        */
C R0, G32ESYS           /* Check for system error */
BE    SYSERR            /* Branch if error        */
   .
   .
   .
```

Implementation Specifics

The **G32DLLOC** function is part of the AIX 3270 Host Connection Program/6000 (HCON).

The **G32DLLOC** function requires one of the following network communication adapters:

- IBM 3270 Connection Adapter and attachment cables for connection to an IBM 3174/3274 Control Unit, IBM 4361 Work Station Adapter, or an IBM 9370 Work Station Subsystem Controller configured for non–SNA distributed function terminal (non–SNA DFT) mode.
- IBM System/370 Host Interface Adapter and attachment cables for connection to an IBM 5088 Graphics Control Unit.

The **G32DLLOC** function requires one of the following IBM System/370 operating system environments be installed on the System/370: VM/SP CMS, VM/XA CMS, MVS/SP TSO/E, or MVS/XA TSO/E.

The **G32DLLOC** function is not available for Japanese Language Support.

Related Information

Additional host interface functions are the **G32ALLOC** function, **G32READ** function, and **G32WRITE** function.

AIX session control subroutines are the **g32_alloc** subroutine, **g32_close** subroutine, **g32_dealloc** subroutine, **g32_open** subroutine, and **g32_openx** subroutine.

AIX message interface subroutines are the **g32_get_status** subroutine, **g32_read** subroutine, and **g32_write** subroutine.

g32_fxfer Function

Purpose

Invokes a file transfer.

Library

HCON Library

File Transfer Library (libfxfer.a)
C (libg3270.a)
Pascal (libg3270p.a)
Fortran (libg3270f.a)

C Syntax

#include <g32_api.h>
#include <fxfer.h>

g32_fxfer(*AS,Xfer***)**
**struct g32_api ***AS*;
**struct fxc ***Xfer*;

Pascal Syntax

const
%include /usr/include/g32const.inc
%include /usr/include/g32fxconst.inc
type
%include /usr/include/g32types.inc
%include /usr/include/fxhfile.inc

function **g32fxfer**(var *AS* : **g32_api**; var *Xfer* : **fxc**) **: integer; external;**

FORTRAN Syntax

INTEGER G32FXFER, RC, *AS*(9)
EXTERNAL G32FXFER
CHARACTER*XX SRCF, DSTF*
INTEGER *FLAGS,RECL,BLKSIZE,SPACE,INCR,UNIT*

RC = **G32FXFER**(*AS,SCRF,DSTF,FLAGS,RECL,BLKSIZE,SPACE,*
+ *INCR,UNIT*)

Description

The **g32_fxfer** function allows a file transfer to take place within an API program without the API program having to invoke a **g32_close** and relinquish the link. The file transfer is run programmatically, meaning the user must set up the flag options, the source file name, and the destination file name using either the programmatic **fxfer fxc** structure for C and Pascal or the numerous variables for FORTRAN. The **g32_fxfer** function will in affect detach from the session without terminating it, run the specified file transfer and then reattach to the session.

If a **g32_alloc** has been issued before invoking the **g32_fxfer** command, be sure that the corresponding **g32_dealloc** is incorporated into the program before the **g32_fxfer** function is called.

The status of the file transfer can be checked by using the **cfxfer** file transfer status check function after the **g32_fxfer** function has been invoked.

HCON application programs using the Pascal language interface must include and link both the C and Pascal libraries. Applications programs using the FORTRAN language for the HCON API must include and link both the C and FORTRAN libraries.

C Parameters

AS Specifies a pointer to the **g32_api** structure. Status is returned in this structure.

Xfer Specifies a pointer to the **fxc** structure defined in the **fxfer.h** file.

Pascal Parameters

AS Specifies a record of type **g32_api**.

Xfer Specifies a record of type **fxc** within the **fxfer.inc** file.

FORTRAN Parameters

AS Specifies the **g32_api** equivalent structure as an array of integers.

SRCF Specifies a character array of *XX* length containing the source file name.

DSTF Specifies a character array of *XX* length containing the destination file name.

FLAGS Contains the option flags value, which is the sum of the desired option values listed below:

1	Upload
2	Download
4	Translate On
8	Translate Carriage Return Line Feed
16	Replace
32	Append
64	Queue – this option may be specified by the user, but it is blocked by the **G32FXFER** command
128	Fixed Length Records
256	Variable Length Records
512	Undefined Length (TSO only)
1024	Host System TSO
2048	Host System CMS

RECL Specifies the logical record length.

BLKSIZE Specifies the block size. (TSO only)

SPACE Specifies the allocation space. (TSO only)

INCR Specifies the allocation space increment. (TSO only)

UNIT Specifies the unit of allocation (TSO only), which is:

–1 is the number of TRACKS

–2 is the number of CYLINDERS

A **positive** number indicates the number of bytes to be allocated.

Note: All FORTRAN character array strings must be NULL–terminated (for example, SRCF = rtfile//CHAR(0)).

Return Values

Upon successful completion:

0 The user may call the **cfxfer** function to get the status of the file transfer.

Upon unsuccessful completion:

1 The file transfer did not complete successfully. The user may call the **cfxfer** function to get the status of the file transfer.

–1 The **g32_fxfer** command failed while accessing the link. The **errcode** bit is set to an error code identifying the error. The **xerrinfo** bit can be set to give more information about the error.

Examples

1. **C:**

```
#include <g32_api.h>  /* API include file */
#include <fxfer.h>    /* file transfer include file */
main()
{
  struct g32_api *as,asx;
  struct fxc *xfer;
  struct fxs sxfer;
  int session_mode=MODE_3270;
  char *aixfile="/etc/motd";
  char *hostfile="test file a";
  char sessionname[30],uid[30],pw[30];
  int mlog=0,ret=0;
  as = &asx;
  sessionname = '\0';  /* We are assuming SNAME is set */
   .
   .
 ret=g32_open(as,mlog,uid,pw,sessionname);
```

```
  printf("The g32_open return code = %d\n",ret);
   .
   .
   /* Malloc space for the file transfer structure */
  xfer = (struct fxc *) malloc(2048);
  /* Set the file transfer flags to upload,
      replace, translate and Host CMS */
  xfer->fxc_opts.f_flags = FXC_UP | FXC_REPL | FXC_TNL | FXC_CMS;
  xfer->fxc_opts.f_lrecl = 80;  /* Set the Logical Record length
                                      to 80 */
  xfer->fxc_src = aixfile;      /* Set the Source file name to
                                    aixfile */
  xfer->fxc_dst = hostfile;     /* Set the Destination file name
                                    to hostfile */
 ret=g32_fxfer(as,xfer);
  printf("The g32_fxfer return code = %d\n",ret);

  /* If the file transfer completed then get the status code of
      the file transfer */
  if ((ret == 0) || (ret == 1)) {
     ret = cfxfer(&sxfer);
     if (ret == 0) {
        printf("Source file:            %s\n",sxfer.fxs_src);
        printf("Destination file:       %s\n",sxfer.fxs_dst);
        printf("Byte Count:             %d\n",sxfer.fxs_bytcnt);
        printf("File transfer time:     %d\n",sxfer.fxs_ctime);
        printf("Status Message Number:  %d\n",sxfer.fxs_stat);
        printf("System Call error number: %d\n",sxfer.fxs_errno);
     }
  }
  .
  .
  .
  ret=g32_close(as);
  printf("The g32_close return code = %d\n",ret);
  return(0);
}
```

2. **Pascal:**

```
program test1(input,output);
const
%include /usr/include/g32const.inc
%include /usr/include/fxconst.inc
type
%include /usr/include/g32hfile.inc
%include /usr/include/g32types.inc
%include /usr/include/fxhfile.inc
var
  as:g32_api;
  xfer:fxc;
  sxfer:fxs;
  ret,sess_mode,flag:integer;
  session,timeout,uid,pw:stringptr;
  source,destination:stringptr;

begin
  sess_mode = MODE_3270;
```

```
flag := 0;
{* Initialize API stringptrs and create space *}
new(uid,8);
uid@ := chr(0);
new(pw,8);
pw@ := chr(0);
new(session,2);
session@ := 'a'; {* Open session a *}
new(timout,8);
timeout := '60';
{* Call g32openx and open session a *}
ret := g32openx(as,flag,uid,pw,session,timeout);
writeln('The g32openx return code = ',ret:4);
 .
 .
 .
{* Set up the file transfer options and file names  *}
new(source,1024);
source := 'testfile';  {*  Source file, assumes testfile exists
                           in the current directory *}
new(destination,1024);
destination := 'testfile'; {* Destination file, TSO file
                              testfile *}
{* Set flags to Upload, Replace, Translate and Host TSO  *}
xfer.fxc_opts.f_flags := FXC_UP + FXC_TSO + FXC_REPL + FXC_TNL;
xfer.fxc_src := source;
xfer.fxc_dst := destination;
{* Call the g32_fxfer using the specified flags and file names
   *}
ret := g32fxfer(as,xfer);
writeln('The g32fxfer return code = ',ret:4);
{* If g32_fxfer returned with 1 or 0 call the file transfer
   status check function *}
if (ret >= 0) then begin
   ret := pcfxfer(sxfer);
   if (ret = 0) then begin
      writeln('Source file:            ',sxfer.fxs_src@);
      writeln('Destination file:       ',sxfer.fxs_dst@);
      writeln('File Transfer Time:     ',sxfer.fxs_ctime@);
      writeln('Byte Count:             ',sxfer.fxs_bytcnt);
     writeln('Status Message Number:  ',sxfer.fxs_stat);
     writeln('System Call Error Number: ',sxfer.fxs_errno);
   end;
end;
 .
 .
 .
{* Close the session using the g32close function *}
ret := g32close(as);
writeln('The g32close return code = ',ret:4);
end.
```

3. **FORTRAN:**

```
        INTEGER G32OPENX,G32FXFER,G32CLOSE,FCFXFER
        INTEGER RET,AS(9)FLAG
        EXTERNAL G32OPENX
        EXTERNAL G32FXFER
        EXTERNAL G32CLOSE
        EXTERNAL FCFXFER
        CHARACTER*8 UID
        CHARACTER*8 PW
        CHARACTER*2 SESSION
        CHARACTER*8 TIMEOUT
        CHARACTER*256 SRCF
        CHARACTER*256 DSTF
        CHARACTER*256 SRC
        CHARACTER*256 DST
        CHARACTER*40  TIME
        INTEGER BYTCNT,STAT,ERRNO,TIME
        INTEGER FLAGS,RECL,BLKSIZE,SPACE,INCR,UNIT

C       Set up all FORMAT statement
1       FORMAT("THE G32OPENX RETURN CODE = ",I4)
2       FORMAT("THE G32FXFER RETURN CODE = ",I4)
3       FORMAT("THE G32CLOSE RETURN CODE = ",I4)
4       FORMAT("THE FCFXFER  RETURN CODE = ",I4)
5       FORMAT("————————————————————————————————————")
10      FORMAT("SOURCE FILE: ",A)
11      FORMAT("DESTINATION FILE: ",A)
12      FORMAT("BYTE COUNT: ",I10)
13      FORMAT("TIME: ",A)
14      FORMAT("STATUS MESSAGE NUMBER: ",I10)
15      FORMAT("SYSTEM CALL ERROR NUMBER: ",I10)

C       Set up all character values for the G32OPENX command
        UID = CHAR(0)
        PW = CHAR(0)
        SESSION = 'z'//CHAR(0)
        TIMEOUT = '60'//CHAR(0)
        FLAG = 0
        SRCF = 'testcase1'//CHAR(0)
        DSTF = '/u/test.case1'//CHAR(0)
C       Source and Destination files for the fcfxfer status check
command
        SRC = CHAR(0)
        DST = CHAR(0)
C       Set the G32FXFER file transfer flags and options
C       Take the defaults for Logical Record Length, Block Size,
and Space
        RECL = 0
        BLKSIZE = 0
        SPACE = 0
C       Set FLAGS to download (2), translate(4), and Host
TSO(1024)
        FLAGS = 1030
C       Call G32OPENX
        RET = G32OPENX(AS,FLAG,UID,PW,sessionname,TIMEOUT)
        WRITE(*,1) RET
        .
        .
```

```
          .
C         Call G32FXFER
          RET = G32FXFER(AS,SRCF,DSTF,FLAGS,RECL,BLKSIZE,SPACE
        + INCR,UNIT)
          WRITE(*,2) RET
          .
          .
          .
C         Call G32CLOSE
          RET = G32CLOSE(AS)
          WRITE(*,3) RET
C         Call FCFXFER for file transfer status output
          RET = FCFXFER(SRC,DST,BYTCNT,STAT,ERRNO,TIME)
          WRITE(*,4) RET
          WRITE(*,5)
          WRITE(*,10) SRC
          WRITE(*,11) DST
          WRITE(*,12) BYTCNT
          WRITE(*,13) TIME
          WRITE(*,14) STAT
          WRITE(*,15) ERRNO
          WRITE(*,5)
          STOP
          END
```

Implementation Specifics

The **g32_fxfer** function is part of the AIX 3270 Host Connection Program/6000 (HCON).

The **g32_fxfer** function requires one of the following network communication adapters:

- IBM 3270 Connection Adapter plus appropriate cables for attachment to an IBM 3174/3274 Control Unit, IBM 4361 Work Station Adapter, or an IBM 9370 Work Station Subsystem Controller configured for non–SNA distributed function terminal (non–SNA DFT) mode.
- IBM System/370 Host Interface Adapter plus appropriate cables for attachment to an IBM 5088 Graphics Control Unit.

This function requires one of the following IBM System/370 operating system environments be installed on the System/370: VM/SP CMS, VM/XA CMS, MVS/SP TSO/E, or MVS/XA TSO/E.

This function requires that the System/370 IBM Host–Supported File Transfer Program (IND$FILE) be installed on the System/370.

This function is not available for Japanese Language Support.

Files

/usr/include/fxfer.h	File transfer include file with structures and definitions for C.
/usr/include/fxconst.inc	Pascal **fxfer** function constants.
/usr/include/fxhfile.inc	Pascal file transfer invocation include file.
/usr/include/g32_api.h	Contains data structures and associated symbol definitions.

/usr/include/g32const.inc	Defines Pascal API constants
/usr/include/g32hfile.inc	Defines Pascal API external definitions
/usr/include/g32types.inc	Defines Pascal API data types

Related Information

Session control functions are the **g32_open** function, the **g32_openx** function, the **g32_close** function, the **g32_alloc** function, and the **g32_dealloc** function.

The **fxfer** function and **cfxfer** function.

HCON Overview for Programming, Understanding the HCON Application Programming Interfaces, Understanding the AIX Interface for HCON API, API error codes, Sample Flows of API Programs in *Communications Programming Concepts*.

g32_get_cursor Function

Purpose

Sets the row and column components of the **g32_api** structure to the current cursor position in a presentation space.

Library

HCON Library

C (libg3270.a)
Pascal (libg3270p.a)
FORTRAN (libg3270f.a)

C Syntax

#include <g32_api.h>

g32_get_cursor(_as_)
struct g32_api _as_

Pascal Syntax

function g32curs (var _as_ : g32_api) **: integer; external;**

FORTRAN Syntax

EXTERNAL G32GETCURSOR
INTEGER _AS_(9)**, G32GETCURSOR**

RC = G32GETCURSOR(_AS_)

Description

The **g32_get_cursor** function obtains the row and column address of the cursor and places these values in the _as_ structure. An application can only use the **g32_get_cursor** function in API/3270 mode.

HCON application programs using the Pascal language interface must include and link both the C and Pascal libraries. Applications programs using the FORTRAN language for the HCON API must include and link both the C and FORTRAN libraries.

C Parameter

_as_ Specifies a pointer to the **g32_api** structure. The row (**row**) and column (**column**) address of the cursor is set here. Status information is also set in this structure.

Pascal Parameter

_as_ Specifies the **g32_api** structure.

FORTRAN Parameter

_AS_ Specifies the **g32_api** equivalent structure as an array of integers.

Return Values

Upon successful completion:

- A value of 0 is returned.
- The corresponding **row** element of the *as* structure is the row position of the beginning of the matched string.
- The corresponding **column** element of the *as* structure is the column position of the beginning of the matched string.

Upon unsuccessful completion:

- An error code (–1 (–one)) is returned.
- The **errcode** bit is set to the error code identifying the error.
- The **xerrinfo** bit can be set to give more information about the error.

Examples

C Language

1. The following example fragment illustrates the use of the **g32_get_cursor** function in an api_3270 mode program:

   Note: The following example is missing the required **g32_open** and **g32_alloc** functions which are necessary for every HCON Workstation API program.

```
#include <g32_api.h>                /* API include file */
main()
{
struct g32_api *as;                 /* g32 structure */

char *buffer;                              /* pointer to char string
*/
int return;                                /* return code */
char *malloc();                            /* C memory allocation
                                              function */

.
.
.
return = g32_notify(as,1);          /* Turn notification on */
buffer = malloc(10);
return = g32_get_cursor(as);        /* get location of cursor */
printf (" The cursor positionis row: %d col: %d/n";
            as -> row, as -> column);
/* Get data from host starting at the current row and column */
as -> length = 10;                  /* length of a pattern on host */
return = g32_get_data(as,buffer); /* get data from host */
printf("The data returned is <%s>\n",buffer);

/* Try to search for a particular pattern on host */
as ->row =1;                        /* row to start search */
as ->column =1;                     /* column to start search */
return = g32_search(as,"PATTERN");

/*Send a clear key to the host *?
strcpy (buffer, "CLE/0");
return = g32_send_keys(as, buffer);
```

```
/* Turn notification off */
return = g32_notify(as,0);
.
.
.
```

Implementation Specifics

The **g32_get_cursor** function is part of the AIX 3270 Host Connection Program/6000 (HCON).

The **g32_get_cursor** function requires one of the following network communication adapters:

- IBM 3270 Connection Adapter and attachment cables for connection to an IBM 3174/3274 Control Unit, IBM 4361 Work Station Adapter, or an IBM 9370 Work Station Subsystem Controller configured for non–SNA distributed function terminal (non–SNA DFT) mode.
- IBM System/370 Host Interface Adapter and attachment cables for connection to an IBM 5088 Graphics Control Unit.

The **g32_get_cursor** function requires one of the following IBM System/370 operating system environments be installed on the System/370: VM/SP CMS, VM/XA CMS, MVS/SP TSO/E, or MVS/XA TSO/E.

The **g32_get_cursor** function is not available for Japanese Language Support.

Files

/usr/include/g32_api.h	Contains data structures and associated symbol definitions.
/usr/include/g32const.inc	Defines Pascal API constants
/usr/include/g32hfile.inc	Defines Pascal API external definitions
/usr/include/g32types.inc	Defines Pascal API data types

Related Information

Additional logical terminal interface functions are the **g32_get_data** function, **g32_send_keys** function, **g32_notify** function, and **g32_search** function.

AIX session control functions are the **g32_alloc** function, **g32_close** function, **g32_dealloc** function, **g32_open** function, and **g32_openx** function.

HCON Overview for Programming, Understanding the HCON Application Programming Interface, Understanding the AIX Interface for HCON API, API error codes, Sample Flows of API Programs in *Communications Programming Concepts.*

g32_get_data Function

Purpose

Obtains current specified display data from the presentation space.

Library

HCON Library

C (libg3270.a)
Pascal (libg3270p.a)
FORTRAN (libg3270f.a)

C Syntax

#include <g32_api.h>

g32_get_data(*as,buffer*)
struct g32_api **as*;
char **buffer*;

Pascal Syntax

function g32data (var *as* : g32_api;
buffer : integer) **: integer; external;**

FORTRAN Syntax

EXTERNAL G32GETDATA
INTEGER *AS*(9), **G32GETDATA**
CHARACTER **XX* *Buffer*

RC = G32GETDATA(*AS,Buffer*)

Description

The **g32_get_data** function obtains current display data from the presentation space. If the starting offset in the buffer plus the transfer length is greater than the size of the presentation space, the transfer wraps from the last buffer position to the first and the transfer continues from there until the transfer length is exhausted.

Note: The address of a packed array can be obtained by using the **addr()** system call:
Buffer : = addr (<message array name> [1 (one)])

The **g32_get_data** function can only be used in API/3270 session mode.

HCON application programs using the Pascal language interface must include and link both the C and Pascal libraries. Applications programs using the FORTRAN language for the HCON API must include and link both the C and FORTRAN libraries.

C Parameters

as Specifies a pointer to the **g32_api** structure containing the row (**row**) and column (**column**) address where the data begins, and the length (**length**) of data to return. Status information is also returned in this structure.

buffer Specifies a pointer to a buffer where the data is placed.

Pascal Parameters

as Specifies the **g32_api** structure as an array of integers.

buffer Specifies an address of a character–packed array. The array must be the same length or greater than the length field in the **g32_api** structure.

FORTRAN Parameters

AS Specifies the **g32_api** equivalent structure.

Buffer Specifies the character array that receives the retrieved data. The array must be the same length or greater than the length field in the **g32_api** structure.

Note: If the size of the buffer is smaller than *AS(LENGTH)*, a memory fault may occur.

Return Values

Upon successful completion:

- A value of 0 is returned.

Upon unsuccessful completion:

- An error code –1 is returned.
- The **errcode** bit is set to the error code identifying the error.
- The **xerrinfo** bit can be set to give more information about the error.

Examples

C Language

1. The following example fragment illustrates the use of the **g32_get_data** function in an api_3270 mode program:

   Note: The following example is missing the required **g32_open** and **g32_alloc** functions which are necessary for every HCON Workstation API program.

```
#include <g32_api.h>           /* API include file */
main()
{
struct g32_api *as;            /* g32 structure */

char *buffer;                  /* pointer to char string */
int return;                    /* return code */
char *malloc();                /* C memory allocation function */
.
.
.
return = g32_notify(as,1);     /* Turn notification on */
buffer = malloc(10);
return = g32_get_cursor(as);  /* get location of cursor */
printf (" The cursor positionis row: %d col: %d/n";
            as -> row, as -> column);
/* Get data from host starting at the current row and column */
as -> length = 10;             /* length of a pattern on host */
return = g32_get_data(as,buffer); /* get data from host */
printf("The data returned is <%s>\n",buffer);
```

```
/* Try to search for a particular pattern on host */
as ->row =1;                    /* row to start search */
as ->column =1;                 /* column to start search */
return = g32_search(as,"PATTERN");

/*Send a clear key to the host *?
strcpy (buffer, "CLE/0");
return = g32_send_keys(as, buffer);

/* Turn notification off */
return = g32_notify(as,0);
.
.
.
```

Implementation Specifics

The **g32_get_data** function is part of the AIX 3270 Host Connection Program/6000 (HCON).

The **g32_get_data** function requires one of the following network communication adapters:

- IBM 3270 Connection Adapter and attachment cables for connection to an IBM 3174/3274 Control Unit, IBM 4361 Work Station Adapter, or an IBM 9370 Work Station Subsystem Controller configured for non–SNA distributed function terminal (non–SNA DFT) mode.
- IBM System/370 Host Interface Adapter and attachment cables for connection to an IBM 5088 Graphics Control Unit.

The **g32_get_data** function requires one of the following IBM System/370 operating system environments be installed on the System/370: VM/SP CMS, VM/XA CMS, MVS/SP TSO/E, or MVS/XA TSO/E.

The **g32_get_data** function is not available for Japanese Language Support.

Files

/usr/include/g32_api.h	Contains data structures and associated symbol definitions.
/usr/include/g32const.inc	Defines Pascal API constants
/usr/include/g32hfile.inc	Defines Pascal API external definitions
/usr/include/g32types.inc	Defines Pascal API data types

Related Information

Additional Logical Terminal Interface functions are the **g32_get_cursor** function, **g32_notify** function, **g32_search** function, and **g32_send_keys** function.

AIX session control functions are the **g32_alloc** function, **g32_close** function, **g32_dealloc** function, **g32_open** function, and **g32_openx** function.

The API file transfer function is the **g32_fxfer** function.

HCON Overview for Programming, Understanding the HCON Application Programming Interface, Understanding the AIX Interface for HCON API, API error codes, Sample Flows of API Programs in *Communications Programming Concepts.*

g32_get_status Function

Purpose

Returns status information of the logical path.

Library

HCON Library

C (libg3270.a)
Pascal (libg3270p.a)
FORTRAN (libg3270f.a)

C Syntax

#include <g32_api.h>

g32_get_status(*as*)
struct g32_api **as*;

Pascal Syntax

function g32stat (var *as*: g32_api) **: integer; external;**

FORTRAN Syntax

EXTERNAL G32GETSTATUS
INTEGER *AS*(9),**G32GETSTATUS**

RC = G32GETSTATUS(*AS*)

Description

The **g32_get_status** function obtains status information about the communication path. The function is called after an AIX API application determines that an error has occurred while reading from or writing to the communication path or after a time out. The HCON session profile specifies the communication path.

Note: The **g32_get_status** function can only be used in API/API or API/API_T mode.

HCON application programs using the Pascal language interface must include and link both the C and Pascal libraries. Applications programs using the FORTRAN language for the HCON API must include and link both the C and FORTRAN libraries.

C Parameter

as Specifies a pointer to a **g32_api** structure; status is returned in this structure.

Pascal Parameter

as Specifies the **g32_api** structure.

FORTRAN Parameter

AS Specifies a **g32_api** equivalent structure as an array of integers.

Note: This function is used to determine the condition or status of the link. It should not be used to determine whether the previous I/O operation was successful or unsuccessful (the return code will provide this information).

Return Values

Upon successful completion:

- A value of 0 is returned.

The values of **errcode** are as follows:

- No error has occurred (G32_NO_ERROR, error value = 0).
- A communications check has occurred (G32_COMM_CHK, error value = –1).
- A program check has occurred within the emulator (G32_PROG_CHK, error value = –2).
- A machine check has occurred (G32_MACH_CHK, error value = –3).

If **errcode** is anything other than G32_NO_ERROR, then **xerrinfo** contains an emulator program error code.

Upon unsuccessful completion:

- An error code of –1 is returned.
- The **errcode** bit is set to the error code identifying the error.
- The **xerrinfo** bit can be set to give more information about the error.

Example

C Language

1. The following example fragment illustrates the use of the **g32_get_status** function:

```
#include <g32_api.h>                /* API include file */
main()
{
struct g32_api *as;                 /* g32 structure */
int return;

return = g32_write(as, mssg, length);
                                    /* see if unsucessful */
if (return < 0) {
        return = g32_get_status(as);
        printf("Return from g32_get_status = %d \n",return);
        printf("errcode = %d  xerrinfor = %d \n",
                as -> errcode , as -> xerrinfo
.
.
.
```

Implementation Specifics

The **g32_get_status** function is part of the AIX 3270 Host Connection Program/6000 (HCON).

The **g32_get_status** function requires one of the following network communication adapters:

- IBM 3270 Connection Adapter and attachment cables for connection to an IBM 3174/3274 Control Unit, IBM 4361 Work Station Adapter, or an IBM 9370 Work Station Subsystem Controller configured for non–SNA distributed function terminal (non–SNA DFT) mode.
- IBM System/370 Host Interface Adapter and attachment cables for connection to an IBM 5088 Graphics Control Unit.

The **g32_get_status** function requires one of the following IBM System/370 operating system environments be installed on the System/370: VM/SP CMS, VM/XA CMS, MVS/SP TSO/E, or MVS/XA TSO/E.

The **g32_get_status** function is not available for Japanese Language Support.

Files

/usr/include/g32_api.h	Contains data structures and associated symbol definitions.
/usr/include/g32const.inc	Defines Pascal API constants
/usr/include/g32hfile.inc	Defines Pascal API external definitions
/usr/include/g32types.inc	Defines Pascal API data types

Related Information

Additional message interface functions are the **g32_read** function and **g32_write** function.

AIX session control functions are the **g32_alloc** function, **g32_close** function, **g32_dealloc** function, **g32_open** function, and **g32_openx** function.

The API file transfer function is the **g32_fxfer** function.

Host interface functions are the **G32ALLOC** function, **G32DLLOC** function, **G32READ** function, and **G32WRITE** function.

HCON Overview for Programming, Understanding the HCON Application Programming Interfaces, Understanding the AIX Interface for HCON API, API error codes, Sample Flows of API Programs in *Communications Programming Concepts*.

g32_notify Function

Purpose

Turns data notification On or Off.

Library

HCON Library

C (libg3270.a)
Pascal (libg3270p.a)
FORTRAN (libg3270f.a)

C Syntax

#include <g32_api.h>

g32_notify(*as,note***)**
struct g32_api **as*;
int *note*;

Pascal Syntax

subroutine g32Note (var *as* : g32_api;
note : integer) **: integer; external;**

FORTRAN Syntax

EXTERNAL G32NOTIFY
INTEGER *AS*(9), *Note*, **G32NOTIFY**

RC = G32NOTIFY(*AS,Note***)**

Description

The **g32_notify** subroutine is used to turn notification of data arrival On and Off. The **g32_notify** subroutine may be used only by applications in API/3270 session mode.

If an application wants to know when the emulator receives data from the host, it turns notification On. This causes the emulator to send a message to the application whenever it receives data from the host. The message is sent to the IPC message queue who's file pointer is stored in the **eventf** field of the *as* data structure. The application may then use the **poll** system call to wait for data from the host. Once notified the application should clear notification messages from the IPC queue using the **msgrcv** subroutine. When the application no longer wants to be notified, it should turn notification Off with another **g32_notify** call.

HCON application programs using the Pascal language interface must include and link both the C and Pascal libraries. Applications programs using the FORTRAN language for the HCON API must include and link both the C and FORTRAN libraries.

C Parameters

as — Specifies a pointer to the **g32_api** structure. Status is returned in this structure.

note — Specifies to turn notification Off (if the *note* parameter is zero) or On (if the *note* parameter is nonzero).

Pascal Parameters

as — Specifies a **g32_api** structure.

note — Specifies an integer that signals whether to turn notification Off (if the *note* parameter is zero) or On (if the *note* parameter is nonzero).

FORTRAN Parameters

AS — Specifies a **g32_api** equivalent structure as an array of integers.

Note — Specifies to turn notification Off (if *Note* is zero) or On (if *Note* is nonzero).

Return Values

Upon successful completion:

- A value of 0 is returned.

Upon unsuccessful completion:

- An error code –1 is returned.
- The **errcode** bit is set to the error code identifying the error.
- The **xerrinfo** bit can be set to give more information about the error.

Example

C Language

1. The following example fragment illustrates the use of the **g32_notify** function in an api_3270 mode program:

   Note: The following example is missing the required **g32_open** and **g32_alloc** functions which are necessary for every HCON Workstation API program.

```
#include  <g32_api.h>          /* API include file           */
main()
{
struct  g32_api  *as;         /* g32 structure              */

char  *buffer;                /* pointer to char string     */
int  return;                    /* return code                */
char  *malloc();                /* C memory allocation function */
.
.
.
return = g32_notify(as,1);      /* Turn notification on         */
buffer = malloc(10);
return = g32_get_cursor(as);    /* get location of cursor       */
printf (" The cursor positionis row: %d col: %d/n";
            as -> row, as -> column);
/* Get data from host starting at the current row and column    */
as -> length = 10;             /* length of a pattern on host   */
return = g32_get_data(as,buffer);/* get data from host          */
printf("The data returned is <%s>\n",buffer);
```

```
/* Try to search for a particular pattern on host                   */
as ->row =1;                      /* row to start search            */
as ->column =1;                   /* column to start search         */
return = g32_search(as,"PATTERN");

strcpy (buffer, "CLE/0");
return = g32_send_keys(as, buffer);  /* Send clear key to host */


return = g32_notify(as,0);          /* Turn notification off */
.
.
.
```

Implementation Specifics

The **g32_notify** function is part of the AIX 3270 Host Connection Program/6000 (HCON).

The **g32_notify** function requires one of the following network communication adapters:

- IBM 3270 Connection Adapter and attachment cables for connection to an IBM 3174/3274 Control Unit, IBM 4361 Work Station Adapter, or an IBM 9370 Work Station Subsystem Controller configured for non–SNA distributed function terminal (non–SNA DFT) mode.
- IBM System/370 Host Interface Adapter and attachment cables for connection to an IBM 5088 Graphics Control Unit.

The **g32_notify** function requires one of the following IBM System/370 operating system environments be installed on the System/370: VM/SP CMS, VM/XA CMS, MVS/SP TSO/E, or MVS/XA TSO/E.

The **g32_notify** function is not available for Japanese Language Support.

Files

/usr/include/g32_api.h	Contains data structures and associated symbol definitions.
/usr/include/g32const.inc	Defines Pascal API constants.
/usr/include/g32hfile.inc	Defines Pascal API external definitions.
/usr/include/g32types.inc	Defines Pascal API data types.

Related Information

Additional logical terminal interface subroutines are the **g32_get_cursor** subroutine, **g32_get_data** subroutine, **g32_search** subroutine, and **g32_send_keys** subroutine.

AIX session control functions are the **g32_alloc** function, **g32_close** function, **g32_dealloc** function, **g32_open** function, and **g32_openx** function.

The API file transfer function is the **g32_fxfer** function.

HCON Overview for Programming, Understanding the HCON Application Programming Interfaces, Understanding the AIX Interface for HCON API, API error codes, Sample Flows of API Programs in *Communications Programming Concepts*.

g32_open Function

Purpose

Attaches to a session. If the session does not exist, the session is started.

Library

HCON Library

C (libg3270.a)
Pascal (libg3270p.a)
FORTRAN (libg3270f.a)

C Syntax

#include <g32_api.h>

g32_open(*as,flag,uid,pw,sessionname*)
struct g32_api **as*;
int *flag*;
char * *uid*;
char * *pw*;
char * *sessionname*;

Pascal Syntax

function g32open(*var as : g32_api; flag : integer;*
uid : stringptr;
pw : stringptr;
sessionname : stringptr;) **: integer; external;**

FORTRAN Syntax

INTEGER G32OPEN, RC, *AS*(9), *FLAG*
EXTERNAL G32OPEN
CHARACTER*XX* *UID, PW, SESSIONNAME*

RC = G32OPEN(*AS, FLAG, UID, PW, SESSIONNAME*)

Description

The **g32_open** function attaches to a session with the host. If the session does not exist, the session is started (i.e. implicit). The user is logged on to the host if request. This function is a subset of the capability provided by the **g32_openx** function. An application program must call the **g32_open** or **g32_openx** function before calling any other API function. If an API application is running implicitly an implicit logon is performed.

HCON application programs using the Pascal language interface must include and link both the C and Pascal libraries. Applications programs using the FORTRAN language for the HCON API must include and link both the C and FORTRAN libraries.

C Parameters

as	Specifies a pointer to the **g32_api** structure. Status is returned in this structure.

flag Signals whether the logon procedure should be performed. Flag values are as follows:

- If the emulator is running and the user is logged on to the host, the value of the *flag* parameter must be 0 (zero).
- If the emulator is running, the user is not logged on to the host, and the API logs on to the host, the value of the *flag* parameter must be set to 1 (one).
- If the emulator is not running and the API application executes an implicit logon/logoff procedure, the value of *flag* parameter is ignored.

uid If the **g32_open** function is to log on to the host, the *uid* parameter specifies a pointer to the logon ID string. If the logon ID is a null string, the Logon procedure prompts the user for both the logon ID and the password unless the host login ID is specified in the session profile in which case the user is prompted only for a password. The logon ID is a string consisting of the host user ID and, optionally, a list of comma–separated AUTOLOG variables, which is passed to the implicit procedure. The following is a sample list of AUTOLOG variables:

```
userid, node_id, trace, time=n,...
```

pw Specifies a pointer to the password string associated with the logon ID string. The following usage considerations apply to the *pw* parameter:

- If no password is to be specified, the user can specify a null string.
- If no value is provided and the program is running implicitly, the logon procedure prompts the user for the password.
- if the *uid* parameter is a null string, the *pw* parameter is ignored.

sessionname Specifies a pointer to the name of a session. The session name is a single character in the range of a–z. Capital letters are interpreted as lowercase letters.

Pascal Parameters

as Specifies the **g32_api** structure.

flag Signals whether the logon procedure should be performed.

- If the emulator is running, the user is logged on to host, and the API application executes as a subshell of the emulator, the value of the *flag* parameter must be 0 (zero).
- If the emulator is running, the user is not logged on to host, and the API application executes as a subshell of the emulator and the application is to perform an implicit logon/logoff procedure, the value of the *flag* parameter must be set to 1 (one).
- If the emulator is not running and the API application executes an implicit logon/logoff procedure, the value of *flag* parameter is ignored.

uid Specifies a pointer to the logon ID string. If the user ID is a null string, the Logon procedure prompts the user for both the user ID and the password unless the host login ID is specified in the session profile. In the latter case, the user is prompted only for a password.

pw Specifies a pointer to the password string associated with the logon ID string. If it points to a null string, the Logon procedure prompts the user for the password. This parameter is ignored if the *uid* parameter is a null string.

sessionname Specifies a pointer to the name of a session, which indicates the host connectivity to be used by the API application. The session name is a single character in the range of a–z. Capital letters are interpreted as lowercase letters.

FORTRAN Parameters

When creating strings in FORTRAN that are to be passed as parameters, the strings must be terminated by with a null character CHAR(0).

AS Specifies the **g32_api** equivalent structure as an array of integers.

FLAG Signals whether the logon procedure should be performed.

UID Specifies a pointer to the logon ID string. If the user ID is a null string, the Logon procedure prompts the user for both the user ID and the password unless the host login ID is specified in the session profile. In the latter case, the user is prompted only for a password.

PW Specifies a pointer to the password string associated with the logon ID string. If the parameter specifies a null string, the Logon procedure prompts the user for the password. This parameter is ignored if the *uid* parameter is a null string.

SESSIONNAME
Specifies the name of a session, which indicates the host connectivity to be used by the API application. The session name is a single character in the range of a–z. Capital letters are interpreted as lowercase letters.

Return Values

Upon successful completion:

- A value of 0 is returned
- The **lpid** bit is set to the session ID.

Upon unsuccessful completion:

- A value of –1 is returned.
- The **errcode** bit is set to an error code identifying the error.
- The **xerrinfo** bit can be set to give more information about the error.

Examples

1. **C**:

```
#include <g32_api.h>
main()
{
    struct g32_api *as, asx;    /* asx is statically declared */
    int flag=0;
    int ret;
    char uid[30],pw[30];
    char *sn;
    char nm='a';
    int log=0;
    as = &asx;          /* as points to an allocated structure */
    sn = &nm;

    ret=g32_open(as,log,uid,pw,sn);
    .
    .
    .
}
```

2. **Pascal**:

```
program apitest (input, output);
const
%include /usr/include/g32const.inc
type
%include /usr/include/g32types.inc
var
    as : g32_api;
    rc : integer;
    flag : integer;
    sn : stringptr;
    ret : integer;
    uid, pw : stringptr;
%include /usr/include/g32hfile.inc
begin
    flag := 0;
    new(uid,20);
    uid@ := chr(0);
    new (pw,20);
    pw@ := chr(0);
    new (sn,1);
    sn@ := 'a';
    ret := g32open(as,flag,uid,pw,sn);
    .
    .
    .
end.
```

3. **FORTRAN**:

```
INTEGER G32OPEN
        INTEGER RC, AS(9), FLAG
        CHARACTER*20 UID
        CHARACTER*10 PW
        CHARACTER*1 SN
        EXTERNAL G32OPEN
        UID = CHAR(0)
        PW = CHAR(0)
        SN = 'a'//CHAR(0)
        FLAG = 0
        RC = G32OPEN(AS, FLAG, UID, PW, SN)
                .
                .
                .
```

Implementation Specifics

The **g32_open** function is part of the AIX 3270 Host Connection Program/6000 (HCON).

The **g32_open** function requires one of the following network communication adapters:

- IBM 3270 Connection Adapter and attachment cables for connection to an IBM 3174/3274 Control Unit, IBM 4361 Work Station Adapter, or an IBM 9370 Work Station Subsystem Controller configured for non–SNA distributed function terminal (non–SNA DFT) mode.
- IBM System/370 Host Interface Adapter and attachment cables for connection to an IBM 5088 Graphics Control Unit.

The **g32_open** function requires one of the following IBM System/370 operating system environments be installed on the System/370: VM/SP CMS, VM/XA CMS, MVS/SP TSO/E, or MVS/XA TSO/E.

The **g32_open** function does not feature Japanese Language Support.

Files

/usr/include/g32_api.h	Contains data structures and associated symbol definitions.
/usr/include/g32const.inc	Defines Pascal API constants
/usr/include/g32hfile.inc	Contains Pascal API external definitions
/usr/include/g32types.inc	Defines Pascal API data types

Related Information

Additional session control functions are the **g32_alloc** function, **g32_close** function, **g32_dealloc** function, and **g32_openx** function.

Additional logical terminal interface functions are the **g32_get_cursor** function, **g32_get_data** function, **g32_notify** function, **g32_search** function, and **g32_send_keys** function.

AIX message interface functions are the **g32_get_status** function, **g32_read** function, and **g32_write** function.

The API file transfer function is the **g32_fxfer** function.

Host interface functions are the **G32ALLOC** function, **G32DLLOC** function, **G32READ** function, and **G32WRITE** function.

HCON Overview for Programming, Understanding the HCON Application Programming Interfaces, Understanding the AIX Interface for HCON API, API error codes, Sample Flows of API Programs in *Communications Programming Concepts*.

g32_openx Function

Purpose

Attaches to a session and provides extended open capabilities. If the session does not exist, the session is started.

Library

HCON Library

C (libg3270.a)
Pascal (libg3270p.a)
FORTRAN (libg3270f.a)

C Syntax

#include <g32_api.h>

g32_openx(*as, flag, uid, pw, sessionname, timeout*)

struct g32_api **as*;
int *flag*;
char * *uid*;
char * *pw*;
char * *sessionname*;
char * *timeout*;

Pascal Syntax

function g32openx(var *as* : g32_api; *flag*: integer;

uid : stringptr;

pw : stringptr;

sessionname : stringptr;

timeout : stringptr) **: integer; external;**

FORTRAN Syntax

INTEGER G32OPENX,RC,*AS***(9),***FLAG*

EXTERNAL G32OPEN

CHARACTER* *XX UID, PW, SESSIONNAME*

RC = G32OPEN (*AS, FLAG, UID, PW, SESSIONNAME, TIMEOUT*)

Description

The **g32_openx** function attaches to a session. If the session does not exist, the session is started. This is an implicit logon. The user is logged on to the host if requested. The **g32_openx** function provides additional capability beyond that of the **g32_open** function. An application program must call **g32_openx** or **g32_open** before any other API function.

If an API application is run implicitly, the function performs an implicit logon is performed.

HCON application programs using the Pascal language interface must include and link both the C and Pascal libraries. Applications programs using the FORTRAN language for the HCON API must include and link both the C and FORTRAN libraries.

C Parameters

The **g32_openx** function allows for a varying number of parameters after the *'flag* parameter. This function uses two required parameters: *as* and *flag* plus the optional parameters: *uid, pw, session,* and *timeout.*

With the **g32_open** function, the *timeout* parameter does not exist and the parameters for *uid, pw,* and *session* are not optional. The reason for making the last four parameters optional is that the system either prompts for the needed information (*uid* and *pw*) or defaults with valid information (*session* or *timeout*).

Unless all of the parameters are defined for this function, the parameter list in the calling statement must be terminated with the integer 0 (zero) (like the *exec* function). Providing an integer of 1 forces a default on an parameter. Use the default to provide a placeholder for optional parameters that you do not need to supply.

as — Specifies a pointer to the **g32_api** structure.

flag — Requires one of the following:

- Set the *flag* parameter to 0 (zero), if the emulator is running and the user is logged on to host.
- Set the *flag* parameter to 1 (one) if the emulator is running, the user is not logged on to host, and the API application is to perform the logon/logoff procedure.

The **g32_open** function ignores the *flag* parameter, if the emulator is not running and the API application executes an implicit logon/logoff procedure.

uid — Specifies a pointer to the logon ID string. If the logon ID is a null string, the Logon procedure prompts the user for both the logon ID and the password, unless the host login ID is specified in the session profile. In the latter case the user is prompted only for a password. The logon ID is a string consisting of the host user ID and, optionally, a list of additional variables separated by session, as shown in the example:

```
userid,var1,var2,...
```

In this example, *var1* is the logon script name (when using AUTOLOG) and *var2* is the optional trace and time values. The list is passed to the implicit procedure.

pw — Specifies a pointer to the password string associated with the logon ID string. The following usage considerations apply to the *pw* parameter:

- If no password is to be specified, the user can specify a null string.
- If no value is provided and the program is running implicitly, the logon procedure prompts the user for the password.
- If the *uid* parameter is a null string, the *pw* parameter is ignored.

sessionname — Points to the name of a session. The session name is a single character in the range of a–z. Capital letters are interpreted as lowercase letters. Parameters for each session are specified in a per–session profile.

timeout Specifies a pointer to a numerical string (such as 30 or 60) that specifies the amount of nonactive time (in seconds) allowed to occur between the workstation and the host operations (that is, **g32_read/G32WRITE**). This parameter is optional. If no value is provided in the calling statement, the default value is 15 seconds. The minimum value allowed is 1. There is no maximum value limitation.

Pascal Parameters

When using C as a programming language, you can make use of the feature of variable numbered parameters. In Pascal, however, this feature is not allowed. Therefore, calls to the **g32_openx** function must contain all six parameters.

To use defaults for the four optional parameters of C, provide a variable whose value is a null string.

Note: The use of the integer one (1) is not allowed in the Pascal version of the **g32_openx** function. Space must be allocated for any string pointers prior to calling the **g32_openx** function.

as Specifies the **g32_api** structure.

flag Signals whether the logon procedure should be performed.

- Set the *flag* parameter to 0 (zero), if the emulator is running, the user is logged on to host.
- Set the *flag* parameter to 1 (one), if the emulator is running, the user is not logged on to host, and the API application performs the logon/logoff procedure.
- If the emulator is not running and the API application executes an implicit logon/logoff procedure, the value of *flag* is ignored.

uid Specifies a pointer to the logon ID string. If the logon ID is a null string, the logon procedure prompts the user for both the logon ID and the password, unless the host login ID is specified in the session profile. In the latter case the user is prompted only for a password.

pw Specifies a pointer to the password string associated with the logon ID string. The following usage considerations apply to the *pw* parameter:

- If no password is to be specified, the user can specify a null string.
- If no value is provided and the program is running implicitly, the logon procedure prompts the user for the password.
- If the *uid* parameter is a null string, the *pw* parameter is ignored.

sessionname Points to the name of a session. The session name is a single character in the range of a–z. Capital letters are interpreted as lowercase letters. Parameters for each session are specified in a per session profile.

timeout Specifies a pointer to a numerical string (such as 30 or 60) that specifies the amount of nonactive time (in seconds) allowed to occur between the workstation and the host operations (that is, **g32_read/g32WRITE**). This parameter is optional. If no value is provided in the calling statement, the default value is 15 seconds. The minimum value allowed is one. There is no maximum value limitation.

FORTRAN Parameters

FORTRAN calls to **G32_OPENX** *must* contain all six parameters. To use defaults for the four optional parameters of C language, provide a variable whose value is a null string. Note that the use of the integer 1 (one) is not allowed in the FORTRAN version of this function. When creating strings in FORTRAN that are to pass as parameters, the strings must be linked with a null character, CHAR (0).

AS Specifies the **g32_api** equivalent structure as an array of integers.

FLAG Signals that the logon procedure should be performed.

- Set the *Flag* parameter to 0 (zero), if the emulator is running, the user is logged on to host.
- Set the *Flag* parameter to 1 (one), if the emulator is running, the user is not logged on to host.
- If the emulator is not running and the API application executes an implicit logon/logoff procedure, the value of *Flag* is ignored.

UID Specifies a pointer to the logon ID string. If the logon ID is a null string, the logon procedure prompts the user for both the logon ID and the password, unless the host login ID is specified in the session profile. In the latter case the user is prompted only for a password.

PW Specifies a pointer to the password string associated with the logon ID string. The following usage considerations apply to the *pw* parameter:

- If no password is to be specified, the user can specify a null string.
- If no value is provided and the program is running implicitly, the logon procedure prompts the user for the password.
- If the *uid* parameter is a null string, the *pw* parameter is ignored.

SESSIONNAME
Specifies the name of a session. The session name is a single character in the range of a–z. Capital letters are interpreted as lowercase letters. Parameters for each session are specified in a per session profile.

TIMEOUT Specifies a numerical string (such as 30 or 60) that specifies the amount of nonactive time (in seconds) allowed to occur between the workstation and the host operations (that is, **g32_read/g32WRITE**). There is no maximum to this, but the minimum is 1 (one).

Return Values

Upon successful completion:

- A value of 0 is returned.
- The **lpid** bit is set to the session ID.

Upon unsuccessful completion:

- A value of –1 is returned.
- The **errcode** bit is set to an error code identifying the error.
- The **xerrinfo** bit can be set to give more information about the error.

Examples

Examples of ways to use the **g32_openx** function are as follows:

1. With fewer than four optional string constant parameters specified and used with AUTOLOG:

   ```
   g32_openx (AS, 0, "john, tso, trace", "j12hn");
   ```

2. With fewer than four optional string constant parameters specified and used with LAF:

   ```
   g32_openx (AS, 1, "john", "j12hn", "Z", 0);
   ```

3. With all optional parameters not specified:

   ```
   g32_openx (AS, 1, 0);
   or
   g32_openx (AS, 0, 0);
   ```

4. With four variable optional parameters:

   ```
   g32_openx (AS, 0, UID, Pw, Sessionname, TimeOut);
   ```

5. With fewer than four variable optional parameters:

   ```
   g32_openx (AS, 1, UID, Pw, 0);
   ```

6. With two default optional parameters:

   ```
   g32_openx (AS, 0, 1, 1, 1, "60");
   ```

7. With a mixture:

   ```
   g32_openx (AS, 0, 1, 1, Session, 0);
   ```

The following examples illustrate the use of the **g32_openx** function within a program segment in the C, Pascal, and FORTRAN languages:

1. **C**:

   ```
   #include <g32_api.h>
   main()
   {
       struct g32_api *as, asx;    /* asx is a temporary struct */
                                   /* g32.api so that storage */
                                   /* is allocated */
       int flag=0;
       int ret;
       char uid[30],pw[30];
       char *sn;
       char nm='a';
       char timeout="60";
       int log=0;

       sn = &nm;
       as = &asx;          /* as points to an allocated structure */
       ret=g32_openx(as,flag,uid,pw,sn,timeout);
       .
       .
       .
   }
   ```

2. **Pascal**:

```
program apitest (input, output);
const
%include /usr/include/g32const.inc
type
%include /usr/include/g32types.inc
var
    as : g32_api;
    rc : integer;
    flag : integer;
    sn : stringptr;
    timeout : stringptr;
    ret : integer;
    uid, pw : stringptr;
%include /usr/include/g32hfile.inc
begin
    flag := 0;
    new(uid,20);
    uid@ := chr(0);
    new (pw,20);
    pw@ := chr(0);
    new (sn,1);
    sn@ := 'a';
    new (timeout,32);
    timeout@ := '60';
    ret := g32openx(as,flag,uid,pw,sn,timeout);
    .
    .
    .
end.
```

3. **FORTRAN**:

```
INTEGER G32OPENX
INTEGER RC, AS(9), FLAG
CHARACTER*20 UID
CHARACTER*10 PW
CHARACTER*10 TIMEOUT
CHARACTER*1 SN
EXTERNAL G32OPENX
UID = CHAR(0)
TIMEOUT = CHAR(0)
MODEL = CHAR(0)
PW = CHAR(0)
SN = 'a'//CHAR(0)
TIMEOUT = '60'//CHAR(0)
FLAG = 0
RC = G32OPENX(AS, FLAG, UID, PW, SN, TIMEOUT)
    .
    .
    .
```

Implementation Specifics

The **g32_openx** function is part of the AIX 3270 Host Connection Program/6000 (HCON).

The **g32_openx** function requires one of the following network communication adapters:

- IBM 3270 Connection Adapter and attachment cables for connection to an IBM 3174/3274 Control Unit, IBM 4361 Work Station Adapter, or an IBM 9370 Work Station Subsystem Controller configured for non–SNA distributed function terminal (non–SNA DFT) mode.
- IBM System/370 Host Interface Adapter and attachment cables for connection to an IBM 5088 Graphics Control Unit.

The **g32_openx** function requires one of the following IBM System/370 operating system environments be installed on the System/370: VM/SP CMS, VM/XA CMS, MVS/SP TSO/E, or MVS/XA TSO/E.

The **g32_openx** function is not available for Japanese Language Support.

Files

/usr/include/g32_api.h	Contains data structures and associated symbol definitions.
/usr/include/g32const.inc	Defines Pascal API constants
/usr/include/g32hfile.inc	Defines Pascal API external definitions
/usr/include/g32types.inc	Defines Pascal API data types

Related Information

Additional session control functions are the **g32_alloc** function, **g32_close** function, **g32_dealloc** function, and **g32_open** function.

Additional logical terminal interface functions are the **g32_get_cursor** function, **g32_get_data** function, **g32_search** function, **g32_notify** function, and **g32_send_keys** function.

AIX message interface functions are the **g32_get_status** function, **g32_read** function, and **g32_write** function.

The API file transfer functions is the **g32_fxfer** function.

Host interface functions are the **G32ALLOC** function, **G32DLLOC** function, **G32READ** function, and **G32WRITE** function.

HCON Overview for Programming, Understanding the HCON Application Programming Interfaces, Understanding the AIX Interface for HCON API, API error codes, Sample Flows of API Programs in *Communications Programming Concepts*.

G32READ Function

Purpose

Receives a message from the AIX API application running simultaneously on the RISC System/6000.

Syntax

G32READ

Description

The **G32READ** function receives a message from an AIX API application. The **G32READ** function returns when a message is received. The status of the transmission is returned in register zero (R0).

The **G32READ** function returns the following values:

R0 Is the number of bytes read.

R1 Is the address of the message buffer.

Return Values

The **G32READ** function sets register zero (R0) to the following values:

>= 0 Normal return. This is the length of the message (the number of bytes read).

< 0 Less than zero. Host API error condition.

In VM/CMS, storage for the read command is obtained using the DMSFREE macro. R0 contains the number of bytes read. R1 contains the address of the buffer. It is the responsibility of the host application to release the buffer with a DMSFRET call. Assuming the byte count and address are in R0 and R1, respectively, the following code fragment should be used to free the buffer:

```
SRL  R0,3
A    R0,=F'1'
DMSFRET DWORDS=(0),LOC=(1)
```

In MVS/TSO, storage for the READ command is obtained using the GETMAIN macro. R0 contains the number of bytes read. R1 contains the address of the buffer. The host application must release the buffer with a FREEMAIN call.

In MVS/TSO, when programming an API assembly language application, you must be careful with the TPUT macro. If it is used in a sequence of G32READ and G32WRITE subroutines, it will interrupt the API/API mode and switch the host to API/3270 mode to exist. You will not be able to get the API/API mode back until you send the Enter key.

Example

The following 370 Assembler code example illustrates the use of the host **G32READ** function:

```
                .
                .
                .
MEMORY   L   12,=v(G32DATA)        /* SET POINTER TO API DATA AREA */
         .
         .
         .
         L  2,=F'2'
         G32READ                   /* RECEIVE MESSAGE FROM AIX */
         ST 1,ADDR                 /* STORE ADDRESS OF MESSAGE */
         ST 0,LEN                  /* STORE LENGTH OF MESSAGE */
         BAL 14,CHECK
                .
                .
                .
```

Implementation Specifics

The **G32READ** function is part of the AIX 3270 Host Connection Program/6000 (HCON).

The **G32READ** function requires one of the following network communication adapters:

- IBM 3270 Connection Adapter and attachment cables for connection to an IBM 3174/3274 Control Unit, IBM 4361 Work Station Adapter, or an IBM 9370 Work Station Subsystem Controller configured for non–SNA distributed function terminal (non–SNA DFT) mode.
- IBM System/370 Host Interface Adapter and attachment cables for connection to an IBM 5088 Graphics Control Unit.

The **G32READ** function requires one of the following IBM System/370 operating system environments be installed on the System/370: VM/SP CMS, VM/XA CMS, MVS/SP TSO/E, or MVS/XA TSO/E.

The **G32READ** function is not available for Japanese Language Support.

Related Information

Additional host interface functions are the **G32ALLOC** function, **G32DLLOC** function, and **G32WRITE** function.

AIX session control subroutines are the **g32_alloc** subroutine, **g32_close** subroutine, **g32_dealloc** subroutine, **g32_open** subroutine, and **g32_openx** subroutine.

AIX message interface subroutines are the **g32_get_status** subroutine, **g32_read** subroutine, and **g32_write** subroutine.

For documentation on the DMSFREE and DMSFRET macros, consult the *VM/SP System Programmer's Guide.*

For documentation on the GETMAIN and FREEMAIN macros, consult the *MVS/XA System Macros and Facilities, Volume 2* or *MVS/XA Supervisor Services and Macro Instructions.*

HCON Overview for Programming, Understanding the HCON Application Programming Interfaces, Understanding the HCON Host Interface in *Communications Programming Concepts*.

How to Compile a Host HCON API Program, Host API Errors, Sample Flows of API Programs in *Communications Programming Concepts*.

g32_read Function

Purpose

Receives a message from a host application.

Library

HCON Library

C (libg3270.a)
Pascal (libg3270p.a)
FORTRAN (libg3270f.a)

C Syntax

#include <g32_api.h>

g32_read (*as*, *msgbuf*, *msglen*)
struct g32_api **as*;
char ***msgbuf*;
int **msglen*;

Pascal Syntax

function g32read (*var as : g32_api;*
var Buffer : stringptr;
var msglen : integer) : **integer; external;**

FORTRAN Syntax

EXTERNAL G32READ
INTEGER *AS*(9), *BUFLEN*, **G32READ**
INTEGER *AS*(9), *BUFLEN*, **G32READ**
CHARACTER *XX *MSGBUF*

RC= G32READ (*AS*, *MSGBUF*, *BUFLEN*)

Description

The **g32_read** function receives a message from a host application. The **g32_read** function may only be used by those applications having API/API or API/API_T mode specified with the **g32_alloc** function.

- In C or Pascal, a buffer is obtained, a pointer to the buffer is saved, and the message from the host is read into the buffer. The length of the message and the address of the buffer are returned to the user application.
- In FORTRAN, the calling procedure must pass a buffer large enough for the incoming message. The *BUFLEN* parameter must be the actual size of the buffer. The **G32READ** function uses the *BUFLEN* parameter as the upper array bound. Therefore, any messages larger than *BUFLEN* are truncated to fit the buffer.

HCON application programs using the Pascal language interface must include and link both the C and Pascal libraries. Applications programs using the FORTRAN language for the HCON API must include and link both the C and FORTRAN libraries.

C Parameters

as — Specifies a pointer to a **g32_api** structure.

msgbuf — Specifies a pointer to a pointer to a buffer where a message from the host is placed. The API obtains space for this buffer by using the AIX **malloc** library subroutine, and the user is responsible for releasing it by issuing a **free** call after the **g32_read** function.

msglen — Specifies a pointer to an integer where the length, in bytes, of the *msgbuf* parameter is placed. The message length must be greater than 0 (zero) but less than or equal to the maximum I/O buffer size parameter specified in the HCON session profile.

Pascal Parameters

as — Specifies the **g32_api** structure.

Buffer — Specifies a **stringptr**. The API obtains space for this buffer by using the AIX **malloc** C library subroutine, and the user is responsible for releasing it by issuing a **dispose** subroutine after the **g32_read** function.

msglen — Specifies an integer where the number of bytes read is placed. The message length must be greater than 0 (zero) but less than or equal to the maximum I/O buffer size parameter specified in the HCON session profile.

FORTRAN Parameters

AS — Specifies the **g32_api** equivalent structure.

MSGBUF — Specifies the storage area for the character data read from the host.

BUFLEN — Specifies the size, in bytes, of the value contained in the *MSGBUF* parameter. The message length must be greater than 0 (zero) and less than the maximum I/O buffer size parameter specified in the HCON session profile.

Return Values

Upon successful completion:

- The number of bytes read is returned (≥ 0).

Upon unsuccessful completion:

- An error code –1 is returned.
- The **errcode** bit is set to the error code identifying the error.
- The **xerrinfo** bit can be set to give more information about the error.

Example

C Language

1. The following example illustrates the use of the **g32read** function:

```
#include <g32_api>          /* API include file */
main()
{
struct g32_api *as;         /* g32_api structure */
```

```
char **msg_buf;          /* pointer to host msg buffer */
char *messg;             /* pointer to character string */
int *msg_len;            /* pointer to host msg length */

char * malloc();         /* C memory allocation function */
int return;              /* return code is no. of bytes read */
.
.
.
messg = malloc(30);            /* allocate 30 bytes */
msg_buff = &messg;             /* point to a string */
msg_len = malloc(sizeof(int)); /* allocate storage */

return = g32_read(as, msg_buff, msg_len);
.
.
.
```

Implementation Specifics

The **g32_read** function is part of the AIX 3270 Host Connection Program/6000 (HCON).

The **g32_read** function requires one of the following network communication adapters:

- IBM 3270 Connection Adapter and attachment cables for connection to an IBM 3174/3274 Control Unit, IBM 4361 Work Station Adapter, or an IBM 9370 Work Station Subsystem Controller configured for non–SNA distributed function terminal (non–SNA DFT) mode.
- IBM System/370 Host Interface Adapter and attachment cables for connection to an IBM 5088 Graphics Control Unit.

The **g32_read** function requires one of the following IBM System/370 operating system environments be installed on the System/370: VM/SP CMS, VM/XA CMS, MVS/SP TSO/E, or MVS/XA TSO/E.

The **g32_read** function is not available for Japanese Language Support.

Files

/usr/include/g32_api.h	Contains data structures and associated symbol definitions.
/usr/include/g32const.inc	Defines Pascal API constants
/usr/include/g32hfile.inc	Defines Pascal API external definitions
/usr/include/g32types.inc	Defines Pascal API data types

Related Information

Additional message interface functions are the **g32_get_status** function and **g32_write** function.

AIX session control functions are the **g32_alloc** function, **g32_close** function, **g32_dealloc** function, **g32_open** function, and **g32_openx** function.

The API file transfer function is the **g32_fxfer** function.

Host interface functions are the **G32ALLOC** function, **G32DLLOC** function, **G32READ** function, and **G32WRITE** function.

The **malloc** subroutine and **free** subroutine.

HCON Overview for Programming, Understanding the HCON Application Programming Interface, Understanding the AIX Interface for HCON API, API error codes, Sample Flows of API Programs in *Communications Programming Concepts.*

g32_search Function

Purpose

Searches for a character pattern in a presentation space.

Library

HCON Library

C (libg3270.a)

Pascal (libg3270p.a)

FORTRAN (libg3270f.a)

C Syntax

#include <g32_api.h>

g32_search(*as,pattern*)
struct g32_api **as*;
char **pattern*;

Pascal Syntax

function g32srch(var *as* : g32_api;
pattern : stringptr) **: integer; external;**

FORTRAN Syntax

EXTERNAL G32SEARCH
INTEGER *AS*(9), **G32SEARCH**
CHARACTER **XX* *PATTERN*

RC = G32SEARCH(*AS,PATTERN*)

Description

The **g32_search** function searches for the specified byte pattern in the presentation space associated with the application.

Note: The **g32_search** function can only be used in API/3270 mode.

The search is performed from the row and column given in the **g32_api** structure to the end of the presentation space. Note that the row and column positions start at 1 (one) and not 0 (zero). If you start at 0 for row and column, you get invalid position errors.

In any given search pattern, the following characters have special meaning:

?	The Question mark is the arbitrary character, matching any one character.
*	The Asterisk is the wildcard character, matching any sequence of zero or more characters.
\	The Backslash is the escape character meaning the next character is to be interpreted literally.

The following rules apply to the use of wildcard characters:

- The pattern can not begin with the wildcard character.
- The pattern can not end with the wildcard character.
- The pattern can not contain two consecutive wildcard characters.

Pattern Matching Example

The string AB?DE matches any of ABCDE, AB9DE, ABxDE, but does not match ABCD, ABCCDE, or ABDE.

The string AB*DE matches any of ABCDE, AB9DE, ABCCDE, ABDE, but does not match ABCD, ABCDF, or ABC.

Pattern Matching in C and Pascal:

If the pattern needs to contain either a question mark or an asterisk as a literal character, these symbols must be preceded by two escape characters (\\? or *). For example, to search for the string, How are you today?, the pattern might be:

```
How are you today \\?
```

The backslash can be used as a literal character by specifying four backslash characters (\\\\) in the pattern. For example, to search for the string, We found the \., the pattern might be:

```
We found the \\\\.
```

Pattern Matching in FORTRAN:

If the pattern needs to contain either a question mark or an asterisk as a literal character, these symbols must be preceded by one escape character (\? or *). For example, to search for the string, How are you today?, the pattern might be:

```
How are you today\?
```

The backslash can be used as a literal character by specifying two backslash characters (\\) in the pattern. For example, to search for the string, We found the \., the pattern might be:

```
We found the \\.
```

HCON application programs using the Pascal language interface must include and link both the C and Pascal libraries. Applications programs using the FORTRAN language for the HCON API must include and link both the C and FORTRAN libraries.

C Parameters

as — Specifies a pointer to a **g32_api** structure. It also contains the row and column where the search should begin. Status information is returned in this structure.

pattern — Specifies a pointer to a byte pattern, which is searched for in the presentation space.

Pascal Parameters

as — Specifies the **g32_api** structure.

pattern Specifies pointer to a string containing the pattern to search for in the presentation space. The string must be at least as long as the length indicated in the **g32_api** structure.

FORTRAN Parameters

AS Specifies a **g32_api** equivalent structure as an array of integers.

Pattern Specifies string that is searched for in the presentation space.

Return Values

Upon successful completion:

- A value of 0 is returned
- The corresponding **row** element of the *as* structure is the row position of the beginning of the matched string.
- The corresponding **column** element of the *as* structure is the column position of the beginning of the matched string.
- The corresponding **length** element of the *as* structure is the length of the matched string.

Upon unsuccessful completion:

- An error code –1 is returned.
- The **errcode** bit is set to the error code identifying the error.
- The **xerrinfo** bit can be set to give more information about the error.

Example

C Language

1. The following example fragment illustrates the use of the **g32_search** function in an api_3270 mode program:

   Note: The following example is missing the required **g32_open** and **g32_alloc** functions which are necessary for every HCON Workstation API program.

```
#include <g32_api.h>            /* API include file */
main()
{
struct g32_api *as;             /* g32 structure */
```

```
char *buffer;                     /* pointer to char string */
int return;                       /* return code */
char *malloc();                   /* C memory allocation function */
.
.
.
return = g32_notify(as,1);        /* Turn notification on */
buffer = malloc(10);
return = g32_get_cursor(as);  /* get location of cursor */
printf (" The cursor positionis row: %d col: %d/n";
            as —> row, as —> column);
/* Get data from host starting at the current row and column */
as —> length = 10;                /* length of a pattern on host */
return = g32_get_data(as,buffer); /* get data from host */
printf("The data returned is <%s>\n",buffer);

/* Try to search for a particular pattern on host */
as —>row =1;                      /* row to start search */
as —>column =1;                   /* column to start search */
return = g32_search(as,"PATTERN");

/*Send a clear key to the host *?
strcpy (buffer, "CLE/0");
return = g32_send_keys(as, buffer);

/* Turn notification off */
return = g32_notify(as,0);
.
.
.
```

Implementation Specifics

The **g32_search** function is part of the AIX 3270 Host Connection Program/6000 (HCON).

The **g32_search** function requires one of the following network communication adapters:

- IBM 3270 Connection Adapter and attachment cables for connection to an IBM 3174/3274 Control Unit, IBM 4361 Work Station Adapter, or an IBM 9370 Work Station Subsystem Controller configured for non–SNA distributed function terminal (non–SNA DFT) mode.
- IBM System/370 Host Interface Adapter and attachment cables for connection to an IBM 5088 Graphics Control Unit.

The **g32_search** function requires one of the following IBM System/370 operating system environments be installed on the System/370: VM/SP CMS, VM/XA CMS, MVS/SP TSO/E, or MVS/XA TSO/E.

The **g32_search** function is not available for Japanese Language Support.

Files

/usr/include/g32_api.h	Contains data structures and associated symbol definitions.
/usr/include/g32const.inc	Defines Pascal API constants
/usr/include/g32hfile.inc	Defines Pascal API external definitions
/usr/include/g32types.inc	Defines Pascal API data types

Related Information

Additional Logical Terminal Interface functions are the **g32_get_cursor** function, **g32_get_data** function, **g32_notify** function, and **g32_send_keys** function.

AIX session control functions are the **g32_alloc** function, **g32_close** function, **g32_dealloc** function, **g32_open** function, and **g32_openx** function.

The API file transfer function is the **g32_fxfer** function.

HCON Overview for Programming, Understanding the HCON Application Programming Interfaces, Understanding the AIX Interface for HCON API, API error codes, Sample Flows of API Programs in *Communications Programming Concepts.*

g32_send_keys Function

Purpose

Sends key strokes to the terminal emulator.

Library

HCON Library

C (libg3270.a)
Pascal (libg3270p.a)
FORTRAN (libg3270f.a)

C Syntax

#include <g32_api.h>
#include <g32_keys.h>

g32_send_keys(*as*,*buffer*)

struct g32_api **as*;
char **buffer*;

Pascal Syntax

const
%include /usr/include/g32keys.inc
function g32sdky (var *as* : g32_api;
buffer : stringptr) **: integer; external;**

FORTRAN Syntax

EXTERNAL G32SENDKEYS
INTEGER *AS*(9), **G32SENDKEYS**
CHARACTER *XX *BUFFER*

RC = G32SENDKEYS(*AS*,*BUFFER*)

Description

The **g32_send_keys** function sends one or more key strokes to a terminal emulator as though they came from the keyboard. ASCII characters are sent by coding their ASCII value. Other keys (such as Enter and the cursor–movement keys) are sent by coding their values from the **g32_keys.h** file (for C programs) or **g32keys.inc** file (for Pascal programs). FORTRAN users send other keys by passing the name of the key through the **G32SENDKEYS** buffer.

The **g32_send_keys** function can only be used in API/3270 mode.

C Parameters

as — Specifies a pointer to the **g32_api** structure. Status is returned in this structure.

buffer — Specifies a pointer to a buffer of key stroke data.

Pascal Parameters

as — Specifies the **g32_api** structure. Status is returned in this structure.

buffer — Specifies a pointer to a string containing the keys to be sent to the host. The string must be at least as long as indicated in the **g32_api** structure.

FORTRAN Parameters

AS — Specifies the **g32_api** equivalent structure as an array of integers.

BUFFER — The character array containing the key sequence to send to the host. A special emulator key can be sent by the **g32_send_keys** function as follows:

```
BUFFER = 'ENTER'//CHAR(0)
RC = G32SENDKEYS (AS,BUFFER)
```

The special emulator strings recognized by the **g32_send_keys** function are as follows:

```
CLEAR     DELETE    DUP       ENTER
EOF       ERASE     FMARK     HOME
INSERT    NEWLINE   RESET     SYSREQ
LEFT      RIGHT     UP        DOWN
LLEFT     RRIGHT    UUP       DDOWN
TAB       BTAB
PA1       PA2       PA3
PF1       PF2       PF3       PF4
PF5       PF6       PF7       PF8
PF9       PF10      PF11      PF12
PF13      PF14      PF15      PF16
PF17      PF18      PF19      PF20
PF21      PF22      PF23      PF24
```

Return Values

Upon successful completion:

- A value of 0 is returned.

Upon unsuccessful completion:

- An error code –1 is returned.
- The **errcode** bit is set to the error code identifying the error.
- The **xerrinfo** bit can be set to give more information about the error.

Examples

C Language

1. The following example fragment illustrates the use of the **g32_send_keys** function in an api_3270 mode program:

   Note: The following example is missing the required **g32_open** and **g32_alloc** functions which are necessary for every HCON Workstation API program.

```
#include <g32_api.h>    /* API include file */
main()
{
struct g32_api *as;           /* g32 structure */

char *buffer;                 /* pointer to char string */
int return;                   /* return code */
char *malloc();               /* C memory allocation function */
.
.
.
return = g32_notify(as,1);    /* Turn notification on */
buffer = malloc(10);
return = g32_get_cursor(as);  /* get location of cursor */
printf (" The cursor positionis row: %d col: %d/n";
            as -> row, as -> column);
/* Get data from host starting at the current row and column */
as -> length = 10;            /* length of a pattern on host */
return = g32_get_data(as,buffer); /* get data from host */
printf("The data returned is <%s>\n",buffer);

/* Try to search for a particular pattern on host */
as ->row =1;                    /* row to start search */
as ->column =1;                 /* column to start search */
return = g32_search(as,"PATTERN");

/*Send a clear key to the host *?
strcpy (buffer, "CLE/0");
return = g32_send_keys(as, buffer);

/* Turn notification off */
return = g32_notify(as,0);
.
.
.
```

Implementation Specifics

The **g32_send_keys** function is part of the AIX 3270 Host Connection Program/6000 (HCON).

The **g32_send_keys** function requires one of the following network communication adapters:

- IBM 3270 Connection Adapter and attachment cables for connection to an IBM 3174/3274 Control Unit, IBM 4361 Work Station Adapter, or an IBM 9370 Work Station Subsystem Controller configured for non–SNA distributed function terminal (non–SNA DFT) mode.
- IBM System/370 Host Interface Adapter and attachment cables for connection to an IBM 5088 Graphics Control Unit.

The **g32_send_keys** function requires one of the following IBM System/370 operating system environments be installed on the System/370: VM/SP CMS, VM/XA CMS, MVS/SP TSO/E, or MVS/XA TSO/E.

The **g32_send_keys** function is not available for Japanese Language Support.

Files

/usr/include/g32_api.h	Contains data structures and associated symbol definitions.
/usr/include/g32_keys.h	Defines key values for C language use.
/usr/include/g32keys.inc	Defines key values for Pascal language use.
/usr/include/g32const.inc	Defines Pascal API constants.
/usr/include/g32hfile.inc	Defines Pascal API external definitions.
/usr/include/g32types.inc	Defines Pascal API data types.

Related Information

Additional Logical Terminal Interface functions are the **g32_get_cursor** function, **g32_get_data** function, **g32_notify** function, and **g32_search** function.

AIX session control functions are the **g32_alloc** function, **g32_close** function, **g32_dealloc** function, **g32_open** function, and **g32_openx** function.

The API file transfer function is the **g32_fxfer** function.

HCON Overview for Programming, Understanding the HCON Application Programming Interfaces, Understanding the AIX Interface for HCON API, API error codes, Sample Flows of API Programs in *Communications Programming Concepts.*

G32WRITE Function

Purpose

Sends a message to an AIX API application running simultaneously on the RISC System/6000.

Syntax

G32WRITE *MSG,LEN*

Description

The **G32WRITE** function sends a message to an AIX API application. The maximum number of bytes that may be transferred is specified by the value returned in R0 after a successful completion of the **G32ALLOC** function.

The **G32 WRITE** function is a HCON API function that can be called by a 370 Assembler applications program.

Parameters

MSG The address of the message to be sent. It may be:

- *Label* A label on a DC or DS statement declaring the message.
- *0(reg)* A register containing the address of the message.

LEN The length, specified in bytes, of the message. It is a full word, whose contents cannot exceed the value returned by the **G32ALLOC** function in R0. It must be:

- *Label* The address of a full word containing the length of the message.

Return Values

The **G32WRITE** function sets register 0 (zero) to the following values:

0 Zero. A normal return; call successful.

< 0 Less than zero. Host API error condition.

Examples

The following 370 Assembler code example illustrates the use of the host **G32WRITE** function:

```
L R11,=v(G32DATA)
USING G32DATAD,R11
G32WRITE MSG1, LEN1      /* write "Hello" to AIX */
LTR R0,R0                /* check return code    */
BE WRITEOK               /* if good, go to write */
( error code )
.
.
.
MSG1 DC  C 'HELLO'
LEN1 DC  AL4(*—MSG1)
```

Implementation Specifics

The **G32WRITE** function is part of the AIX 3270 Host Connection Program/6000 (HCON).

The **G32WRITE** function requires one of the following network communication adapters:

- IBM 3270 Connection Adapter and attachment cables for connection to an IBM 3174/3274 Control Unit, IBM 4361 Work Station Adapter, or an IBM 9370 Work Station Subsystem Controller configured for non–SNA distributed function terminal (non–SNA DFT) mode.
- IBM System/370 Host Interface Adapter and attachment cables for connection to an IBM 5088 Graphics Control Unit.

The **G32WRITE** function requires one of the following IBM System/370 operating system environments be installed on the System/370: VM/SP CMS, VM/XA CMS, MVS/SP TSO/E, or MVS/XA TSO/E.

The **G32WRITE** function is not available for Japanese Language Support.

Related Information

Additional host interface functions are the **G32ALLOC** function, **G32DLLOC** function, and **G32WRITE** function.

AIX session control subroutines are the **g32_alloc** subroutine, **g32_close** subroutine, **g32_dealloc** subroutine, **g32_open** subroutine, and **g32_openx** subroutine.

AIX message interface subroutines are the **g32_get_status** subroutine, **g32_read** subroutine, and **g32_write** subroutine.

HCON Overview for Programming, Understanding the HCON Application Programming Interfaces, Understanding the HCON Host Interface in *Communications Programming Concepts.*

How to Compile a Host HCON API Program, Host API Errors, Sample Flows of API Programs in *Communications Programming Concepts.*

g32_write Function

Purpose

Sends a message to a host application.

Library

HCON Library

C (libg3270.a)
Pascal (libg3270p.a)
FORTRAN (libg3270f.a)

C Syntax

#include <g32_api.h>

g32_write(*as*, *msgbuf*, *msglen*)

struct g32_api **as*;
char **msgbuf*;
int *msglen*;

Pascal Syntax

function g32wrte (*var as : g32_api;*
Buffer : integer;
msglen : integer) : **integer; external;**

FORTRAN Syntax

EXTERNAL G32WRITE
INTEGER *AS*(9), *MSGLEN*, **G32WRITE**
CHARACTER* *XX MSGBUF*

RC = G32WRITE(*AS, MSGBUF, MSGLEN*)

Description

The **g32_write** function sends the message pointed to by the *msgbuf* parameter to the host. This function may only be used by those applications having API/API or API/API_T mode specified by the **g32_alloc** command.

HCON application programs using the Pascal language interface must include and link both the C and Pascal libraries. Applications programs using the FORTRAN language for the HCON API must include and link both the C and FORTRAN libraries.

C Parameters

as — Specifies the pointer to a **g32_api** structure.

msgbuf — Specifies a pointer to a message, which is a byte string.

msglen — Specifies the length, in bytes, of the message pointed to by the *msgbuf* parameter. The value of the *msglen* parameter must be greater than 0 and and less than or equal to the maximum I/O buffer size specified in the HCON session profile.

Pascal Parameters

as — Specifies the **g32_api** structure.

Buffer — Specifies an address of a character–packed array.

Note: The address of a packed array can be obtained by the **addr()** function call: buffer := addr (<msg array name> [1 (one)])

msglen — Specifies an integer indicating the length of the message to send to the host. The *msglen* parameter must be greater than 0 and less than or equal to the maximum I/O buffer size specified in the HCON session profile.

FORTRAN Parameters

AS — Specifies the **g32_api** equivalent structure as an array of integers.

MSGBUF — Specifies a character array containing the data to be sent to the host.

MSGLEN — Specifies the number of bytes to be sent to the host. The *MSGLEN* parameter must be greater than 0 and less than or equal to the maximum I/O buffer size specified in the HCON session profile.

Return Values

Upon successful completion:

- The number of bytes written is returned (>= 0).

Upon unsuccessful completion:

- An error code –1 is returned.
- The **errcode** bit is set to the error code identifying the error.
- The **xerrinfo** bit can be set to give more information about the error.

Example

C Language

1. The following example illustrates the use of the **g32_write** function:

```
#include <g32_api>        /* API include */
main()
{
struct g32_api *as;       /* the g32 structure */

char *messg;                      /* pointer to a character string
                                     to send to the host */
int length;                       /* Number of bytes sent */

char *malloc();                   /* C memory allocation function
*/
int return;                       /* return code is no. of bytes
                                     sent */
.
.
.
```

```
messg = malloc(30);      /* allocate 30 bytes for the string */
                /* initialize message string with information */
strcpy(messg,"string to be sent to host/0"
length = strlen(messg); /* length of the message */
return = g32_write(as,messg,length);
.
.
.
```

Implementation Specifics

The **g32_write** function is part of the AIX 3270 Host Connection Program/6000 (HCON).

The **g32_write** function requires one of the following network communication adapters:

- IBM 3270 Connection Adapter and attachment cables for connection to an IBM 3174/3274 Control Unit, IBM 4361 Work Station Adapter, or an IBM 9370 Work Station Subsystem Controller configured for non–SNA distributed function terminal (non–SNA DFT) mode.
- IBM System/370 Host Interface Adapter and attachment cables for connection to an IBM 5088 Graphics Control Unit.

The **g32_write** function requires one of the following IBM System/370 operating system environments be installed on the System/370: VM/SP CMS, VM/XA CMS, MVS/SP TSO/E, or MVS/XA TSO/E.

The **g32_write** function is not available for Japanese Language Support.

Files

/usr/include/g32_api.h	Contains data structures and associated symbol definitions.
/usr/include/g32const.inc	Defines Pascal API constants
/usr/include/g32hfile.inc	Defines Pascal API external definitions
/usr/include/g32types.inc	Defines Pascal API data types

Related Information

Additional message interface functions are the **g32_get_status** function and **g32_read** function.

AIX session control functions are the **g32_alloc** function, **g32_close** function, **g32_dealloc** function, **g32_open** function, and **g32_openx** function.

The API file transfer functions is the **g32_fxfer** function.

Host interface functions are the **G32ALLOC** function, **G32DLLOC** function, **G32READ** function, and **G32WRITE** function.

HCON Overview for Programming, Understanding the HCON Application Programming Interfaces, Understanding the AIX Interface for HCON API, API error codes, Sample Flows of API Programs in *Communications Programming Concepts.*

IF–ELSE Statement

Purpose

Provides a two–way alternative test for conditional execution of Logon Assist Feature (LAF) statements.

Syntax

IF (*condition*) *t–statement* [**ELSE** *f–statement*]

Description

The **IF–ELSE** statement provides a two–way alternative test for conditional execution of LAF statements. The **IF–ELSE** statement is one of the script statements in the LAF language that are used to compose a LAF script.

Expressions

condition Condition to be evaluated

t–statement Statement performed if condition evaluates true

f–statement Statement performed if condition evaluates false

Example

The statements below search for a pattern. If a match is found, PA2 is sent to the host and a **WAIT** statement is executed, else the program exists with a return code of three (3).

```
IF(MATCH)DO
  SEND(PA2);
  WAIT(1);
  END;
ELSE
  EXIT(3);
```

Implementation Specifics

The **IF–ELSE** statement is part of the Logon Assist Feature of the AIX 3270 Host Connection Program/6000 (HCON).

Related Information

How To Use a Logon Assist Feature Script, Understanding the Logon Assist Feature (LAF) in *Communications Programming Concepts*.

HCON Overview for Programming in *Communications Programming Concepts*.

MATCH Statement

Purpose

Searches for a pattern in the current presentation space.

Syntax

MATCH(*rownum,colnum,string|ARG(N)***);**

Description

The Logon Assist Feature (LAF) **MATCH** statement searches for a pattern in the current presentation space. The presentation space is the characters that appear on a terminal display. The **MATCH** statement searches without waiting for receipt of data from the host. The **MATCH** statement is one of the script statements in the LAF language that are used to compose a LAF script.

The special variable *MATCH* is set to 0 (zero) if the operation is not successful and to 1 (one) if the operation is successful . If the search is successful, the special variables *ROW* and *COL* are set to reflect the location of the beginning of the match in the presentation space.

Note: The **WAIT** statement can be used before **MATCHAT** (or **MATCH)** to control the time delay to receive data from the host before searching the presentation space.

Parameters

rownum	Specifies the row number in the presentation space at which to begin the search for the pattern.
colnum	Specifies the column number in the presentation space to begin searching for the pattern.
string	Contains the string pattern to be used in the search.
ARG(N)	Contains the string pattern that is the Nth argument in the LAF logon ID string and should be used in the search.

Example

The **MATCH** statement searches the entire presentation space for the string `MORE` starting at row 24, column 1.

```
MATCH(24,1,"MORE");
```

Implementation Specifics

The **MATCH** statement is part of the Logon Assist Feature of the AIX 3270 Host Connection Program/6000 (HCON).

Related Information

How To Use a Logon Assist Feature Script, Understanding the Logon Assist Feature (LAF) in *Communications Programming Concepts.*

HCON Overview for Programming in *Communications Programming Concepts.*

MATCHAT Statement

Purpose

Searches for a pattern in the current presentation space.

Syntax

MATCHAT(*rownum,colnum,string|ARG(N)***);**

Description

The **MATCHAT** LAF statement is very similar to the **MATCH** statement. It searches for a pattern in the current presentation space without waiting for receipt of data from the host. The search is successful only if a match is found in the presentation space beginning at the specified position. The **MATCHAT** statement is one of the script statements in the LAF language that are used to compose a LAF script.

If a **MATCHAT** search operation is successful, *ROW* and *COL* are always set equal to *rownum* and *colnum*. The special variable *MATCH* is set to 0 (zero), if the operation is not successful and to 1 (one) for successful completion.

Note: The **WAIT** statement can be used before **MATCHAT (**or **MATCH)** to control the time delay to receive data from the host before searching the presentation space.

Parameters

rownum	Specifies the row number in the presentation space to begin the search for the pattern.
colnum	Specifies the column number in the presentation space at which to search for the pattern.
string	Contains the string pattern to be used in the search
ARG(N)	Contains the string pattern that is the Nth argument in the LAF logon ID string and should be used in the search.

Example

The **MATCHAT** statement searches for `VM/370?ONLINE` string starting at row 1, column 1:

```
MATCHAT(1,1,'VM/370?ONLINE');
```

Implementation Specifics

The **MATCHAT** statement is part of the Logon Assist Feature of the AIX 3270 Host Connection Program/6000 (HCON).

Related Information

The **MATCH** statement and **RECEIVE** statement.

How To Use a Logon Assist Feature Script, Understanding the Logon Assist Feature (LAF) in *Communications Programming Concepts.*

NODEBUG Statement

Purpose

Disables debugging messages in a LAF script.

Syntax

NODEBUG;

Description

The **NODEBUG** statement turns off the generation of run–time debugging messages. The **NODEBUG** statement is one of the script statements in the LAF language that are used to compose a LAF script.

Implementation Specifics

The **NODEBUG** statement is part of the Logon Assist Feature of the AIX 3270 Host Connection Program/6000 (HCON).

Related Information

How To Use a Logon Assist Feature Script, Understanding the Logon Assist Feature (LAF) in *Communications Programming Concepts.*

HCON Overview for Programming in *Communications Programming Concepts.*

RECEIVE Statement

Purpose

Waits for data to be received from the host and then searches the presentation space for a pattern.

Syntax

RECEIVE(*rownum,colnum,string* | *ARG(N)***)**;

Description

The **RECEIVE** statement waits 15 seconds or until data is received from the host and then searches the presentation space for a pattern. The **RECEIVE** statement is one of the script statements in the LAF language that are used to compose a LAF script.

The special variable *MATCH* is set to 0 (zero), if the **RECEIVE** operation is not successful, and to 1 if the operation is successful. If the search is successful, the special variables *ROW* and *COL* are set to reflect the location of the beginning of the match in the presentation space.

Parameters

rownum Specifies the row number in the presentation space at which to begin the search for the pattern.

colnum Specifies the column number in the presentation space at which to begin the search for the pattern.

string Specifies a text string.

ARG(N) Contains the string pattern which is the Nth argument in the LAF logon ID string and should be used in the search.

Example

The **RECEIVE** statement searches for `MORE...`, starting in row 25, column 75:

```
RECEIVE(25,75,'MORE...');
```

Note: The **RECEIVE** statement waits up to 15 seconds to receive data from the host before searching the presentation space, but the **MATCH** and **MATCHAT** statements search immediately. The **WAIT** statement can be used in combination with **MATCH** and **MATCHAT** to control the time delay to receive data from the host.

Implementation Specifics

The **RECEIVE** statement is part of the Logon Assist Feature of the AIX 3270 Host Connection Program/6000 (HCON).

Related Information

The **MATCHAT** statement and **WAIT** statement.

How To Use a Logon Assist Feature Script, Understanding the Logon Assist Feature (LAF) in *Communications Programming Concepts*.

HCON Overview for Programming in *Communications Programming Concepts*.

RECVAT Statement

Purpose

Waits for data to be received from the host and then searches the presentation space for a pattern.

Syntax

RECVAT(*rownum,colnum,string* | *ARG(N)*);

Description

The **RECVAT** statement is very similar to the **RECEIVE** statement. It waits for data to be received from the host and then searches the presentation space for a pattern. The search is successful only if a match is found in the presentation space beginning at the specified position. The **RECVAT** statement is one of the script statements in the LAF language that are used to compose a LAF script.

The special variable MATCH is set to 0 (zero) if the search is not successful and to 1 if the search is successful. If the search is successful, the special variables *ROW* and *COL* are set to indicate the location of the beginning of the match in the presentation space. Unlike the **RECEIVE** statement, if a **RECVAT** statement is successful, *ROW* and *COL* are always set equal to the *rownum* and *colnum* parameters, respectively.

Parameters

rownum	Specifies the row number in the presentation space at which to begin the search for the pattern.
colnum	Specifies the column number in the presentation space at which to begin the search for the pattern.
string	Contains the string pattern to be used in the search
ARG(N)	Contains the string pattern which is the Nth argument in the LAF logon ID string and should be used in the search.

Example

The **RECVAT** statement searches for the string passed in the fifth token of the logon ID string. The search begins at row 3, column 1:

```
RECVAT(3,1,ARG(4));
```

Implementation Specifics

The **RECVAT** statement is part of the Logon Assist Feature of the AIX 3270 Host Connection Program/6000 (HCON).

Related Information

How To Use a Logon Assist Feature Script, Understanding the Logon Assist Feature (LAF) in *Communications Programming Concepts.*

HCON Overview for Programming in *Communications Programming Concepts.*

REPEAT–UNTIL Statement

Purpose

Executes LAF script subject statement until the tested condition is found to be true.

Syntax

REPEAT *statemenlist* **UNTIL** (*condition*);

Description

The **REPEAT–UNTIL** statement executes the subject statement until the tested condition is found to be true.

Expressions

statementlist	Statement or statements to be executed until condition is true.
condition	Condition that halts execution of **REPEAT–UNTIL** loop,when true.

Example

The following **REPEAT–UNTIL** statement causes the **WAIT** statement to continue to execute until the **TIMEOUT** flag is set:

```
REPEAT
  WAIT(2);
UNTIL(TIMEOUT);
```

Implementation Specifics

The **REPEAT–UNTIL** statement is part of the Logon Assist Feature of the AIX 3270 Host Connection Program/6000 (HCON).

Related Information

How To Use a Logon Assist Feature Script, Understanding the Logon Assist Feature (LAF) in *Communications Programming Concepts.*

HCON Overview for Programming in *Communications Programming Concepts.*

SELECT Statement

Purpose

Provides a multiple alternative test for conditional execution of Logon Assist Feature (LAF) statements.

Syntax

SELECT; WHEN–*clause* **[OTHERWISE–***clause***] END**;

WHEN (*condition*) *statement*
OTHERWISE *statement*

Description

The **SELECT** statement provides a multiple alternative test for conditional execution of LAF statements. The **SELECT** statement is one of the script statements in the LAF language that are used to compose a LAF script.

Reserved Words

WHEN–*clause*	Evaluates each statement in the WHEN clause until a true condition is found. The statement in the WHEN clause is then executed and control passes to the next statement following the SELECT statement. There may be multiple WHEN clauses in a SELECT statement.
OTHERWISE–*clause*	Executed only if none of the **WHEN** clauses is true.

Expressions

Condition	A condition, when true, causes the statement in the **WHEN** clause to be executed.
Statement	Statement to execute.

If there is no OTHERWISE clause and none of the WHEN clauses are true, the SELECT statement does nothing.

Example

These statements check for the `ENTER*PASSWORD:` string starting at row 1, column 1 until a timeout occurs. If the timeout occurs the routine exits with a return code of three (3).

```
REPEAT
  DO;
  MATCHAT(1,1,'ENTER*PASSWORD:');
    SELECT;
      WHEN(NOT MATCH) WAIT(2);
      WHEN(TIMEOUT) EXIT(3);
    END;
  END;
UNTIL(MATCH);
```

Implementation Specifics

The **SELECT** statement is part of the Logon Assist Feature of the AIX 3270 Host Connection Program/6000 (HCON).

Related Information

How To Use a Logon Assist Feature Script, Understanding the Logon Assist Feature (LAF) in *Communications Programming Concepts.*

HCON Overview for Programming in *Communications Programming Concepts.*

SEND Statement

Purpose

Sends a string of keys to the emulator and from there to the host.

Syntax

SEND(*string* | *keydef* | *UID* | *PW* | *ARG(N)*);

Description

The **SEND** statement sends a string of keys to an emulator and from there to the host.

- If the *string* parameter is coded, that *string* is sent to the host.
- If the *keydef* parameter is coded, that special key is sent to the host.
- If any one or more of the *UID*, *PW*, or *ARG(N)* parameters are coded, they are passed as parameters to the LAF script and sent as strings to the host.

The **SEND** statement is one of the script statements in the LAF language that are used to compose a LAF script.

Parameters

string	Specifies a text string
keydef	Contains the string or key definition to be sent to the host.
UID	Specifies the host user ID.
PW	Specifies the password associated with the *UID*.
ARG(N)	Contains the string pattern which is the Nth argument in the LAF logon ID string and should be used in the search.

Example

The **SEND** statement sends the Enter key to the host:

```
SEND(ENTER);
```

Implementation Specifics

The **SEND** statement is part of the Logon Assist Feature of the in AIX 3270 Host Connection Program/6000 (HCON).

START Statement

Purpose

Begins a Logon Assist Feature (LAF) script.

Syntax

START [*string*]

Description

The **START** statement begins a LAF script. The **START** statement is one of the script statements in the LAF language that are used to compose a LAF script.

Parameters

string	Defines the name of the generated C function. If a name is not supplied, the **g32_logon** script is used. Each script must have one **START** statement, which must be the first statement in the script.

Example

The following **START** statement specifies the start of a new script labeled **g32_logoff**:

```
START "g32_logoff";
```

Implementation Specifics

The **START** statement is part of the Logon Assist Feature of the AIX 3270 Host Connection Program/6000.

Related Information

How To Use a Logon Assist Feature Script, Understanding the Logon Assist Feature (LAF) in *Communications Programming Concepts.*

HCON Overview for Programming in *Communications Programming Concepts.*

WAIT Statement

Purpose

Causes the Logon Assist Feature (LAF) script to wait until data is received from the host or until the specified number of seconds has elapsed.

Syntax

WAIT(*number*);

Description

The **WAIT** statement causes the LAF script to wait until data is received from the host or until the specified number of seconds has elapsed. The special variable *TIMEOUT* is set to 0 (zero) if data is received from the host and to one (1) if the specified time has elapsed.

Note: Use of the WAIT statement at the beginning of a script is not a good practice as the initial data from the host is received immediately.

The **WAIT** statement is one of the script statements in the LAF language that are used to compose a LAF script.

Expression

number Specifies in seconds, the amount of time to wait. A negative value indicates that the **WAIT** statement only returns when data is received from the host. A value of 0 indicates that the **WAIT** statement returns immediately.

Example

This statement executes a one–second wait if a match is not found:

```
IF(NOT MATCH) WAIT(1);
```

Due to variability in the amount of time it may take to log on, the **WAIT** statement should be used sparingly outside of loops. **REPEAT–UNTIL** loops are another means of waiting for an event to occur at the host.

The following loop can be used instead of a **WAIT** statement:

```
REPEAT
  DO
  MATCH(1,1,'MORE');
    IF (MATCH) DO;
      SEND(PA2);
      END;
  MATCH(1,1,'R; T');
    IF (NOT MATCH) DO
      WAIT(2);
    IF (TIMEOUT)
      BREAK;
      END;
  END;
UNTIL (MATCH);
```

Implementation Specifics

The **WAIT** statement is part of the Logon Assist Feature of the AIX 3270 Host Connection Program/6000 (HCON).

Related Information

How To Use a Logon Assist Feature Script, Understanding the Logon Assist Feature (LAF) in *Communications Programming Concepts*.

HCON Overview for Programming in *Communications Programming Concepts*.

WHILE Statement

Purpose

Executes a Logon Assist Feature (LAF) script subject statement.

Syntax

WHILE (*condition*) *statement*

Description

The **WHILE** statement executes a subject statement as long as the tested condition remains true. The **WHILE** statement is one of the script statements in the LAF language that are used to compose a LAF script.

Expression

condition A condition may be any of the following:

MATCH
TIMEOUT
RECOVERY
ROW <comparison operator> <number>
COL <comparison operator> <number>
<condition> *AND* <condition>
<condition> *OR* <condition>
NOT <condition>

Example

The following example is a condition in the LAF language. Conditions are used in the **WHILE**, **REPEAT–UNTIL**, **IF–ELSE**, and **SELECT** statements.

```
WHILE(NOT TIMEOUT) WAIT(2);
```

The **WHILE** statement continues to execute until the *TIMEOUT* flag is set.

Implementation Specifics

The **WHILE** statement is part of the Logon Assist Feature of the AIX 3270 Host Connection Program/6000 (HCON).

Related Information

How To Use a Logon Assist Feature Script, Understanding the Logon Assist Feature (LAF) in *Communications Programming Concepts*.

HCON Overview for Programming in *Communications Programming Concepts*.

Data Link Controls

close Subroutine Interface for Data Link Control (dlc) Devices

Purpose

Closes the GDLC device manager using a file descriptor.

Syntax

int close (*fildes***);**
int *fildes*;

Description

The **close** subroutine disables a generic data link control (GDLC) channel. If this is the last channel to close on a port, the GDLC device manager is reset to an idle state on that port and the communications device handler is closed.

Parameter

fildes	Specifies the file descriptor of the GDLC being closed.

Return Values

Upon successful completion, the **close** subroutine returns a value of 0 (zero).

If an error occurs, a value of –1 is returned with one of the following error numbers available using **errno**, as defined in the **errno.h** header file:

EBADF	Bad file number

Implementation Specifics

This **close** subroutine interface is part of the *device manager* Data Link Control in BOS Extensions 2.

Insert the Standard Ethernet, SDLC, Token-Ring, IEEE Etherent (802.3), or X.25 QLLC (or any combination) in place of *device manager* above, depending on which device manager you decide to use.

Related Information

The **close** subroutine.

open Subroutine Interface for Data Link Control (dlc) Devices

Generic Data Link Control (GDLC) Environment Overview in *Communications Programming Concepts.*

dlcclose Entry Point of the GDLC Device Manager

Purpose

Entry point to close a GDLC channel.

Syntax

#include <sys/device.h>

int dlcclose (*devno*, *chan*, *ext*)
dev_t *devno*;
int *chan*, *ext*;

Note: The **dlc** prefix is replaced with the 3-digit prefix for the specific GDLC device manager being closed.

Description

The **dlcclose** routine is called when a user's application program invokes the **close** subroutine or when a kernel user calls the **fp_close** kernel service. This routine disables a generic data link control (GDLC) channel for the user. If this is the last channel to close on the port, the GDLC device manager issues a close to the network device handler and deletes the kernel process that serviced device handler events on behalf of the user.

Parameters

devno	Indicates major and minor device numbers. This is a **dev_t** device number that specifies both the major and minor device numbers of the GDLC device manager. There is one **dev_t** device number for each type of GDLC, such as Ethernet, Token-Ring, or SDLC.
chan	Specifies the channel ID assigned by GDLC in the **dlcmpx** routine at open time.
ext	Specifies the extended subroutine parameter. This parameter is ignored by GDLC.

Return Values

Upon successful completion, this service returns a value of 0 (zero).

If an error occurs, the following error value is returned, as defined in the **errno.h** header file:

EBADF	Bad file number.

Implementation Specifics

This **dlcclose** entry point of the GDLC is part of the *device manager* Data Link Control in BOS Extensions 2.

Insert the Standard Ethernet, SDLC, Token-Ring, IEEE Etherent (802.3), or X.25 QLLC (or any combination) in place of *device manager* above, depending on which device manager you decide to use.

Related Information

The **ddclose** device entry point.

The **fp_close** kernel service.

The **close** subroutine.

dlcopen Entry Point of the GDLC Device Manager.

dlcmpx Entry Point of the GDLC Device Manager.

Generic Data Link Control (GDLC) Environment Overview in *Communications Programming Concepts.*

dlcconfig Entry Point of the GDLC Device Manager

Purpose

Entry point to configure the GDLC device manager.

Syntax

#include <sys/uio.h>
#include <sys/device.h>

int dlcconfig (*devno*, *op*, *uiop***)**
dev_t *devno*;
int *op*;
struct uio **uiop*;

Note: The **dlc** prefix is replaced with the 3-digit prefix for the specific GDLC device manager being configured.

Description

The **dlcconfig** routine is called during the kernel startup procedures to initialize the GDLC device manager with its device information. This routine is also called by the operating system when the GDLC is being terminated or queried for vital product data.

Parameters

devno — Indicates major and minor device numbers. This is a **dev_t** device number that specifies both the major and minor device numbers of the GDLC device manager. There is one **dev_t** device number for each type of GDLC, such as Ethernet, Token-Ring, or SDLC.

op — Specifies the operation code that indicates the function to be performed:

- **INIT** — Initializes the GDLC device manager.
- **TERM** — Terminates the GDLC device manager.
- **QVPD** — Queries GDLC vital product data. This operation code is optional.

uiop — A pointer to the **uio** structure specifying the location and length of the caller's data area for the INIT and QVPD operation codes. No data areas are specifically defined for GDLC, but DLC's may define the data areas for a particular network.

Return Values

Upon successful completion, this service returns a value of 0 (zero).

If an error occurs, one of the following error values is returned, as defined in the **errno.h** header file:

EINVAL	Invalid value
ENODEV	No such device handler
EFAULT	Kernel service, such as **uiomove** or **devswadd**, has failed.

Implementation Specifics

This **dlcconfig** entry point of the GDLC is part of the *device manager* Data Link Control in BOS Extensions 2.

Insert the Standard Ethernet, SDLC, Token-Ring, IEEE Etherent (802.3), or X.25 QLLC (or any combination) in place of *device manager* above, depending on which device manager you decide to use.

Related Information

The**ddconfig** device entry point.

The **uiomove** kernel service.

Generic Data Link Control (GDLC) Environment Overview in *Communications Programming Concepts.*

dlcioctl Entry Point of the GDLC Device Manager

Purpose

Entry point to issue specific commands to GDLC.

Syntax

#include <sys/device.h>
#include <sys/gdlextcb.h>

int dlcioctl (*devno*, *op*, *arg*, *devflag*, *chan*, *ext***)**
dev_t *devno*;
ulong_t *devflag*;
int *op*, *arg*, *chan*, *ext*;

Note: The **dlc** prefix is replaced with the 3-digit prefix for the specific GDLC device manager being controlled.

Description

The **dlcioctl** routine is called when a user's application program invokes the **ioctl** subroutine or when a kernel user calls the **fp_ioctl** kernel service. The **dlcioctl** routine decodes commands for special functions in the generic data link control (GDLC).

Parameters

devno Indicates major and minor device numbers. This is a **dev_t** device number that specifies both the major and minor device numbers of the GDLC device manager. There is one **dev_t** device number for each type of GDLC, such as Ethernet, Token-Ring, or SDLC.

op Specifies the parameter from the subroutine that specifies the operation to be performed. Ioctl Operations for DLC provides a listing of all possible operators.

arg Indicates the parameter from the subroutine that specifies the address of a parameter block. Parameter Blocks by ioctl Operation for DLC provides a listing of all possible arguments.

devflag Specifies the flag word with the following flags defined:

- **DKERNEL** Entry point called by kernel routine using the **fp_open** kernel service. This indicates that the **arg** parameter points to kernel space.
- **DREAD** Open for reading. This flag is ignored.
- **DWRITE** Open for writing. This flag is ignored.
- **DAPPEND** Open for appending. This flag is ignored.
- **DNDELAY** Device open in nonblocking mode. This flag is ignored.

chan Specifies the channel ID assigned by GDLC in the **dlcmpx** routine at open time.

ext Specifies the extended subroutine parameter. This parameter is ignored by GDLC.

Return Values

Upon successful completion, this service returns a value of 0.

If an error occurs, one of the following error values is returned, as defined in the **errno.h** header file:

EBADF Bad file number

EINVAL Invalid value

ENOMEM Not enough resources to satisfy the **ioctl** subroutine.

Implementation Specifics

This **dlcioctl** entry point of the GDLC is part of the *device manager* Data Link Control in BOS Extensions 2.

Insert the Standard Ethernet, SDLC, Token-Ring, IEEE Etherent (802.3), or X.25 QLLC (or any combination) in place of *device manager* above, depending on which device manager you decide to use.

Related Information

The **ddioctl** device entry point.

The **fp_ioctl** kernel service, fp_open kernel service.

The **ioctl** subroutine.

dlcmpx Entry Point of the GDLC Device Manager.

Generic Data Link Control (GDLC) Environment Overview in *Communications Programming Concepts.*

dlcmpx Entry Point of the GDLC Device Manager

Purpose

Entry point to decode the device handlers special file name appended to the open call.

Syntax

#include <sys/device.h>

int dlcmpx (*devno*, *chanp*, *channame*)
dev_t *devno*;
int **chanp*;
char **channame*;

Note: The **dlc** prefix is replaced with the 3-digit prefix for the specific GDLC device manager being opened.

Description

The **dlcmpx** routine is called by the operating system when a generic data link control (GDLC) channel is being allocated. This routine decodes the name of the device handler that is appended to the end of the GDLC's special file name at open time. GDLC allocates the channel and returns the value in the *chanp* parameter.

This routine is also called following a **close** subroutine to deallocate the channel. In this case the *chanp* parameter is passed to GDLC in order to identify the channel being deallocated. Since GDLC allocates a new channel for each **open** subroutine, there is a **dlcmpx** routine following each call to the **dlcclose** routine.

Parameters

devno Indicates major and minor device numbers. This is a **dev_t** device number that specifies both the major and minor device numbers of the GDLC device manager. There is one **dev_t** device number for each type of GDLC, such as Ethernet, Token-Ring, or SDLC.

chanp Specifies the channel ID returned if a valid path name exists for the device handler, and the **openflag** is set. If no channel ID is allocated, this field is set to a value of –1 by GDLC.

channame Specifies a pointer to the appended path name (path name extension) of the device handler that is used by GDLC to attach to the network. If this is NULL, the channel is to be deallocated.

Return Values

Upon successful completion, this service returns a value of 0 (zero).

If an error occurs, one of the following error values is returned, as defined in the **errno.h** header file:

EBADF Bad file number.

EINVAL Invalid value.

Implementation Specifics

This **dlcmpx** entry point of the GDLC is part of the *device manager* Data Link Control in BOS Extensions 2.

Insert the Standard Ethernet, SDLC, Token-Ring, IEEE Etherent (802.3), or X.25 QLLC (or any combination) in place of *device manager* above, depending on which device manager you decide to use.

Related Information

The **ddmpx** device entry point.

The **close** subroutine, **open** subroutine.

dlcclose Entry Point of the GDLC Device Manager.

dlcopen Entry Point fo the GDLC Device Manager.

Generic Data Link Control (GDLC) Environment Overview in *Communications Programming Concepts.*

dlcopen Entry Point of the GDLC Device Manager

Purpose

Entry point to open a GDLC channel.

Syntax

#include <sys/device.h>
#include <sys/gdlextcb.h>

int dlcopen (*devno*, *devflag*, *chan*, *ext*)
dev_t *devno*;
ulong_t *devflag*;
int *chan*, *ext*;

Note: The **dlc** prefix is replaced with the 3-digit prefix for the specific GDLC device manager being opened.

Description

The **dlcopen** routine is called when a user's application program invokes the **open** or **openx** subroutine, or when a kernel user calls the **fp_open** kernel service. The generic data link control (GDLC) device manager opens the specified communications device handler and creates a kernel process to catch posted events from that port. Additional opens to the same port share both the device handler open and the GDLC kernel process created on the original open.

Note: It may be more advantageous to handle the actual device handler open and kernel process creation in the **dlcmpx** routine. This is left as a specific DLC's option.

Parameters

devno — Indicates major and minor device numbers. This is a **dev_t** device number that specifies both the major and minor device numbers of the GDLC device manager. There is one **dev_t** device number for each type of GDLC, such as Ethernet, Token-Ring, or SDLC.

devflag — Specifies the flag word with the following flags defined:

DKERNEL	Entry point called by kernel routine using the **fp_open** kernel service. All command extensions and **ioctl** arguments will be in kernel space.
DREAD	Open for reading. This flag is ignored.
DWRITE	Open for writing. This flag is ignored.
DAPPEND	Open for appending. This flag is ignored.
DNDELAY	Device open in non-blocking mode. This flag is ignored.

chan — Specifies the channel ID assigned by GDLC in the **dlcmpx** routine.

ext — Specifies the extended subroutine parameter. This is a pointer to the **dlc_open_ext** extended I/O structure for **open** subroutine.

Return Values

Upon successful completion, this service returns a value of 0 (zero).

If an error occurs, one of the following error values is returned, as defined in the **errno.h** header file:

ECHILD	Cannot create a kernel process.
EINVAL	Invalid value.
ENODEV	No such device handler.
ENOMEM	Not enough resources to satisfy the **open** subroutine.
EFAULT	Kernel service, such as **copyin** or **initp**, failed.

Implementation Specifics

This **dlcopen** entry point of the GDLC is part of the *device manager* Data Link Control in BOS Extensions 2.

Insert the Standard Ethernet, SDLC, Token-Ring, IEEE Etherent (802.3), or X.25 QLLC (or any combination) in place of *device manager* above, depending on which device manager you decide to use.

Related Information

The **ddopen** device entry point.

The **open**, **openx** subroutine.

The**fp_open** kernel service, **copyin** kernel service, **initp** kernel service.

dlcclose Entry Point of the GDLC Device Manager.

dlcmpx Entry Point of the GDLC Device Manager.

Generic Data Link Control (GDLC) Environment Overview in *Communications Programming Concepts*.

dlcread Entry Point of the GDLC Device Manager

Purpose

Entry point to read receive data from GDLC.

Syntax

#include <sys/device.h>
#include <sys/gdlextcb.h>

int dlcread (*devno*, *uiop*, *chan*, *ext*)
dev_t *devno*;
struct uio **uiop*;
int *chan*, *ext*;

Note: The **dlc** prefix is replaced with the 3-digit prefix for the specific GDLC device manager being read.

Description

The **dlcread** routine is called when a user's application program invokes the **readx** subroutine. Kernel users do *not* call an **fp_read** kernel service. All receive data is returned to the user in the same order as received. The type of data that was read is indicated, as well as the service access point (SAP) and link station (LS) identifiers.

The following fields in the **uio** and **iov** structures are used to control the read-data transfer operation:

uio_iov Points to an **iovec** structure.

uio_iovcnt Number of elements in the **iovec** structure. This must be set to a value of 1. Vectored read operations are not supported.

uio_offset The file offset established by a previous **fp_lseek** subroutine. This field is ignored by generic data link control (GDLC).

uio_segflag Indicates whether the data area is in application or kernel space. This is set to the UIO_USERSPACE value by the file I/O subsystem to indicate application space.

uio_fmode Contains the value of the file mode set with the **open** applications subroutine to GDLC.

uio_resid This field is initially the total byte count of the receive data area. GDLC decrements this count for each packet byte received using the **uiomove** subroutine.

iovec structure A structure that contains the starting address and length of the received data.

iov_base A variable in the **iovec** structure where GDLC writes the address of the received data.

iov_len A variable in the **iovec** structure that contains the byte length of the data.

Parameters

devno	Indicates major and minor device numbers. This is a **dev_t** device number that specifies both the major and minor device numbers of the GDLC device manager. There is one **dev_t** device number for each type of GDLC, such as Ethernet, Token-Ring, or SDLC.
uiop	Points to the **uio** structure containing the read parameters.
chan	Specifies the channel ID assigned by GDLC in the **dlcmpx** routine at open time.
ext	Specifies the extended subroutine parameter. This is a pointer to the extended I/O structure. The argument to this parameter must always be in the application space. DLC Extended Parameters for read Subroutine provides more information on this parameter.

Return Values

Reads that are successful and reads that must be truncated due to limited user data space each return a value of 0. If more data is received from the media than will fit into the application data area, the DLC_OFLO value indicator is set in the command extension area (**dlc_io_ext**) to indicate that the read is truncated. All excess data is lost.

If other errors occur, one of the following error values is returned, as defined in the **errno.h** header file:

EBADF	Bad file number.
EINTR	A signal interrupted the subroutine before it received data.
EINVAL	Invalid value.
ENOMEM	Not enough resources to satisfy the read.

Implementation Specifics

This **dlcread** entry point of the GDLC is part of the *device manager* Data Link Control in BOS Extensions 2.

Insert the Standard Ethernet, SDLC, Token-Ring, IEEE Etherent (802.3), or X.25 QLLC (or any combination) in place of *device manager* above, depending on which device manager you decide to use.

Related Information

The **ddread** device entry point.

The **fp_read** kernel service.

The **readx** subroutine, **fp_lseek** subroutine, **uiomove** subroutine, **open** subroutine.

dlcmpx Entry Point of the GDLC Device Manager.

DLC Extended Parameters for read Subroutine.

dlcwrite Entry Point of the GDLC Device Manager.

Generic Data Link Control (GDLC) Environment Overview in *Communications Programming Concepts.*

dlcselect Entry Point of the GDLC Device Manager

Purpose

Entry point to select for asynchronous criteria from GDLC, such as receive data completion and exception conditions.

Syntax

#include <sys/device.h>
#include <sys/gdlextcb.h>

int dlcselect (*devno*, *events*, *reventp*, *chan*)
dev_t *devno*;
ushort_t *events*;
ushort_t **reventp*;
int *chan*;

Note: The **dlc** prefix is replaced with the three-digit prefix for the specific GDLC device manager being selected.

Description

The **dlcselect** routine is called when a user's application program invokes a **select** or **poll** subroutine. This allows the user to select receive data or exception conditions. The DPOLLOUT write-availability criteria is not supported. If no results are available at the time of a **select** subroutine, the user process is put to sleep until an event occurs.

If one or more events specified in the *events* parameter are true, the **dlcselect** routine updates the returned events parameter (passed by reference), *reventp*, by setting the corresponding event bits that indicate which events are currently true.

If none of the requested events are true, the **dlcselect** routine sets the returned events parameter to a value of 0 (passed by reference using the *reventp* parameter) and checks the **DPOLLSYNC** flag in the *events* parameter. If this flag is true, the routine returns because the event request was a synchronous request. If the **DPOLLSYNC** flag is false, an internal flag is set for each event requested in the *events* parameter.

When one or more of the requested events become true, generic data link control (GDLC) issues the **selnotify** kernel service to notify the kernel that a requested event or events have become true. The internal flag indicating that the event was being requested is then reset to prevent renotification of the event.

If the port in use is in a closed state, implying that the requested event or events can never be satisfied, GDLC sets the returned events flags to a value of 1 for each event that can never be satisfied. This is done so that the **select** or **poll** subroutine does not wait indefinitely.

Kernel users do *not* call an **fp_select** kernel service since their receive data and exception notification functions are called directly by GDLC. The DLC Extended Parameters for open Subroutine details how these function handlers are specified.

Parameters

devno Indicates major and minor device numbers. This is a **dev_t** device number that specifies both the major and minor device numbers of the GDLC device manager. There is one **dev_t** device number for each type of GDLC, such as Ethernet, Token-Ring, or SDLC.

events Identifies the events that are to be checked. The following events are:

DPOLLIN Read selection.

DPOLLOUT Write selection. This is not supported by GDLC.

DPOLLPRI Exception selection.

DPOLLSYNC This request is a synchronous request only. The routine should *not* perform a **selnotify** kernel service routine due to this request if the events occur later.

reventp Identifies a returned events pointer. This is a parameter passed by reference to indicate which of the selected events are true at the time of the call. See the preceding *events* parameter for possible values.

chan Specifies the channel ID assigned by GDLC in the **dlcmpx** routine at open time.

Return Values

Upon successful completion, this service returns a value of 0 (zero).

If an error occurs, one of the following error numbers is returned, as defined in the **errno.h** header file:

EBADF Bad file number.

EINTR A signal interrupted the subroutine before it found any of the selected events.

EINVAL The specified DPOLLOUT write selection is not supported.

Implementation Specifics

This **dlcselect** entry point of the GDLC is part of the *device manager* Data Link Control in BOS Extensions 2.

Insert the Standard Ethernet, SDLC, Token-Ring, IEEE Etherent (802.3), or X.25 QLLC (or any combination) in place of *device manager* above, depending on which device manager you decide to use.

Related Information

The **ddselect** device entry point.

The **select** subroutine, **poll** subroutine.

The **fp_select** kernel service.

DLC Extended Parameters for open Subroutine.

dlcselect Entry Point of the GDLC Device Manager.

dlcmpx Entry Point of the GDLC Device Manager.

Generic Data Link Control (GDLC) Environment Overview in *Communications Programming Concepts.*

dlcwrite Entry Point of the GDLC Device Manager

Purpose

Entry point to write transmit data to GDLC.

Syntax

#include <sys/uio.h>
#include <sys/device.h>

#include <sys/gdlextcb.h>
int dlcwrite (*devno, uiop, chan, ext***)**
dev_t *devno*;
struct uio **uiop*;
int *chan, ext*;

Note: The **dlc** prefix is replaced with the 3-digit prefix for the specific GDLC device manager being written.

Description

The **dlcwrite** routine is called when a user's application program invokes a **writex** subroutine or when a kernel user calls the **fp_write** kernel service. An extended write is used in order to specify the type of data being sent, as well as the service access point (SAP) and link station (LS) identifiers.

The following fields in the **uio** and **iov** structures are used to control the write data transfer operation:

uio_iov Points to an **iovec** structure.

uio_iovcnt Number of elements in the **iovec** structure. This must be set to a value of 1 for the kernel user, indicating that there is a single communications memory buffer (**mbuf**) chain associated with the **write** subroutine.

uio_offset The file offset established by a previous **fp_lseek** kernel service. This field is ignored by GDLC.

uio_segflag Indicates whether the data area is in application or kernel space. This field is set to the UIO_USERSPACE value by the file I/O subsystem if the data area is in application space. The field must be set to the UIO_SYSSPACE value by the kernel user to indicate kernel space.

uio_fmode Contains the value of the file mode set during an application **open** subroutine to GDLC or can be set directly during a kernel user's **fp_open** kernel service to GDLC.

uio_resid For application users this field is set to the total byte count of the transmit data area. For kernel users, GDLC ignores this field since the communications memory buffer (**mbuf**) also carries this information.

iovec structure A structure that contains the starting address and length of the transmit. (See the **iov_base** field and **iov_len** field.)

iov_base A variable in the **iovec** structure where GDLC gets the address of the application user's transmit data area or the address of the kernel user's transmit **mbuf**.

iov_len A variable in the **iovec** structure that contains the byte length of the application user's transmit data area. This variable is ignored by GDLC for kernel users, since the transmit **mbuf** contains a length field.

Parameters

devno Indicates major and minor device numbers. This is a **dev_t** device number that specifies both the major and minor device numbers of the GDLC device manager. There is one **dev_t** device number for each type of GDLC, such as Ethernet, Token-Ring, or SDLC.

uiop Points to the **uio** structure containing the write parameters.

chan Specifies the channel ID assigned by GDLC in the **dlcmpx** routine at open time.

ext Specifies the extended subroutine parameter. This is a pointer to the extended I/O structure. This data must be in the application space if the **iov_fmode** field indicates an application subroutine or in the kernel space if the **iov_fmode** field indicates a kernel subroutine. DLC Extended Parameters for Write provides more information on this parameter.

Return Values

Upon successful completion, this service returns a value of 0 (zero).

If an error occurs, one of the following error values is returned, as defined in the **errno.h** header file:

EAGAIN Transmit is temporarily blocked, and a sleep cannot be issued.

EBADF Bad file number (application).

EINVAL Invalid value, such as too much data for a single packet.

ENOMEM Not enough resources to satisfy the **write** subroutine, such as a lack of communications memory buffers (**mbufs**).

ENXIO Invalid file pointer (kernel).

Implementation Specifics

This **dlcwrite** entry point of the GDLC is part of the *device manager* Data Link Control in BOS Extensions 2.

Insert the Standard Ethernet, SDLC, Token-Ring, IEEE Etherent (802.3), or X.25 QLLC (or any combination) in place of *device manager* above, depending on which device manager you decide to use.

Related Information

The **ddwrite** device entry point.

dlcmpx Entry Point of the GDLC Device Manager.

The **writex** subroutine, **open** subroutine.

The **fp_write** kernel service, **fp_lseek** kernel service, **fp_open** kernel service.

dlcread Entry Point of the GDLC Device Manage.

Generic Data Link Control (GDLC) Environment Overview in *Communications Programming Concepts.*

fp_close Kernel Service for Data Link Control (DLC) Devices

Purpose

Allows kernel closes to the GDLC device manager using a file pointer.

Syntax

int fp_close (*fp*, *ext***);**
struct file ****fp*;**

Description

The **fp_close** kernel service disables a generic data link control (GDLC) channel. If this is the last channel to close on a port, the GDLC device manager resets to an idle state on that port and the communications device handler is closed.

Parameters

fp	Specifies the file pointer of the GDLC being closed.
ext	Specifies the extension parameter. This parameter is ignored by GDLC.

Return Values

Upon successful completion, this service returns a value of 0 (zero).

If an error occurs, the following error value is returned, as defined in the **errno.h** header file:

ENXIO	Invalid file pointer.

Implementation Specifics

This **fp_close** kernel service is part of the *device manager* Data Link Control in BOS Extensions 2.

Insert the Standard Ethernet, SDLC, Token-Ring, IEEE Etherent (802.3), or X.25 QLLC (or any combination) in place of *device manager* above, depending on which device manager you decide to use.

Related Information

The **fp_close** kernel service.

fp_open Kernel Service for Data Link Control (DLC) Devices.

Generic Data Link Control (GDLC) Environment Overview in *Communications Programming Concepts.*

fp_ioctl Kernel Service for Data Link Control (DLC) Devices

Purpose

Transfers special commands from the kernel to GDLC using a file pointer.

Syntax

#include <sys/gdlextcb.h>
#include <fcntl.h>

int fp_ioctl (*fp*, *cmd*, *arg*, *ext***)**
struct file **fp*;
unsigned int *cmd*;
caddr_t *arg*;
int *ext*;

Description

Various generic data link control (GDLC) functions can be initiated using the **fp_ioctl** kernel service, such as changing configuration parameters, contacting the remote, and testing a link. Most of these operations can be completed before returning to the user synchronously. Some operations take longer, so asynchronous results are returned some time later using the **exception** function handler. GDLC calls the kernel user's exception handler to complete these results. For more information on the functions that can be initiated using the **fp_ioctl** kernel service, see Ioctl Operations (op) DLC and Parameter Blocks by Operation for DLC.

Note: The DLC_GET_EXCEP **ioctl** command operation is not used since all exception conditions are passed to the kernel user through the exception handler.

Parameters

fp	Specifies the file pointer of the target GDLC.
cmd	Specifies the operation to be performed by GDLC. For a listing of all possible operators, see Ioctl Operations (op) for DLC.
arg	Specifies the address of the parameter block. The argument for this parameter must be in the kernel space. For a listing of possible values, see Parameters Blocks by Operations for DLC.
ext	Specifies the extension parameter. This parameter is ignored by GDLC.

Return Values

Upon successful completion, the **fp_ioctl** kernel service returns a value of 0 (zero).

If an error occurs, one of the following error values is returned, as defined in the **errno.h** header file:

ENXIO	Invalid file pointer
EINVAL	Invalid value
ENOMEM	Not enough resources to satisfy the **ioctl** subroutine.

Implementation Specifics

This **fp_ioctl** kernel service is part of the *device manager* Data Link Control in BOS Extensions 2.

Insert the Standard Ethernet, SDLC, Token-Ring, IEEE Etherent (802.3), or X.25 QLLC (or any combination) in place of *device manager* above, depending on which device manager you decide to use.

Related Information

The **fp_ioctl** kernel service.

The **ioctl** subroutine.

The **ioctl** subroutine.

Generic Data Link Control (GDLC) Environment Overview in *Communications Programming Concepts.*

fp_open Kernel Service for Data Link Control (DLC) Devices

Purpose

Allows kernel opens to the GDLC device manager by its device name.

Syntax

#include <sys/gdlextcb.h>
#include <fcntl.h>

fp_open (*path*, *oflags*, *cmode*, *ext*, *segflag fpp*)
char *path*;
unsigned int *oflags*;
unsigned int *cmode*;
int *ext*;
unsigned int *segflag*;
struct file ***fpp*;

Description

The **fp_open** kernel service allows the kernel user to open a generic data link control (GDLC) device manager by specifying the special file names of both the DLC and the communications device handler. Since the GDLC device manager is multiplexed, more than one process can open it (or the same process multiple times) and still have unique channel identifications.

Each open carries the communications device handler's special file name so that the DLC knows which port to transfer data on.

The kernel user must also provide functional entry addresses in order to obtain receive data and exception conditions. Using Special Kernel Services provides related information.

Parameters

path
: Consists of a character string containing the **/dev** special file name of the GDLC device manager, with the name of the communications device handler appended. The format is shown in the following example:

```
/dev/dlcether/ent0
```

oflags
: Specifies a value to set the file status flag. The GDLC device manager ignores all but the following values:

O_RDWR
: Open for reading and writing. This must be set for GDLC or the open will fail.

O_NDELAY, O_NONBLOCK
: Subsequent writes return immediately if no resources are available. The calling process is not put to sleep.

cmode
: Specifies the O_CREAT mode parameter. This is ignored by GDLC.

ext
: Specifies the extended kernel service parameter. This is a pointer to the **dlc_open_ext** extended I/O structure for **open** subroutines. The argument

for this parameter must be in the kernel space. DLC Extended Parameters for open Subroutine provides more information on the extension parameter.

segflag — Specifies the segment flag indicating where the *path* parameter is located:

FP_SYS — The *path* parameter is stored in kernel memory.

FP_USR — The *path* parameter is stored in application memory.

fpp — Specifies the returned file pointer. This parameter is passed by reference and updated by the file I/O subsystem to be the file pointer for this **open** subroutine.

Return Values

Upon successful completion, this service returns a value of 0 (zero) and a valid file pointer in the *fp* parameter.

If an error occurs, one of the following error values is returned as defined in the **errno.h** header file:

ECHILD — Cannot create a kernel process.

EINVAL — Invalid value.

ENODEV — No such device handler.

ENOMEM — Not enough resources to satisfy the open.

EFAULT — Kernel service, such as **copyin** or **initp**, has failed.

Implementation Specifics

This **fp_open** kernel service is part of the *device manager* Data Link Control in BOS Extensions 2.

Insert the Standard Ethernet, SDLC, Token-Ring, IEEE Etherent (802.3), or X.25 QLLC (or any combination) in place of *device manager* above, depending on which device manager you decide to use.

Related Information

The **fp_open** kernel service, **copyin** kernel service, **initp** kernel service.

fp_close Kernel Service for Data Link Control (DLC) Devices.

DLC Extended Parameters for the open Subroutine.

Generic Data Link Control (GDLC) Environment Overview in *Communications Programming Concepts*.

fp_write Kernel Service for Data Link Control (DLC) Devices

Purpose

Allows kernel data to be sent using a file pointer.

Syntax

#include <sys/gdlextcb.h>
#include <sys/fp_io.h>

int fp_write (*fp*, *buf*, *nbytes*, *ext*, *segflag*, *countp***)**
struct file **fp*;
char **buf*;
int *nbytes*;
int *ext*;
int *segflag*;
int **countp*;

Description

Four types of data can be sent to GDLC. Network data can be sent to a service access point (SAP), and normal, Exchange Identification (XID), or datagram data can be sent to a link station (LS).

Kernel users pass a communications memory buffer (**mbuf**) directly to generic data link control (GDLC) on the **fp_write** kernel service. In this case, a **uiomove** kernel service is not required, and maximum performance can be achieved by merely passing the buffer pointer to GDLC. Each write buffer is required to have the proper buffer header information and enough space for the data link headers to be inserted. A write data offset is passed back to the kernel user at start LS completion for this purpose.

All data must fit into a single packet for each write call. That is, GDLC does not separate the user's write data area into multiple transmit packets. A maximum write data size is passed back to the user at DLC_ENABLE_SAP completion and at DLC_START_LS completion for this purpose.

Normally, a **write** subroutine can be satisfied immediately by GDLC by completing the data link headers and sending the transmit packet down to the device handler. In some cases, however, transmit packets can be blocked by the particular protocol's flow control or a resource outage. GDLC reacts to this differently, based on the system blocked/nonblocked file status flags (set by the file system and based on the O_NDELAY and O_NONBLOCKED values passed on the **fp_open** kernel service). Nonblocked **write** subroutines that cannot get enough resources to queue the communications memory buffer (**mbuf**) return an error indication. Blocked **write** subroutines put the calling process to sleep until the resources free up or an error occurs.

Parameters

fp	Specifies file pointer returned from the **fp_open** kernel service.
buf	Points to a kernel **mbuf**.
nbytes	Contains the byte length of the write data. It is not necessary to set this field to the actual length of write data, however, since the **mbuf** contains a length

field. Instead, this field can be set to any non-negative value (generally set to 0 (zero)).

ext Specifies the extended kernel service parameter. This is a pointer to the **dlc_io_ext** extended I/O structure for writes. The argument for this parameter must be in the kernel space. For more information on this parameter, see DLC Extended Parameters for write Subroutine.

segflag Specifies the segment flag indicating where the *path* parameter is located. The only valid value is:

FP_SYS The *path* parameter is stored in kernel memory.

countp Points to the location where a count of bytes actually written is to be returned (must be in kernel space). GDLC does not provide this information for a kernel user since **mbufs** are used, but the file system requires a valid address and writes a copy of the *nbytes* parameter to that location.

Return Values

Upon successful completion, this service returns a value of 0 (zero).

If an error occurs, one of the following error values is returned, as defined in the **errno.h** header file:

EAGAIN Transmit is temporarily blocked, and the calling process cannot be put to sleep.

EINVAL Invalid argument, such as too much data for a single packet.

ENXIO Invalid file pointer.

Implementation Specifics

This **fp_write** kernel service is part of the *device manager* Data Link Control in BOS Extensions 2.

Insert the Standard Ethernet, SDLC, Token-Ring, IEEE Etherent (802.3), or X.25 QLLC (or any combination) in place of *device manager* above, depending on which device manager you decide to use.

Related Information

The **fp_write** kernel service.

The **uiomove** subroutine, **fp_open** kernel service.

Parameter Blocks by ioctl Operation for DLC.

DLC Extended Parameters for the write Subroutine.

Generic Data Link Control (GDLC) Environment Overview in *Communications Programming Concepts.*

ioctl Subroutine Interface for Data Link Control (DLC) Devices

Purpose

Transfers special commands to GDLC using a file descriptor.

Syntax

#include <sys/ioctl.h>
#include <sys/devinfo.h>
#include <sys/gdlextcb.h>

int ioctl (*fildes*, *op*, *arg*);
int *fildes*;
int *op*;
char **arg*;

Description

The **ioctl** subroutine initiates various generic data link control (GDLC) functions, such as changing configuration parameters, contacting a remote link, and testing a link. Most of these operations can be completed before returning to the user (synchronously). Since some operations take longer, asynchronous results are returned some time later using the exception condition notification. Application users can obtain these exceptions using the DLC_GET_EXCEP **ioctl** operation. For more information on the functions that can be initiated using the **ioctl** subroutine, see Ioctl Operations for DLC (**op**) and Parameter Blocks by Operations for DLC.

Parameters

fildes	Specifies the file descriptor of the target GDLC.
op	Specifies the operation to be performed by GDLC. For a listing of all possible operators, see Ioctl Operations.
arg	Specifies the address of the parameter block. For a listing of possible values, see Parameter Blocks by Operations for DLC.

Return Values

Upon successful completion, the **ioctl** subroutine returns a value of 0 (zero).

If an error occurs, a value of –1 is returned with one of the following error numbers available using **errno**, as defined in the **errno.h** header file:

EBADF	Bad file number.
EINVAL	Invalid argument.
ENOMEM	Not enough resources to satisfy the **ioctl** subroutine.

Implementation Specifics

This **ioctl** subroutine interface is part of the *device manager* Data Link Control in BOS Extensions 2.

Insert the Standard Ethernet, SDLC, Token-Ring, IEEE Etherent (802.3), or X.25 QLLC (or any combination) in place of *device manager* above, depending on which device manager you decide to use.

Related Information

Ioctl Operations (op) for DLC and Parameter Blocks by Operations for DLC .

The **ioctl** subroutine.

Generic Data Link Control (GDLC) Environment Overview in *Communications Programming Concepts.*

ioctl Subroutine Operations (op) for DLC

Description

GDLC supports the following **ioctl** command operations:

```
#define DLC_ENABLE_SAP  1
#define DLC_DISABLE_SAP 2
#define DLC_START_LS    3
#define DLC_HALT_LS             4
#define DLC_TRACE               5
#define DLC_CONTACT             6
#define DLC_TEST                7
#define DLC_ALTER               8
#define DLC_QUERY_SAP   9
#define DLC_QUERY_LS    10
#define DLC_ENTER_LBUSY 11
#define DLC_EXIT_LBUSY  12
#define DLC_ENTER_SHOLD 13
#define DLC_EXIT_SHOLD  14
#define DLC_GET_EXCEP   15
#define DLC_ADD_GRP            16
#define IOCINFO         /* see /usr/include/sys/ioctl.h */
```

DLC_ADD_GRP Add a group or multicast receive address to a port. This command allows additional address values to be filtered in receive as supported by the individual communication device handlers. See the device handler specifications to determine which address values are supported.

DLC_ALTER Alters link station (LS) configuration.

DLC_CONTACT Contacts the remote LS. This **ioctl** operation does not complete processing before returning to the user. The DLC_CONTACT notification is returned asynchronously to the user using exception.

DLC_DISABLE_SAP Disables a service access point (SAP). This **ioctl** operation does not fully complete the disable SAP processing before returning to the user. The DLC_DISABLE_SAP notification is returned asynchronously to the user some time later using exception.

DLC_ENABLE_SAP Enables a SAP. This **ioctl** operation does not fully complete the enable SAP processing before returning to the user. The DLC_ENABLE_SAP notification is returned asynchronously to the user some time later using exception.

DLC_ENTER_LBUSY Enters local busy mode on an LS.

DLC_ENTER_SHOLD Enters short hold mode on an LS.

DLC_EXIT_LBUSY Exits local busy mode on an LS.

DLC_EXIT_SHOLD Exits short hold mode on an LS.

DLC_GET_EXCEP Returns asynchronous exception notifications to the application user.

Note: This **ioctl** command operation is not used by the kernel user since all exception conditions are passed to the kernel user via their exeception handler routine.

DLC_HALT_LS Halts an LS. This **ioctl** operation does not complete processing before returning to the user. Notification of the **ioctl** operation, DLC_HALT_LS, is returned asynchronously to the user using exception.

DLC_QUERY_LS Queries an LS.

DLC_QUERY_SAP Queries a SAP.

DLC_START_LS Starts an LS. This **ioctl** operation does not complete processing before returning to the user. Notification of the **ioctl** operation, DLC_START_LS, is returned asynchronously to the user using exception.

DLC_TEST Tests LS connectivity. This **ioctl** operation does not complete processing before returning to the user. Notification of the **ioctl** operation, DLC_TEST completion, is returned asynchronously to the user using exception.

DLC_TRACE Traces LS activity.

IOCINFO Returns a structure that describes the device. Refer to the description of the **sys/devinfo.h** file in *AIX Version 3 Application Programming Interface, File Formats.* The first byte is set to an **ioctype** of DD_DLC. The subtype and data are defined by the individual DLC devices.

Implementation Specifics

These **ioctl** operations for DLC are part of the *device manager* Data Link Control in BOS Extensions 2.

Insert the Standard Ethernet, SDLC, Token-Ring, IEEE Etherent (802.3), or X.25 QLLC (or any combination) in place of *device manager* above, depending on which device manager you decide to use.

Related Information

Generic Data Link Control (GDLC) Environment Overview in *Communications Programming Concepts.*

ioctl Subroutine Operations Parameter Blocks for DLC

Each command operation has a specific parameter block associated with the command that is pointed to by the **arg** pointer. Some parameters are sent to GDLC and some are returned.

The ioctl command operations for DLC are as follows:

- DLC_ENABLE_SAP ioctl Operation for DLC
- DLC_DISABLE_SAP ioctl Operation for DLC
- DLC_START_LS ioctl Operation for DLC
- DLC_HALT_LS ioctl Operation for DLC
- DLC_TRACE ioctl Operation for DLC
- DLC_CONTACT ioctl Operation for DLC
- DLC_TEST ioctl Operation for DLC
- DLC_ALTER ioctl Operation for DLC
- DLC_QUERY_SAP ioctl Operation for DLC
- DLC_QUERY_LS ioctl Operation for DLC
- DLC_ENTER_LBUSY ioctl Operation for DLC
- DLC_EXIT_LBUSY ioctl Operation for DLC
- DLC_ENTER_SHOLD ioctl Operation for DLC
- DLC_EXIT_SHOLD ioctl Operation for DLC
- DLC_GET_EXCEP ioctl Operation for DLC
- DLC_ADD_GRP ioctl Operation for DLC
- IOCINFO ioctl Operation for DLC.

DLC_ENABLE_SAP ioctl Operation for DLC

The following parameter enables a service access point (SAP).

```
#define DLC_MAX_NAME    20  /* maximum size of the address/name   */
#define DLC_MAX_GSAPS   7   /* maximum number of group sap        */
#define DLC_MAX_ADDR    8   /* maximum byte length of an address */

struct dlc_esap_arg
    {
    ulong_t gdlc_sap_corr;                 /* GDLC SAP correlator  */
                                           /* RETURNED             */
    ulong_t user_sap_corr;                /* User's SAP correlator */
    ulong_t len_func_addr_mask;           /* length of the field   */
                                           /* below it             */
    uchar_t func_addr_mask[DLC_MAX_ADDR];/* Mask of the valid     */
                                           /* functional address   */
    ulong_t len_grp_addr;                 /* length of the field   */
                                           /* below it             */
    uchar_t grp_addr[DLC_MAX_ADDR];     /* Address of group packet */
                                           /* to be received       */
    ulong_t max_ls;                       /* Max number of link    */
                                           /* stations per SAP     */
```

```
    ulong_t flags;                          /* Enable SAP flags      */
    ulong_t len_laddr_name;                 /* Length of the local   */
                                             /* name/address         */
    u_char_t laddr_name[DLC_MAX_NAME];     /* The local address/name */
    u_char_t num_grp_saps;                  /* Number of group SAPs  */
    u_char_t grp_sap[DLC_MAX_GSAPS];      /* Group SAPs the SAP will */
                                             /* rsp to               */
    u_char_t res1[3];                        /* reserved             */
    u_char_t local_sap;                     /* ID of local SAP       */
};
```

gdlc_sap_corr GDLC SAP correlator: The GDLC's service access point (SAP) identifier that is returned to the user. This correlator must accompany all subsequent commands associated with this service access point.

user_sap_corr User SAP correlator: The user's SAP identifier to be returned by GDLC on all SAP results. It allows routing of the SAP-specific results when multiple SAPs have been opened by a single user.

len_func_addr_mask
Length of functional address mask: Specifies the byte length of the following functional address mask. This field must be set to 0 (zero) if no functional address is required. Length values of 0 through 8 are supported.

func_addr_mask
Functional address mask: The functional address mask to be ORed with the functional address on the adapter. This address mask allows packets that are destined for specified functions to be received by the local adapter. See the individual DLC interface documentation to determine the format and length of this field.

Note: GDLC does not distinguish whether a received packet was accepted by the adapter due to a pre-set network, group, or functional address. If the SAP address matches and the packet is otherwise valid (no protocol errors, for instance), the received packet is passed to the user.

len_grp_addr Length of group address: Specifies the byte length of the following group address. This field must be set to 0 if no group address is required. Length values of 0 through 8 are supported.

grp_addr Group address: The group address value to be written to the adapter. It allows packets that are destined for a specific group to be received by the local adapter.

Note: Most adapters allow only one group address to be active at a time. If this field is nonzero and the adapter rejects the group address because it is already in use, the enable SAP call fails with an appropriate error code.

max_ls Maximum link stations (LS): Specifies the maximum number of LSs allowed to operate concurrently on a particular SAP. This field can be set to a value from 1 through 255 inclusive.

flags Common SAP flags: The following flags are supported:

```
#define DLC_ESAP_NTWK   0x40000000  /* teleprocessing network     */
                                    /* type (LEASED)              */
#define DLC_ESAP_LINK   0x20000000  /* teleprocessing link        */
                                    /* type (multi)               */
#define DLC_ESAP_PHYC   0x10000000  /* physical network call      */
#define DLC_ESAP_ANSW   0x08000000  /* teleprocessing auto        */
                                    /* call/answer                */
#define DLC_ESAP_ADDR   0x04000000  /* local address/name         */
                                    /* indicator (ADDR)           */
```

DLC_ESAP_NTWK Teleprocessing network type:

0 = Switched (default)
1 = Leased.

DLC_ESAP_LINK Teleprocessing link type:

0 = Point to point (default)
1 = Multipoint.

DLC_ESAP_PHYC Physical network call (teleprocessing):

0 = Listen for incoming call.
1 = Initiate call.

DLC_ESAP_ADDR Local address or name indicator:

0 = Local name specified (default)
1 = Local address specified.

Specifies whether the local address or name field contains an address or a name.

DLC_ESAP_ANSW Teleprocessing autocall or autoanswer:

0 = Manual call and answer (default)
1 = Automatic call and answer.

len_laddr_name
Length of local address or name: Specifies the byte length of the following local address or name. Length values of 1 through 20 are supported.

laddr_name Local address or name: Contains the unique network name or address of the user's local SAP as indicated by the **DLC_ESAP_ADDR** flag.

num_grp_saps Number of group SAPs: Specifies the number of group SAPs the user's local SAP responds to. If no group SAPs are needed, this field must contain a 0. Up to seven group SAPs can be specified.

grp_sap Group SAP array: Contains the specific group SAP values that the user's local SAP responds to (maximum of seven).

local_sap Local SAP address: Specifies the local SAP address being opened. Receive packets with this LSAP value indicated in the destination SAP field are routed to the LSs opened under this particular SAP.

Protocol Specific Data Area
Optional: Allows parameters to be defined by the specific GDLC device manager, such as X.21 call-progress signals or smartmodem call-establishment data. This data area must directly follow (or append to) the end of the **dlc_esap_arg** structure.

Implementation Specifics

This DLC_ENABLE_SAP ioctl operation for DLC is part of the *device manager* Data Link Control in BOS Extensions 2.

Insert the Standard Ethernet, SDLC, Token-Ring, IEEE Ethernet (802.3), or X.25 QLLC (or any combination) in place of *device manager* above, depending on which device manager you decide to use.

Related Information

Parameter Blocks by ioctl Operation for DLC on page .

Generic Data Link Control (GDLC) Environment Overview in *Communications Programming Concepts.*

DLC_DISABLE_SAP ioctl Operation for DLC

The following parameter disables a service access point (SAP).

```
struct dlc_corr_arg
    {
    ulong_t gdlc_sap_corr;  /* GDLC SAP correlator                  */
    ulong_t gdlc_ls_corr;   /* << not used for disabling a SAP >> */
    };
```

gdlc_sap_corr GDLC SAP correlator: Indicates the GDLC SAP identifier to be disabled.

Implementation Specifics

This DLC_DISABLE_SAP ioctl operation for DLC is part of the *device manager* Data Link Control in BOS Extensions 2.

Insert the Standard Ethernet, SDLC, Token-Ring, IEEE Ethernet (802.3), or X.25 QLLC (or any combination) in place of *device manager* above, depending on which device manager you decide to use.

Related Information

Generic Data Link Control (GDLC) Environment Overview in *Communications Programming Concepts.*

DLC_START_LS ioctl Operation for DLC

The following parameter starts a link station (LS) on a particular SAP as caller or listener.

```
#define DLC_MAX_DIAG      16  /* the maximum string of chars      */
                              /*        in the diag name          */

struct dlc_sls_arg
      {
      ulong_t gdlc_ls_corr;    /* GDLC User link station          */
                                /* correlator                     */
      u_char_t ls_diag[DLC_MAX_DIAG]; /* the char name of the ls  */
      ulong_t gdlc_sap_corr;          /* GDLC SAP correlator      */
      ulong_t user_ls_corr;           /* User's SAP correlator    */
      ulong_t flags;                  /* Start Link Station flags */
      ulong_t trace_chan;             /* Trace Channel            */
                                      /* (rc of trcstart)         */
      ulong_t len_raddr_name;         /* Length of the remote     */
                                       /* name/addr               */
      u_char_t raddr_name[DLC_MAX_NAME]; /* The Remote addr/name  */
      ulong_t maxif;                     /* Maximum number of byte*/
                                          /* in an I-field        */
      ulong_t rcv_wind;                  /* Maximum size of the   */
                                          /* receive window       */
      ulong_t xmit_wind;                 /* Maximum size of the */
                                          /* transmit window      */
      u_char_t rsap;                     /* Remote SAP value      */
      u_char_t rsap_low;                 /* Remote SAP low range  */
                                          /* value              */
      u_char_t rsap_high;                /* Remote SAP high range */
                                          /* value                */
      u_char_t res1;                      /* Reserved             */
      ulong_t max_repoll;                /* Maximum Repoll count  */
      ulong_t repoll_time;               /* Repoll timeout value  */
      ulong_t ack_time;                  /* Time to delay trans of*/
                                          /* an ack               */
      ulong_t inact_time;                /* Time before inactivity*/
                                          /* times out            */
      ulong_t force_time;                /* Time before a forced  */
                                          /* disconnect           */
      };
```

gdlc_ls_corr GDLC LS correlator: The GDLC LS identifier returned to the user as soon as resources are determined to be available. This correlator must accompany all commands associated with this LS.

ls_diag LS diagnostic tag: Any ASCII 1 to 16-character name to be written to GDLC trace, error log, and status entries for LS identification. (The end-of-name delimiter is the AIX null character).

gdlc_sap_corr GDLC SAP correlator: The correlator returned by GDLC when the SAP is enabled by the user. This correlator identifies the user's service access point to the GDLC protocol process.

user_ls_corr User LS correlator: The user's LS identifier to be returned by GDLC on all results and data. It allows routing of the station-specific results when multiple logical links have been started by a single user.

flags Common LS flags: The following flags are supported:

```
#define DLC_TRCO            0x80000000    /* Trace Control On        */
#define DLC_TRCL            0x40000000    /* Trace Control Long      */
                                          /* (full packet)           */
#define DLC_SLS_STAT        0x20000000    /* Station type for SDLC   */
                                          /* (primary)               */
#define DLC_SLS_NEGO        0x10000000  /*Negotiate Station Type for*/
                                         /* SDLC                    */
#define DLC_SLS_HOLD        0x08000000 /* Hold link on inactivity   */
#define DLC_SLS_LSVC        0x04000000 /* Link Station Virtual Call */
#define DLC_SLS_ADDR        0x02000000 /* Address Indicator         */
                                        /* (not discovery)          */
```

DLC_TRCO Trace control on:

0 = Disable link trace.

1 = Enable link trace.

DLC_TRCL Trace control long:

0 = Link trace entries are short (80 bytes).

1 = Link trace entries are long (full packet).

DLC_SLS_STAT Station type for SDLC:

0 = Secondary (default)

1 = Primary.

DLC_SLS_NEGO Negotiate station type for SDLC:

0 = No (default)

1 = Yes.

DLC_SLS_HOLD Hold link on inactivity:

0 = No (default), terminate the LS.

1 = Yes, hold it active.

DLC_SLS_LSVC LS virtual call:

0 = Listen for incoming call.

1 = Initiate call.

DLC_SLS_ADDR Address indicator:

0 = Remote is identified by name (discovery).

1 = Remote is identified by address (resolve, SDLC).

trace_chan Trace channel: Specifies the channel number obtained from the **trcstart** subroutine. This field is valid only if the **DLC_TRCO** indicator is set active.

len_raddr_name
Length of remote's address or name: Specifies the byte length of the remote address or name. This field must be set to 0 (zero) if no remote address or name is required to start the LS. Length values of 0 through 20 are supported.

raddr_name Remote's address or name: Contains the unique network address of the remote node if the **DLC_SLS_ADDR** indicator is set active. Contains the unique network name of the remote node if the **DLC_SLS_ADDR** indicator is reset. Addresses are entered in hexadecimal notation, and names are entered in character notation. This field is only valid if the previous length field is nonzero.

maxif Maximum I-field length: Specifies the maximum number of I-field bytes that can be in one packet. This value is reduced by GDLC if the device handler's buffer sizes are too small to hold the maximum I-field specified here. The resultant size is returned from GDLC when the link station has been started.

rcv_wind Receive window: The receive window specifies the maximum number of sequentially numbered receive I-frames the local station can accept prior to sending an acknowledgment.

xmit_wind Transmit window: The transmit window specifies the maximum number of sequentially numbered transmitted I-frames that can be outstanding at any time.

rsap Remote SAP: Specifies the remote service access point address being called. This field is valid only if the **DLC_SLS_LSVC** indicator or the **DLC_SLS_ADDR** indicator is set active.

rsap_low RSAP low range: Specifies the lowest value in the range of remote SAP address values that the local SAP responds to when listening for a remote-initiated attachment. This value cannot be the Null SAP (0x00) or the Discovery SAP (0xFC), and must have the low-order bit set to 0 (B'nnnnnnn0') to indicate an individual address.

rsap_high RSAP high range: Specifies the highest value in the range of remote SAP address values that the local SAP responds to, when listening for a remote-initiated attachment. This value cannot be the Null SAP (0x00) or the Discovery SAP (0xFC), and must have the low-order bit set to 0 (B'nnnnnnn0') to indicate an individual address.

max_repoll Maximum repoll count: Specifies the maximum number of retries for an unacknowledged command frame, or in the case of an I-frame time out, the number of times the nonresponding remote link station is polled with a supervisory command frame.

repoll_time Repoll time-out value: Contains the time-out value (in increments defined by the specific GDLC) used to specify the amount of time allowed prior to retransmitting an unacknowledged command frame.

ack_time Acknowledgment time-out: Contains the time-out value (in increments defined by the specific GDLC) used to specify the amount of time to delay the transmission of an acknowledgment for a received I-frame.

inact_time Inactivity time-out value: Contains the time-out value (in increments of 1 second) used to specify the maximum amount of time allowed before receive inactivity returns an error.

force_time Force halt time-out value: Contains the time-out value (in increments of 1 second) specifying the period to wait for a normal disconnection. Once the time-out occurs, the disconnection is forced and the link station halted.

Protocol Specific Data Area
Optional: Allows parameters to be defined by a specific GDLC device manager, such as token-ring dynamic window increment or SDLC primary slow poll. This data area must directly follow (or append to) the end of the **dlc_sls_arg** structure.

Implementation Specifics

This DLC_START_LS ioctl operation for DLC is part of the *device manager* Data Link Control in BOS Extensions 2.

Insert the Standard Ethernet, SDLC, Token-Ring, IEEE Ethernet (802.3), or X.25 QLLC (or any combination) in place of *device manager* above, depending on which device manager you decide to use.

Related Information

Generic Data Link Control (GDLC) Environment Overview in *Communications Programming Concepts.*

DLC_HALT_LS ioctl Operation for DLC

The following parameter halts a link station (LS).

```
struct dlc_corr_arg
      {
      ulong_t gdlc_sap_corr;      /* GDLC SAP correlator          */
      ulong_t gdlc_ls_corr;       /* GDLC link station correlator */
      };
```

gdlc_sap_corr GDLC SAP correlator: The GDLC SAP identifier of the target LS.

gdlc_ls_corr GDLC LS correlator: The GDLC LS identifier to be halted.

Implementation Specifics

This DLC_HALT_LS ioctl operation for DLC is part of the *device manager* Data Link Control in BOS Extensions 2.

Insert the Standard Ethernet, SDLC, Token-Ring, IEEE Ethernet (802.3), or X.25 QLLC (or any combination) in place of *device manager* above, depending on which device manager you decide to use.

Related Information

Generic Data Link Control (GDLC) Environment Overview in *Communications Programming Concepts*.

DLC_TRACE ioctl Operation for DLC

The following parameter traces a link stations (LS) activity for short or long activities.

```
struct dlc_trace_arg
      {
      ulong_t gdlc_sap_corr;      /* GDLC SAP correlator              */
      ulong_t gdlc_ls_corr;       /* GDLC link station correlator     */
      ulong_t trace_chan;         /* Trace Channel (rc of trcstart) */
      ulong_t flags;              /* Trace Flags                      */
      };
```

gdlc_sap_corr GDLC SAP correlator: The correlator returned by GDLC when the SAP was enabled by the user. This correlator identifies the user's service access point to the GDLC protocol process.

gdlc_ls_corr GDLC LS correlator: The correlator returned by GDLC when the LS was started by the user. This correlator identifies the user's LS to the GDLC protocol process.

trace_chan Trace channel: Specifies the channel number obtained from the **trcstart** subroutine. This field is only valid if the **DLC_TRCO** indicator is set active.

flags Trace flags: The following flags are supported:

```
#define DLC_TRCO          0x80000000    /* Trace Control On        */
#define DLC_TRCL          0x40000000    /* Trace Control Long      */
                                         /* (full packet)           */
```

DLC_TRCO Trace control on:

0 = Disable link trace.

1 = Enable link trace.

DLC_TRCL Trace control long:

0 = Link trace entries are short (80 bytes).

1 = Link trace entries are long (full packet).

Implementation Specifics

This DLC_TRACE ioctl operation for DLC is part of the *device manager* Data Link Control in BOS Extensions 2.

Insert the Standard Ethernet, SDLC, Token-Ring, IEEE Ethernet (802.3), or X.25 QLLC (or any combination) in place of *device manager* above, depending on which device manager you decide to use.

Related Information

Generic Data Link Control (GDLC) Environment Overview in *Communications Programming Concepts.*

DLC_CONTACT ioctl Operation for DLC

The following parameter contacts a remote station for a particular local link station (LS).

```
struct dlc_corr_arg
      {
      ulong_t gdlc_sap_corr;      /* GDLC SAP correlator         */
      ulong_t gdlc_ls_corr;       /* GDLC link station correlator */
      };
```

gdlc_sap_corr GDLC SAP correlator: The GDLC SAP identifier of the target LS.

gdlc_ls_corr GDLC LS correlator: The GDLC LS identifier to be contacted.

Implementation Specifics

This DLC_CONTACT ioctl operation for DLC is part of the *device manager* Data Link Control in BOS Extensions 2.

Insert the Standard Ethernet, SDLC, Token-Ring, IEEE Ethernet (802.3), or X.25 QLLC (or any combination) in place of *device manager* above, depending on which device manager you decide to use.

Related Information

Generic Data Link Control (GDLC) Environment Overview in *Communications Programming Concepts.*

DLC_TEST ioctl Operation for DLC

The following parameter tests the link to a remote for a particular local link station (LS).

```
struct dlc_corr_arg
      {
      ulong_t gdlc_sap_corr;      /* GDLC SAP correlator         */
      ulong_t gdlc_ls_corr;       /* GDLC link station correlator */
      };
```

gdlc_sap_corr GDLC SAP correlator: The GDLC SAP identifier of the target LS.

gdlc_ls_corr GDLC LS correlator: The GDLC LS identifier to be tested.

Implementation Specifics

This DLC_TEST ioctl operation for DLC is part of the *device manager* Data Link Control in BOS Extensions 2.

Insert the Standard Ethernet, SDLC, Token-Ring, IEEE Ethernet (802.3), or X.25 QLLC (or any combination) in place of *device manager* above, depending on which device manager you decide to use.

Related Information

Generic Data Link Control (GDLC) Environment Overview in *Communications Programming Concepts.*

DLC_ALTER ioctl Operation for DLC

The following parameter alters a link station's (LS) configuration parameters.

```
#define DLC_MAX_ROUT     20   /* Maximum Size of Routing Info     */

struct dlc_alter_arg
      {
      ulong_t gdlc_sap_corr;  /* GDLC SAP correlator              */
      ulong_t gdlc_ls_corr;   /* GDLC link station correlator     */
      ulong_t flags;          /* Alter Flags                      */
      ulong_t repoll_time;    /* New Repoll Timeout               */
      ulong_t ack_time;       /* New Acknowledge Timeout          */
      ulong_t inact_time;     /* New Inactivity Timeout           */
      ulong_t force_time;     /* New Force Timeout                */
      ulong_t maxif;          /* New Maximum I-Frame Size         */
      ulong_t xmit_wind;      /* New Transmit Value               */
      ulong_t max_repoll;     /* New Max Repoll Value             */
      ulong_t routing_len;    /* Routing Length                   */
      u_char_t routing[DLC_MAX_ROUT]; /* New Routing Data         */
      ulong_t result_flags;           /* Returned flags           */
      };
```

gdlc_sap_corr GDLC SAP correlator: The GDLC SAP identifier of the target LS.

gdlc_ls_corr GDLC LS correlator: The GDLC LS identifier to be altered.

flags Alter flags: The following flags are supported:

```
#define DLC_ALT_RTO       0x80000000   /* Alter Repoll Timeout */
#define DLC_ALT_AKT       0x40000000   /* Alter Acknowledge Timeout */
#define DLC_ALT_ITO       0x20000000   /* Alter Inactivity Timeout */
#define DLC_ALT_FHT       0x10000000   /* Alter Force Halt Timeout */
#define DLC_ALT_MIF       0x08000000   /* Alter Maximum I-Frame Size*/
#define DLC_ALT_XWIN      0x04000000   /* Alter Tranxmit Window Size*/
#define DLC_ALT_MXR       0x02000000   /* Alter Maximum Repoll Count*/
#define DLC_ALT_RTE       0x01000000   /* Alter Routing            */
#define DLC_ALT_SM1       0x00800000   /* Alter Mode (SDLC) bit 1  */
                                        /* (Primary)               */
#define DLC_ALT_SM2       0x00400000   /* Alter Mode (SDLC) bit 2  */
                                        /* (Secondary)             */
#define DLC_ALT_IT1       0x00200000   /* Alter Inactivity bit 1   */
                                        /* (Notify)                */
#define DLC_ALT_IT2       0x00100000   /* Alter Inactivity bit 2   */
                                        /* (Halt)                  */
```

DLC_ALT_RTO Alter repoll time out:

0 = Do *not* alter repoll time out.

1 = Alter configuration with value specified.

Alters the length of time the LS waits for a response before repolling the remote station. When specified, the repoll time out value specified in the LS's configuration is overridden by the value supplied in the repoll time-out field of the **Alter** command. This new value remains in effect until another value is specified or the LS is halted.

DLC_ALT_AKT Alter acknowledgment time out:

0 = Do *not* alter the acknowledgment time out.

1 = Alter configuration with value specified.

Alters the length of time the LS delays the transmission of an acknowledgment for a received I-frame. When specified, the acknowledgment time out value specified in the LS's configuration is overridden by the value supplied in the acknowledgment time-out field of the **Alter** command. This new value remains in effect until another value is specified or the LS is halted.

DLC_ALT_ITO Alter inactivity time out:

0 = Do *not* alter inactivity time out.

1 = Alter configuration with value specified.

Alters the maximum length of time allowed without receive link activity from the remote station. When specified, the inactivity time-out value specified in the LS's configuration is overridden by the value supplied in the inactivity time-out field of the **Alter** command. This new value remains in effect until another value is specified or the LS is halted.

DLC_ALT_FHT Alter force halt time out:

0 = Do *not* alter force halt time out.

1 = Alter configuration with value specified.

Alters the period to wait for a normal disconnection before forcing the halt LS to occur. When specified, the force halt time-out value specified in the LS's configuration is overridden by the value supplied in the force halt time-out field of the **Alter** command. This new value remains in effect until another value is specified or the LS is halted.

DLC_ALT_MIF Maximum I-field length:

0 = Do *not* alter maximum I-field length.

1 = Alter configuration with value specified.

Sets the value for the maximum length of transmit or receive data in one I-field. If received data exceeds this length, a buffer overflow indication set by GDLC in the receive extension. When specified, the maximum I-field length value specified in the LS's configuration is overridden by the value supplied in the maximum I-field length specified in the **Alter** command. This new value remains in effect until another value is specified or the LS is halted.

DLC_ALT_XWIN Alter transmit window:

0 = Do *not* alter transmit window.

1 = Alter configuration with value specified.

Alters the maximum number of information frames that can be sent in one transmit burst. When specified, the transmit window count value specified in the LS's configuration is overridden by the value supplied in the transmit window field of the **Alter** command. This new value remains in effect until another value is specified or the LS is halted.

DLC_ALT_MXR Alter maximum repoll:

0 = Do *not* alter maximum repoll.

1 = Alter configuration with value specified

Alters the maximum number of retries for an acknowledged command frame, or in the case of an I-frame time out, the number of times the nonresponding remote LS will be polled with a supervisory command frame. When specified, the maximum repoll count value specified in the LS's configuration is overridden by the value supplied in the maximum repoll count field of the **Alter** command. This new value remains in effect until another value is specified or the LS is halted.

DLC_ALT_RTE Alter routing:

0 = Do *not* alter routing.

1 = Alter configuration with value specified.

Alters the route that subsequent transmit packets take when transferring data across a local area network bridge. When specified, the routing length and routing data values specified in the LS's configuration are overridden by the values supplied in the routing fields of the **Alter** command. These new values remain in effect until another route is specified or the LS is halted.

DLC_ALT_SM1 Set SDLC Control mode — primary:

0 = Do *not* alter SDLC Control mode.

1 = Set SDLC Control mode to primary.

Sets the local station to a primary station in NDM, waiting for a command from PU services to write an XID or TEST, or a command to contact the secondary for NRM data phase. This control can only be issued if not already in NRM, and no XID, TEST, or SNRM is in progress. This flag cannot be set if the **DLC_ALT_SM2** flag is set.

DLC_ALT_SM2 Set SDLC Control mode — secondary:

0 = Do *not* alter SDLC Control mode.

1 = Set SDLC Control mode to secondary.

Sets the local station to a secondary station in NDM, waiting for XID, TEST, or SNRM from the primary station. This control can only be issued if not already in NRM, and no XID, TEST, or SNRM is in progress. This flag cannot be set if the **DLC_ALT_SM1** flag is set.

DLC_ALT_IT1 Set Inactivity Time Out mode — notification only:

0 = Do *not* alter Inactivity Time Out mode.

1 = Set Inactivity Time Out mode to notification only.

Inactivity does not cause the LS to be halted, but notifies the user of inactivity without termination.

DLC_ALT_IT2 Set Inactivity Time Out mode — automatic halt:

0 = Do *not* alter Inactivity Time Out mode.

1 = Set Inactivity Time Out mode to automatic halt.

Inactivity causes an automatic halt of the LS with a reason code of inactivity.

repoll_time Repoll time-out value: Provides a new value to replace the LS's repoll time-out value whenever the **DLC_ALT_RTO** flag is set.

ack_time Acknowledge time-out value: Provides a new value to replace the LS's acknowledgment time-out value whenever the **DLC_ALT_AKT** flag is set.

inact_time Inactivity time-out value: Provides a new value to replace the LS's inactivity time-out value whenever the **Alter DLC_ALT_ITO** flag is set.

force_time Force halt time-out value: Provides a new value to replace the LS's force halt time-out value whenever the **DLC_ALT_FHT** flag is set.

maxif Maximum I-field size value: Provides a new value to replace the LS started result value for the maximum I-field size whenever the **DLC_ALT_MIF** flag is set. GDLC does not allow this value to exceed the capacity of receive buffer and only increases the internal value to the allowed maximum.

xmit_wind Transmit window value: Provides a new value to replace the LS's transmit window count value whenever the **DLC_ALT_XWIN** flag is set.

max_repoll Maximum repoll count value: Provides the new value that is to replace the LS's maximum repoll count value whenever the **DLC_ALT_MXR** flag is set.

routing_len Routing field length value: Provides a new value to replace the LS's routing field length whenever the **DLC_ALT_RTE** flag is set.

routing Routing data field value: Provides a new value to replace the LS's routing data whenever the **DLC_ALT_RTE** flag is set.

result_flags Returned result indicator flags: The following result indicators may be returned at the completion of the alter operation, depending on the command:

```
#define DLC_MSS_RES      0x00040000  /* Mode Set Secondary        */
#define DLC_MSSF_RES     0x00020000  /* Mode Set Secondary Failed */
#define DLC_MSP_RES      0x00010000  /* Mode Set Primary          */
#define DLC_MSPF_RES     0x00008000  /* Mode Set Primary Failed   */
```

DLC_MSS_RES Mode set secondary:

This bit set to 1 indicates that the station mode has been set to secondary as a result of the user's issuing an **Alter (set mode secondary)** command.

DLC_MSSF_RES Mode set secondary failed:

This bit set to 1 indicates that the station mode has been *not* set to secondary as a result of the user's issuing an **Alter (set mode secondary)** command. This occurs whenever an SDLC LS is already in data phase or an SDLC primary command sequence has not yet completed.

DLC_MSP_RES Mode set primary:

This bit set to 1 indicates that the station mode has been set to primary as a result of the user's issuing an **Alter (set mode primary)** command.

DLC_MSPF_RES Mode set primary failed:

This bit set to 1 indicates that the station mode has *not* been set to primary as a result of the user's issuing an **Alter (set mode primary)** command. This occurs whenever an SDLC LS is already in data phase.

Protocol Dependent Area

Optional: Allows additional fields to be provided by a specific protocol type. Corresponding flags may be necessary to support additional fields. This data area must directly follow (or append to) the end of the **dlc_alter_arg** structure.

Implementation Specifics

This DLC_ALTER ioctl operation for DLC is part of the *device manager* Data Link Control in BOS Extensions 2.

Insert the Standard Ethernet, SDLC, Token-Ring, IEEE Ethernet (802.3), or X.25 QLLC (or any combination) in place of *device manager* above, depending on which device manager you decide to use.

Related Information

Generic Data Link Control (GDLC) Environment Overview in *Communications Programming Concepts.*

DLC_QUERY_SAP ioctl Operation for DLC

The following parameter queries statistics of a particular service access point (SAP).

```
#define DLC_MAX_DIAG 16 /* the max string of chars in the */
                        /* diag name */

struct dlc_qsap_arg
{
    ulong_t gdlc_sap_corr;      /* GDLC SAP correlator            */
    ulong_t user_sap_corr;      /* user SAP correlator (returned) */
    ulong_t sap_state;          /* state of the SAP, returned by  */
                                 /* the kernel                     */
    uchar_t dev[DLC_MAX_DIAG];  /* the returned device handler's */
                                 /* device name                    */
    ulong_t devdd_len;          /* device driver dependent data   */
                                 /* byte length                    */
    };
```

gdlc_sap_corr GDLC SAP correlator: The GDLC SAP identifier to be queried.

user_sap_corr User SAP correlator: The user's identifier for the SAP, returned for routing purposes.

sap_state Current SAP state: Contains the current state of this SAP:

```
#define DLC_OPENING     1     /* the SAP or link station is in the */
                              /* process of opening                */
#define DLC_OPENED      2     /* the SAP or ls has been opened     */
#define DLC_CLOSING     3     /* the SAP or link station is in the */
                              /* process of closing                */
```

dev Device handler dev name: Contains the /dev name of the communications I/O device handler being used by this SAP.

devdd_len Length of device driver dependent data: Contains the byte length of the expected device driver statistics that will be appended to the **dlc_qsap_arg** structure.

Device Driver Dependent Data
Optional: Contains the device statistics of the attached device handler. This may be the query device statistics (reliability/availability/serviceability log area) returned from a **DLC_Query_LS**, or if supported by the device handler, this may be the result of a **DLC_Query_SAP** issued to the attached device handler. See the individual device handler's specifications for information on the particular fields returned. This data area must directly follow (or append to) the end of the **dlc_qsap_arg** structure.

Implementation Specifics

This DLC_QUERY_SAP ioctl operation for DLC is part of the *device manager* Data Link Control in BOS Extensions 2.

Insert the Standard Ethernet, SDLC, Token-Ring, IEEE Ethernet (802.3), or X.25 QLLC (or any combination) in place of *device manager* above, depending on which device manager you decide to use.

Related Information

Generic Data Link Control (GDLC) Environment Overview in *Communications Programming Concepts.*

DLC_QUERY_LS ioctl Operation for DLC

The following parameter queries statistics of a particular link station (LS).

```
struct dlc_qls_arg
     {
     ulong_t gdlc_sap_corr;           /* GDLC SAP correlator     */
     ulong_t gdlc_ls_corr;            /* GDLC ls correlator      */
     ulong_t user_sap_corr;           /* user's SAP correlator   */
                                       /* — RETURNED             */
     ulong_t user_ls_corr;            /* user's link station     */
  /* corr — RETURNED */
     u_char_t ls_diag[DLC_MAX_DIAG];  /* the char name of the ls */
     ulong_t ls_state;                /* current ls state        */
     ulong_t ls_sub_state;            /* further clarification   */
                                       /* of state               */
     struct dlc_ls_counters counters;
     ulong_t protodd_len;             /* protocol dependent data */
                                       /* byte length            */
     };
```

gdlc_sap_corr GDLC SAP correlator: The GDLC SAP identifier of the target LS.

gdlc_ls_corr GDLC LS correlator: The GDLC LS identifier to be queried.

user_sap_corr User SAP correlator: The user's SAP identifier returned for routing purposes.

user_ls_corr User LS correlator: The user's LS identifier returned for routing purposes.

ls_diag Link station diagnostic tag: Contains the ASCII character string tag passed to GDLC at the DLC_START_LS ioctl operation to identify the station being queried. For example, SNA Services puts the attachment profile name in this field.

ls_state Current station state: Contains the current state of this LS:

```
#define DLC_OPENING     1     /* the SAP or link station is in the  */
                               /* process of opening                */
#define DLC_OPENED      2     /* the SAP or ls has been opened      */
#define DLC_CLOSING     3     /* the SAP or link station is in the */
                              /* process of closing                 */
#define DLC_INACTIVE    4     /* the link station is in an inactive */
                              /* state at present                   */
```

ls_sub_state Current station substate: Contains the current substate of this LS. Several indicators may be active concurrently.

```
#define DLC_CALLING      0x80000000 /* the ls is calling              */
#define DLC_LISTENING    0x40000000 /* the ls is listening            */
#define DLC_CONTACTED    0x20000000 /* the ls is contacted into */
                                    /* sequenced data mode            */
#define DLC_LOCAL_BUSY   0x10000000 /* the local link station is  */
                                    /* busy right now                 */
#define DLC_REMOTE_BUSY 0x08000000 /* the remote link station      */
                                    /* is busy right now              */
```

counters Link station reliability/availability/serviceability counters: These 14 reliability/availability/serviceability counters are shown as an example only. Each GDLC device manager provides as many of these counters as necessary to diagnose specific network problems for its protocol type.

```
struct dlc_ls_counters
      {
      ulong_t test_cmds_sent; /* number of test commands sent */
      ulong_t test_cmds_fail; /* number of test commands failed */
      ulong_t test_cmds_rec; /* num of test commands received */

      ulong_t data_pkt_sent; /* number of sequenced data */    /* pa
ckets sent */

      ulong_t data_pkt_resent; /* number of sequenced data        */
                               /* packets resent                  */
      ulong_t max_cont_resent; /* maximum number of contiguous    */
                               /* resendings                      */
      ulong_t data_pkt_rec;    /* data packets received           */
      ulong_t inv_pkt_rec;     /* num of invalid packets rcvd     */
      ulong_t adp_rec_err;     /* number of data detected         */
                               /* receive errors                  */
      ulong_t adp_send_err;    /* number of data_detected         */
                               /* transmit errors */
      ulong_t rec_inact_to;    /* number of received inactivity   */
                               /* timeouts                        */
```

```
        ulong_t cmd_polls_sent;  /* number of command polls sent    */
        ulong_t cmd_repolls_sent /* number of command repolls sent  */
        ulong_t cmd_cont_repolls;/* maximum number of continuous    */
                                 /* repolls sent */
        };
```

protodd_len Length of protocol dependent data: Contains the byte length of the following area.

Protocol Dependent Data
Optional: Contains any additional statistics that a particular GDLC device manager might provide. See the individual GDLC specifications for information on the specific fields returned. This data area must directly follow (or append to) the end of the **dlc_qls_arg** structure.

Implementation Specifics

This DLC_QUERY_LS ioctl operation for DLC is part of the *device manager* Data Link Control in BOS Extensions 2.

Insert the Standard Ethernet, SDLC, Token-Ring, IEEE Ethernet (802.3), or X.25 QLLC (or any combination) in place of *device manager* above, depending on which device manager you decide to use.

Related Information

Generic Data Link Control (GDLC) Environment Overview in *Communications Programming Concepts.*

DLC_ENTER_LBUSY ioctl Operation for DLC

The following parameter enters local busy mode on a particular link station (LS).

```
struct dlc_corr_arg
      {
      ulong_t gdlc_sap_corr;      /* GDLC SAP correlator          */
      ulong_t gdlc_ls_corr;       /* GDLC link station correlator */
      };
```

gdlc_sap_corr GDLC SAP correlator: The GDLC SAP identifier of the target LS.

gdlc_ls_corr GDLC LS correlator: The GDLC LS identifier to enter local busy mode.

Implementation Specifics

This DLC_ENTER_LBUSY ioctl operation for DLC is part of the *device manager* Data Link Control in BOS Extensions 2.

Insert the Standard Ethernet, SDLC, Token-Ring, IEEE Ethernet (802.3), or X.25 QLLC (or any combination) in place of *device manager* above, depending on which device manager you decide to use.

Related Information

Parameter Blocks by ioctl Operation for DLC.

Generic Data Link Control (GDLC) Environment Overview in *Communications Programming Concepts.*

DLC_EXIT_LBUSY ioctl Operation for DLC

The following parameter exits local busy mode on a particular link station (LS).

```
struct dlc_corr_arg
      {
      ulong_t gdlc_sap_corr;        /* GDLC SAP correlator          */
      ulong_t gdlc_ls_corr;         /* GDLC link station correlator */
      };
```

gdlc_sap_corr GDLC SAP correlator: The GDLC SAP identifier of the target LS.

gdlc_ls_corr GDLC LS correlator: The GDLC LS identifier to exit local busy mode.

Implementation Specifics

This DLC_EXIT_LBUSY ioctl operation for DLC is part of the *device manager* Data Link Control in BOS Extensions 2.

Insert the Standard Ethernet, SDLC, Token-Ring, IEEE Ethernet (802.3), or X.25 QLLC (or any combination) in place of *device manager* above, depending on which device manager you decide to use.

Related Information

Generic Data Link Control (GDLC) Environment Overview in *Communications Programming Concepts.*

DLC_ENTER_SHOLD ioctl Operation for DLC

The following parameter enters short hold mode on a particular link station (LS).

```
struct dlc_corr_arg
      {
      ulong_t gdlc_sap_corr;        /* GDLC SAP correlator          */
      ulong_t gdlc_ls_corr;         /* GDLC link station correlator */
      };
```

gdlc_sap_corr GDLC SAP correlator: The GDLC SAP identifier of the target LS.

gdlc_ls_corr GDLC LS correlator: The GDLC LS identifier to enter short hold mode.

Implementation Specifics

This DLC_ENTER_SHOLD ioctl operation for DLC is part of the *device manager* Data Link Control in BOS Extensions 2.

Insert the Standard Ethernet, SDLC, Token-Ring, IEEE Ethernet (802.3), or X.25 QLLC (or any combination) in place of *device manager* above, depending on which device manager you decide to use.

Related Information

Parameter Blocks by ioctl Operation for DLC.

Generic Data Link Control (GDLC) Environment Overview in *Communications Programming Concepts.*

DLC_EXIT_SHOLD ioctl Operation for DLC

The following parameter exits short hold mode on a particular link station (LS).

```
struct dlc_corr_arg
      {
      ulong_t gdlc_sap_corr;      /* GDLC SAP correlator         */
      ulong_t gdlc_ls_corr;       /* GDLC link station correlator */
      };
```

gdlc_sap_corr GDLC SAP correlator: The GDLC SAP identifier of the target LS.

gdlc_ls_corr GDLC LS correlator: The GDLC LS identifier to exit short hold mode.

Implementation Specifics

This DLC_EXIT_SHOLD ioctl operation for DLC is part of the *device manager* Data Link Control in BOS Extensions 2.

Insert the Standard Ethernet, SDLC, Token-Ring, IEEE Ethernet (802.3), or X.25 QLLC (or any combination) in place of *device manager* above, depending on which device manager you decide to use.

Related Information

Generic Data Link Control (GDLC) Environment Overview in *Communications Programming Concepts.*

DLC_GET_EXCEP ioctl Operation for DLC

The following parameter returns asynchronous exception notifications to the application user.

```
struct dlc_getx_arg
      {
      ulong_t user_sap_corr;         /* user SAP corr - RETURNED  */
      ulong_t user_ls_corr;          /* user ls corr - RETURNED   */
      ulong_t result_ind;            /* the flags identifying the */
                                      /* type of excep            */
      int result_code;               /* the manner of excep       */
      u_char_t result_ext[DLC_MAX_EXT]; /* excep specific ext     */
      };
```

user_sap_corr User service access point (SAP) correlator: The user's SAP identifier for this exception.

user_ls_corr User link station (LS) correlator: The user's LS identifier for this exception.

result_ind Result indicators:

```
#define DLC_TEST_RES    0x08000000  /* a test cmd completion       */
#define DLC_SAPE_RES    0x04000000  /* an enable SAP completion    */
#define DLC_SAPD_RES    0x02000000  /* a disable SAP completion    */
#define DLC_STAS_RES    0x01000000  /* a start ls completion       */
#define DLC_STAH_RES    0x00800000  /* a halt ls completion        */
#define DLC_DIAL_RES    0x00400000  /* manually dial the phone now */
#define DLC_IWOT_RES    0x00200000  /* inactivity without          */
                                     /* termination                 */
#define DLC_IEND_RES    0x00100000  /* the inactivity has ended    */
```

```
#define DLC_CONT_RES    0x00080000  /* the station is now          */
                                     /* contacted                   */
#define DLC_RADD_RES    0x00004000  /* the remote addr has changed */
#define DLC_MAX_EXT 48               /* max size of the result      */
                                     /* extension field             */
```

DLC_TEST_RES
Test complete: A *nonextended* result. Set to 1, this bit indicates that the link test has completed as indicated in the result code.

DLC_SAPE_RES
SAP enabled: An *extended* result. Set to 1, this bit indicates that the SAP is active and ready for LSs to be started. See DLC_SAPE_RES operation for the format of the extension area.

DLC_SAPD_RES
SAP Disabled: A *nonextended* result. Set to 1, this bit indicates that the SAP has been terminated as indicated in the result code.

DLC_STAS_RES
Link station started: An *extended* result. Set to 1, this bit indicates that the link station is connected to the remote station in asynchronous or normal disconnected mode. GDLC is waiting for link receive data from the device driver, or additional commands from the user such as the DLC_CONTACT ioctl operation. See DLC_STAS_RES operation for the format of the extension area.

DLC_STAH_RES
Link station halted: A *nonextended* result. Set to 1, this bit indicates that the LS has terminated due to a DLC_HALT_LS ioctl operation from the user, a remote discontact, or an error condition indicated in the result code.

DLC_DIAL_RES
Dial the phone: A *nonextended* result. Set to 1, this bit indicates that the user may now manually dial an outgoing call to the remote station.

DLC_IWOT_RES
Inactivity without termination: A *nonextended* result. Set to 1, this bit indicates that the LS protocol activity from the remote station has terminated for the length of time specified in the configuration (receive inactivity time out). The local station remains active and notifies the user if the remote station begins to respond. Additional notifications of inactivity without termination are suppressed until the inactivity condition clears up.

DLC_IEND_RES
Inactivity ended: A *nonextended* result. Set to 1, this bit indicates that the LS protocol activity from the remote station has restarted after a condition of inactivity without termination.

DLC_CONT_RES Contacted: A *nonextended* result. Set to 1, this bit indicates that GDLC has either received a Set Mode, or has received a positive response to a Set Mode initiated by the local LS. GDLC is now able to send and receive normal sequenced data on this LS.

DLC_RADD_RES Remote address/name change: An *extended* result. Set to 1, this bit indicates that the remote LS address (or name) has been changed from the previous value. This can occur on SDLC links when negotiating a point to point connection, for example. See the DLC_RADD_RES operation for the format of the extension area.

result_code Result code: The following values specify the result codes for GDLC. Negative return codes that are *even* indicate that the error condition can be remedied by restarting the LS returning the error. Return codes that are *odd* indicate that the error is catastrophic, and, at the minimum, the SAP must be restarted. Additional error data may be obtained from the GDLC error log and link trace entries.

```
#define DLC_SUCCESS        0   /* the result indicated was          */
                               /* successful                        */
#define DLC_PROT_ERR    -906   /* protocol error                    */
#define DLC_BAD_DATA    -908   /* a bad data compare on a TEST      */
#define DLC_NO_RBUF     -910   /* no remote buffering on test       */
#define DLC_RDISC       -912   /* remote initiated discontact       */
#define DLC_DISC_TO     -914   /* discontact abort timeout          */
#define DLC_INACT_TO    -916   /* inactivity timeout                */
#define DLC_MSESS_RE    -918   /* mid session reset                 */
#define DLC_NO_FIND     -920   /* cannot find the remote name       */
#define DLC_INV_RNAME   -922   /* invalid remote name               */
#define DLC_SESS_LIM    -924   /* session limit exceeded            */
#define DLC_LST_IN_PRGS -926   /* listen already in progress        */
#define DLC_LS_NT_COND  -928   /* ls unusual network condition      */
#define DLC_LS_ROUT     -930   /* link station resource outage      */
#define DLC_REMOTE_BUSY -932   /* remote station found, but busy    */
#define DLC_REMOTE_CONN -936   /* specified remote is already       */
                               /* connected */
#define DLC_NAME_IN_USE -901   /* local name already in use         */
#define DLC_INV_LNAME   -903   /* invalid local name                */
#define DLC_SAP_NT_COND -905   /* SAP network unusual network       */
                               /* condition                         */
#define DLC_SAP_ROUT    -907   /* SAP resource outage               */
#define DLC_USR_INTRF   -909   /* user interface error              */
#define DLC_ERR_CODE    -911   /* error in the code has been        */
                               /* detected                          */
#define DLC_SYS_ERR     -913   /* system error                      */
```

result_ext Result extension: Several results carry extension areas to provide additional information about them. The user must provide a full sized area for each result requested since there is no way to tell if the next result is extended or nonextended. The extended result areas are described by type below.

DLC_SAPE_RES — SAP Enabled Result Extension

The following parameter's service access point (SAP) enables a result extension.

```
struct dlc_sape_res
      {
      ulong_t max_net_send;              /* maximum write network */
                                          /* data length          */
      ulong_t lport_addr_len;            /* local port network    */
                                          /* address length       */
      u_char_t lport_addr[DLC_MAX_ADDR]; /* the local port        */
                                          /* address              */
      };
```

max_net_send Maximum write network data length: The maximum number of bytes that the user can write for each packet when writing network data. This is generally based on a communications mbuf/mbuf's page cluster size, but is not necessarily limited to a single mbuf/mbuf's since mbuf/mbuf's can be linked.

lport_addr_len Local port net address length: Contains the byte length of the local port network address.

lport_addr Local port network address: Contains the hexadecimal value of the local port network address.

DLC_STAS_RES — Link Station Started Result Extension

The following parameter starts a link station's (LS) result extension.

```
struct dlc_stas_res
      {
      ulong_t maxif;                     /* max size of the data sent  */
                                          /* on a write                 */
 ulong_t rport_addr_len;                 /* remote port network        */
                                          /* address length             */
 u_char_t rport_addr[DLC_MAX_ADDR];      /* remote port address        */
 ulong_t rname_len;                      /* remote network name length */
 u_char_t rname[DLC_MAX_NAME];           /* remote network name        */
 uchar_t res[3];                         /* reserved                   */
 uchar_t rsap;                           /* remote SAP                 */
      ulong_t max_data_off;             /* the maximum data offsets    */
                                         /* for sends                   */
      };
```

maxif Maximum I-field size: Contains the maximum byte size allowable for user data. This value is derived from the value supplied by the user at start link station (DLC_START_LS) and the actual number of bytes that can be handled by the GDLC and device handler on a single transmit or receive. Generally this value is something less than the size of a communications mbuf page cluster. However, some communications devices may be able to link page clusters together, so the maximum I-field receivable may be even greater than the length of a single mbuf. The returned value will never exceed the value supplied by the user, but may be smaller if buffering is not large enough to hold the specified value.

rport_addr_len Remote port network address length: Contains the byte length of the remote port network address.

rport_addr Remote port network address: Contains the hexadecimal value of the remote port network address.

rname_len Remote network name length: Contains the byte length of the remote port network name. This is returned only when name discovery procedures are used to locate the remote station. Otherwise this field is set to zero. Network names can be 1 to 20 characters in length.

rname Remote network name: Contains the name being used by the remote SAP. This field is valid only if name-discovery procedures were used to locate the remote station.

rsap Remote SAP: Contains the hexadecimal value of the remote SAP address.

max_data_off Write data offset: Contains the data offset in bytes of a communications mbuf where transmit data must minimally begin. This allows ample room for the DLC and MAC headers to be inserted if needed. Some DLC's may be able to prepend additional mbufs for their headers, and will set this field to zero.

This field is only valid for kernel users that pass in a communications mbuf on write operations.

Note: In order to align the data moves to a particular byte boundary, the kernel user may wish to choose a value larger than the minimum value returned

DLC_STAH_RES — Link Station Halted Result Extension

The following parameter halts the link station (LS) result extension.

```
struct dlc_stah_res
      {
      ulong conf_ls_corr;          /* conflicting link station corr */
      };
```

This extension is valid only if the result code value indicates –936 (specified remote is already connected).

conf_ls_corr Conflicting link station correlator: Contains the user's link station identifier that already has the specified remote station attached.

DLC_RADD_RES — Remote Address/Name Change Result Extension

The following parameter changes the remote address or name of the result extension.

```
struct dlc_radd_res
      {
      ulong rname_len;           /* remote network name/addr length */
      u_char rname[DLC_MAX_NAME]; /* remote network name/addr     */
      };
```

rname_len Remote network address or name length: Contains the byte length of the updated remote service access point (SAP)'s network address or name.

rname Remote network address or name: Contains the updated address or name being used by the remote SAP.

Implementation Specifics

This DLC_GET_EXCEP ioctl operation for DLC is part of the *device manager* Data Link Control in BOS Extensions 2.

Insert the Standard Ethernet, SDLC, Token-Ring, IEEE Ethernet (802.3), or X.25 QLLC (or any combination) in place of *device manager* above, depending on which device manager you decide to use.

Related Information

Generic Data Link Control (GDLC) Environment Overview in *Communications Programming Concepts.*

DLC_ADD_GRP ioctl Operation for DLC

The following parameter adds a group or multicast receive address.

```
struct dlc_add_grp
{
        ulong_t gdlc_sap_corr;            /* GDLC SAP correlator  */
        ulong_t grp_addr_len;             /* group address length */
        uchar_t grp_addr[DLC_MAX_ADDR];   /* grp addr to be added */
};
```

gdlc_sap_corr GDLC SAP Correlator: This is GDLC's SAP identifier being requested to add a group or multicast address to a port.

grp_addr_len Group Address Length: Contains the byte length of the group or multicast address to be added.

grp_addr Group Address: Contains the group or multicast address value to be added.

Implementation Specifics

This DLC_ADD_GRP ioctl operation for DLC is part of the *device manager* Data Link Control in BOS Extensions 2.

Insert the Standard Ethernet, SDLC, Token-Ring, IEEE Ethernet (802.3), or X.25 QLLC (or any combination) in place of *device manager* above, depending on which device manager you decide to use.

Related Information

Generic Data Link Control (GDLC) Environment Overview in *Communications Programming Concepts.*

IOCINFO ioctl Operation for DLC

Returns a structure that describes the device (refer to the description of the **sys/devinfo.h** file. The first byte is set to an **ioctype** of DD_DLC. The subtype and data are defined by the individual DLC devices. See the **/usr/include/sys/devinfo.h** file for details.

Implementation Specifics

This IOCINFO ioctl operation for DLC is part of the *device manager* Data Link Control in BOS Extensions 2.

Insert the Standard Ethernet, SDLC, Token-Ring, IEEE Ethernet (802.3), or X.25 QLLC (or any combination) in place of *device manager* above, depending on which device manager you decide to use.

Related Information

Generic Data Link Control (GDLC) Environment Overview in *Communications Programming Concepts*.

open, openx Subroutine Interface for Data Link Control (DLC) Devices

Purpose

Opens the GDLC device manager by special file name.

Syntax

#include <sys/fcntl.h>
#include <sys/gdlextcb.h>

int open(*path*, *oflag*, *mode***)**

or

int openx(*path*, *oflag*, *mode*, *ext***)**
char **path*;
int *oflag*;
int *mode*;
int *ext*;

Description

The open subroutine allows the application user to open a generic data link control (GDLC) device manager by specifying the DLC's special file name and the target device handler's special file name. Since the GDLC device manager is multiplexed, more than one process can open it (or the same process many times) and still have unique channel identifications.

Each open carries the communications device handler's special file name so that the DLC knows on which port to transfer data. This name must directly follow the DLC's special file name. For example, in the `/dev/dlcether/ent0` character string, `ent0` is the special file name of the Ethernet device handler. GDLC obtains this name using its **dlcmpx** routine.

Parameters

path — Consists of a character string containing the **/dev** special file name of the GDLC device manager, with the name of the communications device handler appended, as follows:

`/dev/dlcether/ent0`

oflag — Specifies a value for the file status flag. The GDLC device manager ignores all but the following flags:

- **O_RDWR** — Open for reading and writing. This must be set for GDLC or the open will fail.
- **O_NDELAY, O_NONBLOCK** — Subsequent reads with no data present and writes that cannot get enough resources will return immediately. The calling process is not put to sleep.

mode — Specifies the O_CREAT mode parameter. This is ignored by GDLC.

ext	Specifies the extended subroutine parameter. This is a pointer to the **dlc_open_ext** extended I/O structure for the **open** subroutines. DLC Extended Parameters for **open** Subroutine provides more information on this parameter.

Return Values

Upon successful completion, the **open** subroutine returns a valid file descriptor that identifies the opened GDLC channel.

If an error occurs, a value of –1 is returned with one of the following error numbers available using **errno**, as defined in the **errno.h** header file:

ECHILD	Cannot create a kernel process.
EINVAL	Invalid value.
ENODEV	No such device handler.
ENOMEM	Not enough resources to satisfy the **open** subroutine.
EFAULT	Kernel service, such as **copyin** or **initp**, has failed.

Implementation Specifics

This **open** subroutine interface is part of the *device manager* Data Link Control in BOS Extensions 2.

Insert the Standard Ethernet, SDLC, Token-Ring, IEEE Etherent (802.3), or X.25 QLLC (or any combination) in place of *device manager* above, depending on which device manager you decide to use.

Related Information

The **dlcmpx** routine.

The **copyin** kernel service, **initp** kernel service.

open, openx Subroutine, Extended Parameters.

close Subroutine Interface for Data Link Control (DLC) Devices.

Generic Data Link Control (GDLC) Environment Overview in *Communications Programming Concepts*.

open, openx Subroutine, Extended Parameters

Description

An extended **open** (**openx**) subroutine may be issued to alter certain normally defaulted parameters, such as maximum service access points (SAPs) and ring queue depths. Kernel users may change these normally defaulted parameters, but are required to provide additional parameters to notify the **dlcopen** routine that these callers are to be treated as kernel processes and not as application processes. Additional parameters passed include functional addresses that the user wishes GDLC to call for notification of asynchronous events, such as receive data available.

The structure for the **open** subroutine extension parameters is as follows:

```
struct dlc_open_ext
{
      ulong_t maxsaps;  /* 1 (1 to 127) service access points      */
      int (*rcvi_fa)(); /* receive I–frame function address        */
      int (*rcvx_fa)(); /* receive XID function address            */
      int (*rcvd_fa)(); /* receive Datagram function address       */
      int (*rcvn_fa)(); /* receive Network data function address   */
      int (*excp_fa)(); /* exception handler function address      */
};
```

See the **/include/sys/gdlextcb.h** file for more details on GDLC structures.

The first parameter is optional for both the application and the kernel user. If the default value is desired, the field must be set to zero by the user prior to issuing the **open** subroutine.

maxsaps Maximum SAPs: The maximum number of SAPs that this user channel is going to start and have running concurrently. The default is 1. Any value from 1 to 127 can be specified (0 gets the default).

The last five parameters are mandatory for kernel users but are ignored by GDLC for application users. There are no default values. Each field must be filled in by the kernel user. All functional entry addresses must be valid. That is, entry points that the kernel user does not wish to support must at least point to a routine that frees the communication's memory buffer (**mbuf**) passed on the call.

***rcvi_fa** Receive I-Frame Data Function Pointer: The address of a user routine that handles the sequenced I-frame receive data completions. This field is valid for kernel users only and must be set to 0 (zero) by application users.

***rcvx_fa** Receive XID Function Pointer: The address of a user routine that handles the exchange ID receive data completions.

***rcvd_fa** Receive Datagram Function Pointer: The address of a user routine that handles the datagram receive data completions.

***rcvn_fa** Receive Network Data Function Pointer: The address of a user routine that handles the network receive data completions.

***excp_fa** Exception Handler Function Pointer: The address of a user routine that handles the exception conditions, such as DLC_SAPE_RES (SAP Enabled) or DLC_CONT_RES (LS contacted).

Implementation Specifics

These DLC extended parameters for **open** subroutine are part of the *device manager* Data Link Control in BOS Extensions 2.

Insert the Standard Ethernet, SDLC, Token-Ring, IEEE Etherent (802.3), or X.25 QLLC (or any combination) in place of *device manager* above, depending on which device manager you decide to use.

Related Information

The **open**, **openx** subroutine.

The **dlcopen** entry point routine.

Parameter Blocks by ioctl Operation for DLC

Generic Data Link Control (GDLC) Environment Overview in *Communications Programming Concepts.*

Datagram Data Received Routine, for DLC

Function

This routine is coded by the kernel user and called by GDLC each time a datagram packet is received for the kernel user.

Subroutine Call

#include <sys/gdlextcb.h>
int (*dlc_open_ext.rcvd_fa)(*m*, *ext***)**
struct mbuf **m*;
struct dlc_io_ext **ext*;

Parameters

m	Specifies the pointer to a communications memory buffer (**mbuf**).
ext	Specifies the receive extension parameter. This is a pointer to the **dlc_io_ext** extended I/O structure for reads.

Returns to GDLC

int Indicates one of the following return codes from this function call:

DLC_FUNC_OK	The received datagram **mbuf** data has been accepted.
DLC_FUNC_RETRY	The received datagram **mbuf** data cannot be accepted at this time. GDLC should retry this function later. The actual retry wait period depends on the DLC in use. Excessive retries may close the link station.

Implementation Specifics

This DLC datagram data received routine is part of the *device manager* Data Link Control in BOS Extensions 2.

Insert the Standard Ethernet, SDLC, Token-Ring, IEEE Etherent (802.3), or X.25 QLLC (or any combination) in place of *device manager* above, depending on which device manager you decide to use.

Related Information

DLC Extended Parameters for read Subroutine.

Generic Data Link Control (GDLC) Environment Overview in *Communications Programming Concepts.*

Exception Condition Routine

Function

This routine is coded by the kernel user and called by GDLC each time an asynchronous event occurs that must notify the kernel user, such as DLC_SAPD_RES (SAP disabled) or DLC_CONT_RES (contacted).

Subroutine Call

#include <sys/gdlextcb.h>
int (*dlc_open_ext.excp_fa)(*ext***)**
struct dlc_getx_arg ****ext*;**

Parameter

ext Specifies the same structure for a **dlc_getx_arg** (get exception) **ioctl** subroutine.

Returns to GDLC

int Indicates the following return code from the function call:

DLC_FUNC_OK The exception has been accepted.

Note: The function call above has a hidden parameter extension for internal use only, defined as **int** **chanp*, the channel pointer.

Implementation Specifics

This DLC exception condition routine is part of the *device manager* Data Link Control in BOS Extensions 2.

Insert the Standard Ethernet, SDLC, Token-Ring, IEEE Etherent (802.3), or X.25 QLLC (or any combination) in place of *device manager* above, depending on which device manager you decide to use.

Related Information

The **ioctl** subroutine.

Parameter Blocks by ioctl Operation for DLC.

Generic Data Link Control (GDLC) Environment Overview in *Communications Programming Concepts.*

I–Frame Data Received Routine

Function

This routine is coded by the kernel user and called by GDLC each time a normal sequenced data packet is received for the kernel user.

Subroutine Call

#include <sys/gdlextcb.h>

int (*dlc_open_ext.rcvi_fa)(*m*, *ext***)**
struct mbuf **m*;
struct dlc_io_ext **ext*;

Parameters

m	Specifies the pointer to a communications memory buffer (**mbuf**).
ext	Specifies the receive extension parameter. This is a pointer to the **dlc_io_ext** extended I/O structure for reads. The argument to this parameter must be in the kernel space.

Returns to GDLC

int Indicates one of the following return codes from the function call:

DLC_FUNC_OK	The received **I-frame** function call is accepted.
DLC_FUNC_BUSY	The received **I-frame** function call cannot be accepted at this time. The **ioctl** command operation DLC_EXIT_LBUSY must be issued later using the **ioctl** subroutine.
DLC_FUNC_RETRY	The received **I-frame** function call cannot be accepted at this time. GDLC should retry this function call later. The actual retry wait period depends on the DLC in use. Excessive retries can be subject to a halt of the link station.

Implementation Specifics

This DLC **I-frame** data received routine is part of the *device manager* Data Link Control in BOS Extensions 2.

Insert the Standard Ethernet, SDLC, Token-Ring, IEEE Etherent (802.3), or X.25 QLLC (or any combination) in place of *device manager* above, depending on which device manager you decide to use.

Related Information

The **ioctl** subroutine.

Parameter Blocks by ioctl Operation for DLC.

DLC Extended Parameters for read Subroutine.

Generic Data Link Control (GDLC) Environment Overview in *Communications Programming Concepts.*

Network Data Received Routine

Function

This routine is coded by the kernel user and called by GDLC each time network-specific data is received for the kernel user.

Subroutine Call

#include <sys/gdlextcb.h>

int (*dlc_open_ext.rcvn_fa)(*m*, *ext***)**
struct mbuf **m*;
struct dlc_io_ext **ext*;

Parameters

m	Specifies the pointer to a communications memory buffer (**mbuf**).
ext	Specifies the receive extension parameter. This is a pointer to the **dlc_io_ext** extended I/O structure for reads.

Returns to GDLC

int Indicates one of the following return codes from this function call:

DLC_FUNC_OK	The received network **mbuf** data has been accepted.
DLC_FUNC_RETRY	The received network **mbuf** data cannot be accepted at this time. GDLC should retry this function call some time later. The actual retry wait period depends on the DLC in use, and excessive retries can cause a disabling of the service access point.

Implementation Specifics

This DLC network data received routine is part of the *device manager* Data Link Control in BOS Extensions 2.

Insert the Standard Ethernet, SDLC, Token-Ring, IEEE Etherent (802.3), or X.25 QLLC (or any combination) in place of *device manager* above, depending on which device manager you decide to use.

Related Information

DLC Extended Parameters for read Subroutine.

Generic Data Link Control (GDLC) Environment Overview in *Communications Programming Concepts.*

XID Data Received Routine

Function

This routine is coded by the kernel user and called by GDLC each time an exchange identification (XID) packet is received for the kernel user.

Subroutine Call

#include <sys/gdlextcb.h>

int (*dlc_open_ext.rcvx_fa)(*m*, *ext***)**
**struct mbuf ***m*;
**struct dlc_io_ext ***ext*;

Parameters

m — Specifies the pointer to a communication memory buffer (**mbuf**).

ext — Specifies the receive extension parameter. This is a pointer to the **dlc_io_ext** extended I/O structure for reads. The argument to this parameter must be in the kernel space.

Returns to GDLC

int — Indicates one of the following return codes from this function call:

DLC_FUNC_OK — The received XID **mbuf** data has been accepted.

DLC_FUNC_RETRY — The received XID **mbuf** data cannot be accepted at this time. GDLC should retry this function call some time later. The actual retry wait period depends on the DLC in use. Excessive retries may close the link station.

Implementation Specifics

This DLC XID data received routine is part of the *device manager* Data Link Control in BOS Extensions 2.

Insert the Standard Ethernet, SDLC, Token-Ring, IEEE Etherent (802.3), or X.25 QLLC (or any combination) in place of *device manager* above, depending on which device manager you decide to use.

Related Information

DLC Extended Parameters for read Subroutine.

Generic Data Link Control (GDLC) Environment Overview in *Communications Programming Concepts.*

read, readx Subroutine, Extended Parameters

Description

An extended **read** (**readx**) subroutine must be issued by an application user to provide GDLC with a structure to return the type of data and the service access point (SAP) and link station (LS) correlators.

The structure for the **read** subroutine extension parameters is as follows:

```
struct dlc_io_ext
{
      ulong_t sap_corr;           /* Sap correlator
*/
      ulong_t ls_corr;            /* Link Station correlator
*/
      ulong_t flags;              /* flags
*/
      ulong_t dlh_len;            /* data link header length
*/
};
```

sap_corr User SAP Correlator: The user's SAP identifier of the received data.

ls_corr User LS Correlator: The user's LS identifier of the received data.

flags Result Flags: The following flags are supported:

```
#define DLC_INFO          0x80000000      /* normal I-frame
*/
#define DLC_XIDD          0x40000000      /* XID data
*/
#define DLC_DGRM          0x20000000      /* datagram
*/
#define DLC_NETD          0x10000000      /* network data
*/
#define DLC_OFLO          0x00000002      /* receive overflow occurr
ed    */
#define DLC_RSPP          0x00000001      /* response pending
*/
```

DLC_INFO I-Frame Data Received: Indicates that normal sequenced data has been received for a link station. If buffer overflow (OFLO) is indicated, the received data has been truncated because the received data length exceeds either the maximum I-field size derived at completion of DLC_START_LS ioctl operation or the application user's buffer size.

DLC_XIDD XID Data Received: Indicates that exchange identification (XID) data has been received for a link station. If buffer overflow (OFLO) is indicated, the received XID has been truncated because the received data length exceeds either the maximum I-field size derived at DLC_START_LS completion or the application user's buffer size. If response pending (RSPP) is indicated, an XID response is required and must be provided to GDLC using a write XID as soon as possible to avoid repolling and possible termination of the remote LS.

DLC_DGRM
Datagram Data Received: Indicates that a datagram has been received for an LS. If buffer overflow (OFLO) is indicated, the received data has been truncated because the received data length exceeds either the maximum I-field size derived at DLC_START_LS completion or the application user's buffer size.

DLC_NETD
Network Data: Indicates that data has been received from the network for a service access point. This may be link-establishment data such as X.21 call-progress signals or smart modem command responses. It can also be data destined for the user's SAP when no link station has been started that fits the addressing of the packet received. If buffer overflow (OFLO) is indicated, the received data has been truncated because the received data length exceeds either the maximum packet size derived at DLC_ENABLE_SAP completion or the application user's buffer size.

Network data contains the entire MAC layer packet, excluding any fields stripped by the adapter such as Preamble or CRC.

DLC_OFLO
Buffer Overflow: Indicates that overflow of the user data area has occurred and the data was truncated. This error does not set a **u.u_error** indication.

DLC_RSPP
Response Pending: This bit indicates that the XID received requires an XID response to be sent back to the remote link station.

dlh_len
Data Link Header Length: This field has different meaning depending on whether the extension is for a **readx** subroutine call *to* GDLC or a response *from* GDLC.

On the application **readx** subroutine it indicates whether the user wishes to have datalink header information prefixed to the data. If this field is set to 0 (zero), the data link header is *not* to be copied (only the I-field is copied). If this field is set to any nonzero value, the data link header information will be included in the read.

On the response to an application **readx** subroutine this field contains the number of data link header bytes received and copied into the Data Link Header Information field.

On asynchronous receive function handlers to the kernel user, this field contains the length of the data link header within the communications memory buffer (**mbuf**).

Implementation Specifics

These DLC extended parameters for **read** subroutine are part of the *device manager* Data Link Control in BOS Extensions 2.

Insert the Standard Ethernet, SDLC, Token-Ring, IEEE Etherent (802.3), or X.25 QLLC (or any combination) in place of *device manager* above, depending on which device manager you decide to use.

Related Information

The **read**, **readx**, **read**v, or **readvx** Subroutine.

DLC Extended Parameters for write Subroutine

Parameter Blocks by ioctl Operation for DLC

Generic Data Link Control (GDLC) Environment Overview in *Communications Programming Concepts.*

readx Subroutine Interface for Data Link Control (dlc) Devices

Purpose

Allows receive application data to be read using a file descriptor.

Syntax

#include <sys/gdlextcb.h>
#include <sys/uio.h>

int readx (*fildes*, *buf*, *len*, *ext*)
int fildes;
char *buf;
int len;
int ext;

Description

The receive queue for this application user is interrogated for any pending data. The oldest data packet is copied to user space, with the type of data, the link station correlator, and the service access point (SAP) correlator written to the extension area. When attempting to read an empty receive data queue, the default action is to delay until data is available. If the **O_NDELAY** or **O_NONBLOCK** flags are specified in the **open** subroutine, the **readx** subroutine returns immediately to the caller.

Data is transferred using the **uiomove** kernel service between the user space and kernel communications memory buffers (**mbufs**). A complete receive packet must fit into the user's read data area. GDLC does not break up received packets into multiple user data areas.

Parameters

fildes	Specifies the file descriptor returned from the **open** subroutine.
buf	Points to the user data area.
len	Contains the byte count of the user data area.
ext	Specifies the extended subroutine parameter. This is a pointer to the **dlc_io_ext** extended I/O structure for the **readx** subroutine. DLC Extended Parameters for **read** Subroutine provides more information on this parameter.

Note: It is the user's responsibility to set the *ext* parameter area to 0 (zero) prior to issuing the **readx** subroutine to insure valid entries when no data is available.

Return Values

Upon successful completion, the **readx** subroutine returns the number of bytes read and placed into the application data area. If more data is received from the media than will fit into the application data area, the **DLC_OFLO** flag is set in the **dlc_io_ext** command extension area to indicate that the read is truncated. All excess data is lost.

If no data is available and the application user has specified the **O_NDELAY** or **O_NONBLOCK** flags at open time, a zero is returned.

If an error occurs, a value of –1 is returned with one of the following error numbers available using **errno**, as defined in the **errno.h** header file:

EBADF Bad file number.

EINTR A signal interrupted the subroutine before it received data.

EINVAL Invalid value.

ENOMEM Not enough resources to satisfy the read.

Implementation Specifics

This **readx** subroutine interface is part of the *device manager* Data Link Control in BOS Extensions 2.

Insert the Standard Ethernet, SDLC, Token-Ring, IEEE Etherent (802.3), or X.25 QLLC (or any combination) in place of *device manager* above, depending on which device manager you decide to use.

Related Information

The **readx**subroutine, **open** subroutine.

The **uiomove** kernel service.

read, readx Subroutine, Extended Parameters .

writex Subroutine Interface for Data Link.

select Subroutine Interface for Data Link Control (dlc) Devices

Purpose

Allows data to be sent using a file descriptor.

Syntax

#include <sys/select.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 descriptor and message queues to see if they are ready for reading (receiving) or writing (sending), or if they have an exception condition pending.

Note: GDLC does not support transmit for nonblocked notification in the full sense. If the *writelist* parameter is specified in the **select** call, GDLC always returns as if transmit is available. There is no checking to see if internal buffering is available or if internal control-block locks are free. These resources are much too dynamic, and tests for their availability can only be done reasonably at the time of use.

The *readlist* and *exceptlist* parameters are fully supported. Whenever the selection criteria specified by the *SelType* parameter is true, the file system 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 a value of 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. If the selection is not satisfied, the calling process is put to sleep waiting on a **selwakeup** subroutine at a later time.

Parameters

nfdsmsgs Specifies the number of file descriptors and message queues to check.

sellist The *readlist*, *writelist*, and *exceptlist* parameters specify what to check for during reading, writing, and exceptions, respectively. Each **sellist** is a structure that contains a file descriptor bit mask (**fdsmask**) and message queue identifiers (**msgids**).

The *writelist* criterion is always set true by GDLC.

timeout Points to a structure that specifies the maximum length of time to wait for at least one of the selection criteria to be met (if the *timeout* parameter is not a null pointer).

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 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 and exception criteria.

If the time limit specified by the *timeout* parameter expires, then the **select** subroutine returns a value of 0.

If an error occurs, a value of –1 is returned with one of the following error numbers available using **errno**, as defined in the **errno.h** header file:

EBADF Bad file number.

EINTR A signal interrupted the subroutine before it found any of the selected. events.

EINVAL One of the parameters contained an invalid value.

Implementation Specifics

This **select** subroutine interface is part of the *device manager* Data Link Control in BOS Extensions 2.

Insert the Standard Ethernet, SDLC, Token-Ring, IEEE Etherent (802.3), or X.25 QLLC (or any combination) in place of *device manager* above, depending on which device manager you decide to use.

Related Information

The **select** subroutine.

Generic Data Link Control (GDLC) Environment Overview in *Communications Programming Concepts.*

write, writex Subroutine, Extended Parameters

Purpose

An extended **write** (**writex**) subroutine must be issued by an application or kernel user to provide GDLC with the type of data and the service access point (SAP) and link station (LS) correlators. The structure for the **write** subroutine extension parameters is shown below:

```
        ulong_t sap_corr;           /* Sap correlator
*/
        ulong_t ls_corr;            /* Link Station correlator
*/
        ulong_t flags;              /* flags
*/
        ulong_t dlh_len;            /* <<< not used for writes >>>
*/
};
```

sap_corr GDLC SAP Correlator: The user's SAP identifier of the received data.

ls_corr GDLC Link Station Correlator: The user's link station identifier of the received data.

flags Write Flags: The following flags are supported:

```
/*** Read and Write Flags ***/
#define DLC_INFO            0x80000000      /* normal I-frame
*/
#define DLC_XIDD            0x40000000      /* XID data
*/
#define DLC_DGRM            0x20000000      /* datagram
*/
#define DLC_NETD            0x10000000      /* network data
*/
```

DLC_INFO Write I-Frame Data: Requests a sequenced data class of information to be sent (generally called I-frames).

This request is valid any time the target link station has been started and contacted.

DLC_XIDD Write XID Data: Requests an exchange identification (XID) or response to be sent.

This request is valid any time the target link station has been started with the following rules:

GDLC sends the XID as a command as long as no DLC_TEST, DLC_CONTACT, DLC_HALT_LS, or DLC_XIDD **write** subroutine is already in progress, and no received XID is waiting for a response. If a received XID is waiting for a response, GDLC automatically sends the write XID as that response. If no response is pending and a command is already in progress, the write is rejected by GDLC.

DLC_DGRM Write Datagram: Requests an unnumbered datagram to be sent.

This request is valid any time the target link station has been started.

DLC_NETD Write Network Data: Requests that network data be sent.

Examples of network data include special modem control data or user-generated medium access control (MAC) and logical link control (LLC) headers.

Network data must contain the entire MAC layer packet headers so that the packet can be sent without the data link control (DLC)'s intervention. GDLC only provides a pass-through function for this type of write.

This request is valid any time the SAP is open.

Implementation Specifics

These DLC extended parameters for **write** subroutine are part of the *device manager* Data Link Control in BOS Extensions 2.

Insert the Standard Ethernet, SDLC, Token-Ring, IEEE Etherent (802.3), or X.25 QLLC (or any combination) in place of *device manager* above, depending on which device manager you decide to use.

Related Information

The **write, writex** subroutine.

DLC Extended Parameters for read Subroutine.

Generic Data Link Control (GDLC) Environment Overview in *Communications Programming Concepts.*

writex Subroutine Interface for Data Link Control (dlc) Devices

Purpose

Allows application data to be sent using a file descriptor.

Syntax

#include <sys/gdlextcb.h>
#include <sys/uio.h>

int writex (*fildes*, *buf*, *len*, *ext*)
char **buf*;
int *ext*;
int *fildes*, *len*;

Description

Four types of data can be sent to GDLC. Network data can be sent to a service access point (SAP), while normal, Exchange Identification (XID), or datagram data can be sent to a link station (LS). Data is transferred using the **uiomove** subroutine between the application user space and kernel communications I/O buffers (**mbufs**). All data must fit into a single packet for each **write** subroutine. The generic data link control (GDLC) does not separate the user's write data area into multiple transmit packets. A maximum write data size is passed back to the user at DLC_ENABLE_SAP completion and at DLC_START_LS completion for this purpose. See DLC_SAPE_RES and DLC_STAS_RES for further information.

Normally, GDLC can immediately satisfy a **write** subroutine by completing the data link headers and sending the transmit packet down to the device handler. In some cases, however, transmit packets can be blocked by the particular protocol's flow control or by a resource outage. GDLC reacts to this differently based on the systems blocked or nonblocked file status flags. These are set for each channel using the O_NDELAY and O_NONBLOCK values passed on **open** subroutines or on **fcntl** subroutines with the **F_SETFD** parameter.

GDLC only looks at the **uio_fmode** on each **write** subroutine to determine whether the operation is blocked or nonblocked. Nonblocked writes that cannot get enough resources to queue the data return an error indication. Blocked **write** subroutines put the calling process to sleep until the resources free up or an error occurs.

Note: GDLC does not support nonblocked transmit users based on resource availability using the **selwakeup** subroutine. Internal resources such as communications I/O buffers and control block locks are very dynamic. Any **write** subroutines that fail with errors (such as EAGAIN or ENOMEM) should be retried at the users discretion.

Parameters

fildes — Specifies the file descriptor returned from the **open** subroutine.

buf — Points to the user data area.

len — Contains the byte count of the user data area.

ext — Specifies the extended subroutine parameter. This is a pointer to the **dlc_io_ext** extended I/O structure for the **writex** subroutine. DLC Extended

Parameters for **write** subroutine provides more information on this parameter.

Return Values

Upon successful completion, this service returns the number of bytes that were written into a communications packet from the user data area.

If an error occurs, a value of –1 is returned with one of the following error numbers available using **errno**, as defined in the **errno.h** header file.

EAGAIN Not enough resources to satisfy the write; for example, unable to obtain a necessary lock. The user can try again later.

EBADF Bad file number.

EINVAL Invalid value, such as too much data for a single packet.

EIO An I/O error has occurred, such as loss of the port.

ENOMEM Not enough resources to satisfy the write; for example, a lack of communications memory buffers (**mbufs**). The user can try again later.

Implementation Specifics

This **writex** subroutine interface is part of the *device manager* Data Link Control in BOS Extensions 2.

Insert the Standard Ethernet, SDLC, Token-Ring, IEEE Etherent (802.3), or X.25 QLLC (or any combination) in place of *device manager* above, depending on which device manager you decide to use.

Related Information

The **writex** subroutine, **uiomove** subroutine, **fcntl** subroutine, **open** subroutine.

DLC Extended Parameters for write Subroutine

readx Subroutine Interface for Data Link Control (dlc) Devices

Parameter Blocks by ioctl Operation for DLC

Generic Data Link Control (GDLC) Environment Overview in *Communications Programming Concepts.*

Network Computing System (NCS)

lb_$lookup_interface Library Routine (NCS)

Purpose

Looks up information about an interface in the GLB database.

Syntax

void lb_$lookup_interface (*object_interface*, *lookup_handle*, *max_results*, *num_results*, *results*, *status*)
uuid_$t **object_interface*;
lb_$lookup_handle_t **lookup_handle*;
unsigned long *max_results*;
unsigned long **num_results*;
lb_$entry_t *results* **[ ]**;
status_$t **status*;

Parameters

Input

object_interface	Points to the UUID of the interface being looked up.
max_results	Specifies the maximum number of matching entries that can be returned by a single call. This should be the number of elements in the *results* parameter array.

Input/Output

lookup_handle	Specifies a location in the database. On input, the *lookup_handle* value indicates the location in the database where the search begins. An input value of **lb_$default_lookup_handle** specifies that the search starts at the beginning of the database. On return, the *lookup_handle* parameter indicates the next unsearched part of the database (that is, the point at which the next search should begin). A return value of **lb_$default_lookup_handle** indicates that the search reached the end of the database. Any other value indicates that the search found at most the number of matching entries specified by the *max_results* parameter before it reached the end of the database.

Output

num_results	Points to the number of entries that are returned in the *results* parameter array.
results	Specifies the array that contains the matching GLB database entries, up to the number specified in the *max_results* parameter. If the array contains any entries for servers on the local network, those entries appear first.
status	Points to the completion status.

Description

The **lb_$lookup_interface** routine returns GLB database entries whose **object_interface** fields match the specified interface. It returns information about all replicas of all objects that can be accessed through that interface.

The **lb_$lookup_interface** routine cannot return more than the number of matching entries specified by the *max_results* parameter at one time. The *lookup_handle* parameter directs this routine to do sequential lookup calls to find all matching entries.

Notes:

1. The Location Broker does not prevent modification of the database between lookup calls, which can cause the locations of entries relative to a *lookup_handle* value to change. If multiple calls are made to find all matching results in the database, the returned information may skip or duplicate entries from the database.
2. It is also possible for the results of a single lookup call to skip or duplicate entries. This can occur if the size of the results exceeds the size of an RPC packet (64K bytes).

Example

1. To look up information in the GLB database about a matrix multiplication interface, use the following:

```
lb_$lookup_interface (&matrix_if_id, &lookup_handle,
   results_array_size, &num_results,
   &matrix_if_results_array, &status);
```

Implementation Specifics

This Library Routine is part of Network Computing System in Network Support Facilities in Base Operating System (BOS) Runtime.

lb_$lookup_object Library Routine (NCS)

Purpose

Looks up information about an object in the GLB database.

Syntax

void lb_$lookup_object (*object*, *lookup_handle*, *max_results*, *num_results*, *results*, *status*)
uuid_$t **object*;
lb_$lookup_handle_t **lookup_handle*;
unsigned long *max_results*;
unsigned long **num_results*;
lb_$entry_t *results* **[ ];**
status_$t **status*;

Parameters

Input

object

Points to the UUID of the object being looked up.

max_results

Specifies the maximum number of matching entries that can be returned by a single call. This should be the number of elements in the *results* parameter array.

Input/Output

lookup_handle

Specifies a location in the database. On input, the value of the *lookup_handle* parameter indicates the location in the database where the search begins. An input value of **lb_$default_lookup_handle** specifies that the search starts at the beginning of the database.

On return, the *lookup_handle* parameter indicates the next unsearched part of the database (that is, the point at which the next search should begin). A return value of **lb_$default_lookup_handle** indicates that the search reached the end of the database. Any other value indicates that the search found at most the number of matching entries specified by the *max_results* parameter before it reached the end of the database.

Output

num_results

Points to the number of entries that were returned in the *results* parameter array.

results

Specifies the array that contains the matching GLB database entries, up to the number specified in the *max_results* parameter. If the array contains any entries for servers on the local network, those entries appear first.

status

Points to the completion status.

Description

The **lb_$lookup_object** routine returns GLB database entries whose **object** fields match the specified object. It returns information about all replicas of an object and all interfaces to the object.

The **lb_$lookup_object** routine cannot return more than the number of matching entries specified by *max_results* parameter at one time. The *lookup_handle* parameter directs this routine to do sequential lookup calls to find all matching entries.

Notes:

1. The Location Broker does not prevent modification of the database between lookup calls, which can cause the locations of entries relative to a value of the *lookup_handle* parameter to change. If multiple calls are made to find all matching results in the database, the returned information may skip or duplicate entries from the database.
2. It is also possible for the results of a single lookup call to skip or duplicate entries. This can occur if the size of the results exceeds the size of an RPC packet (64K bytes).

Example

1. To look up GLB database entries for the bank **bank_id**, enter the following:

```
lb_$lookup_object(&bank_id, &lookup_handle, MAX_LOCS, &n_locs,
   bank_loc, &st);
```

Implementation Specifics

This Library Routine is part of Network Computing System in Network Support Facilities in Base Operating System (BOS) Runtime.

lb_$lookup_object_local Library Routine (NCS)

Purpose

Looks up information about an object in an LLB database.

Syntax

void lb_$lookup_object_local (*object*, *sockaddr*, *slength*, *lookup_handle*, *max_results*, *num_results*, *results*, *status*)
uuid_$t **object*;
socket_$addr_t **sockaddr*;
unsigned long *slength*;
lb_$lookup_handle_t **lookup_handle*;
unsigned long *max_results*;
unsigned long **num_results*;
lb_$entry_t *results* **[ ];**
status_$t **status*;

Parameters

Input

object
Points to the UUID of the object being looked up.

sockaddr
Specifies the location of the LLB database to be searched. The socket address must specify the network address of a host. However, the port number in the socket address is ignored. The lookup request is always sent to the host's LLB port.

slength
Specifies the length, in bytes, of the socket address specified by the *sockaddr* parameter.

max_results
Specifies the maximum number of matching entries that can be returned by a single call. This should be the number of elements in the *results* parameter array.

Input/Output

lookup_handle
Specifies a location in the database. On input, the value of the *lookup_handle* parameter indicates the location in the database where the search begins. An input value of **lb_$default_lookup_handle** specifies that the search starts at the beginning of the database.

On return, the *lookup_handle* indicates the next unsearched part of the database (that is, the point at which the next search should begin). A return value of **lb_$default_lookup_handle** indicates that the search reached the end of the database. Any other value indicates that the search found at most the number of matching entries specified by the *max_results* parameter before it reached the end of the database.

Output

num_results — Points to the number of entries that were returned in the *results* parameter array.

results — Specifies the array that contains the matching GLB database entries, up to the number specified in the *max_results* parameter. If the array contains any entries for servers on the local network, those entries appear first.

status — Points to the completion status.

Description

The **lb_$lookup_object_local** routine searches the specified LLB database and returns all entries whose **object** fields match the specified object. It returns information about all replicas of an object and all interfaces to the object that are located on the specified host.

The **lb_$lookup_interface** routine cannot return more than the number of matching entries specified by the *max_results* parameter at one time. The *lookup_handle* parameter directs this routine to do sequential lookup calls to find all matching entries.

Notes:

1. The Location Broker does not prevent modification of the database between lookup calls. This can cause the locations of entries relative to a value of the *lookup_handle* parameter to change. If multiple calls are made to find all matching results in the database, the returned information may skip or duplicate entries from the database.
2. It is also possible for the results of a single lookup call to skip or duplicate entries. This can occur if the size of the results exceeds the size of an RPC packet (64K bytes).

Example

1. In the following example, the **repob** object is replicated, with only one replica located on any host. To look up information about the **repob** object, enter the following:

```
lb_$lookup_object_local (&repob_id, &location, location_length,
  &lookup_handle, 1, &num_results, myob_entry, &st);
```

Since there is only one replica located on any host, the routine returns at most one result.

Implementation Specifics

This Library Routine is part of Network Computing System in Network Support Facilities in Base Operating System (BOS) Runtime.

lb_$lookup_range Library Routine (NCS)

Purpose

Looks up information in a GLB or LLB database.

Syntax

void lb_$lookup_range (*object*, *object_type*, *object_interface*, *location*, *lookup_handle*, *location_length*, *max_results*, *num_results*, *results*, *status*)

uuid_$t **object*;
uuid_$t **object_type*;
uuid_$t **object_interface*;
socket_$addr_t **location*;
unsigned long *location_length*;
lb_$lookup_handle_t **lookup_handle*;
unsigned long *max_results*;
unsigned long **num_results*;
lb_$entry_t *results* **[ ];**
status_$t **status*;

Parameters

Input

object — Points to the UUID of the object being looked up.

object_type — Points to the UUID of the type being looked up.

object_interface — Points to the UUID of the interface being looked up.

location — Points to the location of the database to be searched. If the value of the *location_length* parameter is 0, the GLB database is searched. Otherwise, the LLB database at the host specified by the socket address is searched. If the LLB database is searched, the port number in the socket address is ignored, and the lookup request is sent to the LLB port.

location_length — Specifies the length, in bytes, of the socket address indicated by the *location* parameter. A value of 0 indicates that the GLB database is to be searched.

max_results — Specifies the maximum number of matching entries that can be returned by a single call. This should be the number of elements in the *results* array.

Input/Output

lookup_handle
Specifies a location in the database. On input, the value of the *lookup_handle* parameter indicates the location in the database where the search begins. An input value of **lb_$default_lookup_handle** specifies that the search starts at the beginning of the database.

On return, the *lookup_handle* parameter indicates the next unsearched part of the database (that is, the point at which the next search should begin). A return value of **lb_$default_lookup_handle** indicates that the search reached the end of the database. Any other value indicates that the search found at most the number of matching entries specified by the *max_results* parameter before it reached the end of the database.

Output

num_results
Points to the number of entries that were returned in the *results* parameter array.

results
Specifies the array that contains the matching GLB database entries, up to the number specified in the *max_results* parameter. If the array contains any entries for servers on the local network, those entries appear first.

status
Points to the completion status.

Description

The **lb_$lookup_range** routine returns database entries that contain matching **object**, **obj_type**, and **obj_interface** identifiers. A value of **uuid_$nil** in any of these input parameters acts as a wild card and matches all values in the corresponding entry field. You can include wild cards in any combination of these parameters.

The **lb_$lookup_interface** routine cannot return more than the number of matching entries specified by the *max_results* parameter at one time. The *lookup_handle* parameter directs this routine to do sequential lookup calls to find all matching entries.

Notes:

1. The Location Broker does not prevent modification of the database between lookup calls, which can cause the locations of entries relative to a value of the *lookup_handle* parameter value to change. If multiple calls are made to find all matching results in the database, the returned information may skip or duplicate entries from the database.
2. It is also possible for the results of a single lookup call to skip or duplicate entries. This can occur if the size of the results exceeds the size of an RPC packet (64K bytes).

Example

1. To look up information in the GLB database about the **change_if** interface to the **proc_db2** object (which is of the **proc_db** type), enter the following:

   ```
   lb_$lookup_range (&proc_db2_id, &proc_db_id, &change_if_id,
      glb, 0, &lookup_handle, 10, &num_results, results, &st);
   ```

   The name `glb` is defined elsewhere as a null pointer. The *results* parameter is a 10-element array of the **lb_$entry_t** type.

Implementation Specifics

This Library Routine is part of Network Computing System in Network Support Facilities in Base Operating System (BOS) Runtime.

lb_$lookup_type Library Routine (NCS)

Purpose

Looks up information about a type in the GLB database.

Syntax

void lb_$lookup_type (*object_type*, *lookup_handle*, *max_results*, *num_results*, *results*, *status*)
uuid_$t **object_type*;
lb_$lookup_handle_t **lookup_handle*;
unsigned long *max_results*;
unsigned long **num_results*;
lb_$entry_t *results* **[ ]**;
status_$t **status*;

Parameters

Input

object_type
Points to the UUID of the type being looked up.

max_results
Specifies the maximum number of matching entries that can be returned by a single call. This should be the number of elements in the *results* parameter array.

Input/Output

lookup_handle
Specifies a location in the database. On input, the value of the *lookup_handle* parameter indicates the location in the database where the search begins. An input value of **lb_$default_lookup_handle** specifies that the search starts at the beginning of the database.

On return, the *lookup_handle* parameter indicates the next unsearched part of the database (that is, the point at which the next search should begin). A return value of **lb_$default_lookup_handle** indicates that the search reached the end of the database. Any other value indicates that the search found at most the number of matching entries specified by the *max_results* parameter before it reached the end of the database.

Output

num_results
Points to the number of entries that were returned in the *results* parameter array.

results
Specifies the array that contains the matching GLB database entries, up to the number specified in the *max_results* parameter. If the array contains any entries for servers on the local network, those entries appear first.

status
Points to the completion status.

Description

The **lb_$lookup_type** routine returns GLB database entries whose **obj_type** fields match the specified type. It returns information about all replicas of all objects of that type and about all interfaces to each object.

The **lb_$lookup_type** routine cannot return more than the number of matching entries specified by the *max_results* parameter at one time. The *lookup_handle* parameter directs this routine to do sequential lookup calls to find all matching entries.

Notes:

1. The Location Broker does not prevent modification of the database between lookup calls, which can cause the locations of entries relative to a value of the *lookup_handle* parameter to change. If multiple calls are made to find all matching results in the database, the returned information may skip or duplicate entries from the database.
2. It is also possible for the results of a single lookup call to skip or duplicate entries. This can occur if the size of the results exceeds the size of an RPC packet (64K bytes).

Example

1. To look up information in the GLB database about the **array_proc** type, enter the following:

   ```
   lb_$lookup_type (&array_proc_id, &lookup_handle, 10,
        &num_results, &results, &st)
   ```

   The *results* parameter is a 10-element array of the **lb_$entry_t** type.

Implementation Specifics

This Library Routine is part of Network Computing System in Network Support Facilities in Base Operating System (BOS) Runtime.

lb_$register Library Routine (NCS)

Purpose

Registers an object and an interface with the Location Broker.

Syntax

void lb_$register (*object*, *object_type*, *object_interface*, *flags*, *annotation*, *sockaddr*, *slength*, *entry*, *status*)
uuid_$t **object*;
uuid_$t **object_type*;
uuid_$t **object_interface*;
b_$server_flag_t **flags*;
char *annotation* **[ ]**;
socket_$addr_t **sockaddr*;
unsigned long *slength*;
lb_$entry_t **entry*;
status_$t **status*;

Parameters

Input

object	Points to the UUID of the object being looked up.
object_type	Points to the UUID of the type being looked up.
object_interface	Points to the UUID of the interface being looked up.
flags	Points to the server that implements the interface. The value must be 0 or **lb_$server_flag_local**.
annotation	Specifies information, such as textual descriptions of the object and the interface. It is set in a 64-character array.
sockaddr	Points to the socket address of the server that exports the interface to the object.
slength	Specifies the length, in bytes, of the socket address (*sockaddr*).

Output

entry	Points to the copy of the entry that was entered in the Location Broker database.
status	Points to the completion status.

Description

The **lb_$register** routine registers with the Location Broker a specific interface to an object and the location of a server that exports that interface. This routine replaces an existing entry in the Location Broker database that matches the *object*, *object_type*, and *object_interface* parameters as well as both the address family and host in the socket address specified by the *sockaddr* parameter. If no such entry exists, the routine adds a new entry to the database.

If the *flags* parameter has a value of **lb_$server_flag_local**, the entry is registered only in the LLB database at the host where the call is issued. Otherwise, the entry is registered in both the LLB and the GLB databases.

Example

1. To register the **bank** interface to the **bank_id** object, enter the following:

```
lb_$register (&bank_id, &bank_$uuid, &bank_$if_spec.id, 0,
   BankName, &saddr, slen, &entry, &st);
```

Implementation Specifics

This Library Routine is part of Network Computing System in Network Support Facilities in Base Operating System (BOS) Runtime.

lb_$unregister Library Routine (NCS)

Purpose

Removes an entry from the Location Broker database.

Syntax

void lb_$unregister (*entry*, *status*)
lb_$entry_t **entry*;
status_$t **status*;

Parameters

Input

entry	Points to the entry being removed from the Location Broker database.

Output

status	Points to the completion status.

Description

The **lb_$unregister** routine removes from the Location Broker database the entry that matches the value supplied in the *entry* parameter. The value of the *entry* parameter should be identical to that returned by the **lb_$register** routine when the database entry was created. However, the **lb_$unregister** routine does not compare all of the fields in the *entry* parameter. It ignores the **flags** field, the **annotation** field, and the port number in the **saddr** field.

This routine removes the entry from the LLB database on the local host (the host that issues the call). If the **flags** field of the *entry* parameter is not the value **lb_$server_flag_local**, this routine also removes the entry from all replicas of the GLB database.

Example

1. To unregister the entry specified by the **BankEntry** results structure, which was obtained from a previous call to the **lb_$register** routine, enter the following:

   ```
   lb_$unregister (&BankEntry, &st);
   ```

Implementation Specifics

This Library Routine is part of Network Computing System in Network Support Facilities in Base Operating System (BOS) Runtime.

pfm_$cleanup Library Routine (NCS)

Purpose

Establishes a cleanup handler.

Syntax

#include <idl/c/base.h>
#include <idl/c/pfm.h>

status_$t
pfm_$cleanup(*cleanup_record***)**
pfm_$cleanup_rec **cleanup_record*;

Parameters

Input

cleanup_record — A record of the context in which the **pfm_$cleanup** routine is called. A program should treat this as an opaque data structure and not try to alter or copy its contents. It is needed by the **pfm_$cleanup** and **pfm_$reset_cleanup** routines to restore the context of the calling process at the cleanup handler entry point.

Description

The **pfm_$cleanup** routine establishes a cleaunup handler that is executed when a fault occurs. A cleaunup handler is a piece of code executed before a program exits when a signal is received by the process. The cleaunup handler begins with a call to the **pfm_$cleanup** routine. This routine registers an entry point with the system where program execution resumes when a fault occurs. When a fault occurs, execution resumes after the most recent call to the **pfm_$cleanup** routine.

There can be more than one cleaunup handler in a program. Multiple cleaunup handlers are executed consecutively on a last-in/first-out basis, starting with the most recently established handler and ending with the first cleaunup handler. The system provides a default cleaunup handler established at program invocation. The default cleaunup handler is always called last, just before a program exits, and releases any system resources still held before returning control to the process that invoked the program.

When called to establish a cleaunup handler, the **pfm_$cleanup** routine returns the **pfm_$cleanup_set** status to indicate that the cleaunup handler was successfully established. When the cleaunup handler is entered in response to a fault signal, the **pfm_$cleanup** routine effectively returns the value of the fault that triggered the handler.

Note: Cleanup handler code runs with asynchronous faults inhibited. When the **pfm_$cleanup** routine returns something other than **pfm_$cleanup_set** status, which indicates that a fault has occurred, there are four possible ways to leave the clean_up code:

- The program can call the **pfm_$signal** routine to start the next cleaunup handler with a different fault signal.
- The program can call the **pfm_$exit** routine to start the next cleaunup handler with the same fault signal.

- The program can continue with the code following the cleaunup handler. It should generally call the **pfm_$enable** routine to re-enable asynchronous faults. Execution continues from the end of the cleaunup handler code; it does not resume where the fault signal was received.
- The program can re-establish the handler by calling the **pfm_$reset_cleanup** routine before proceeding.

Example

1. To establish a cleaunup handler for a routine, use the following:

   ```
   fst = pfm_cleanup(crec)
   ```

   where `fst` is of type **status_$t** and `crec` is of type **pfm_$cleanup_crec**.

Implementation Specifics

This Library Routine is part of Network Computing System in Network Support Facilities in Base Operating System (BOS) Runtime.

Related Information

The **pfm_$signal** routine.

pfm_$enable Library Routine (NCS)

Purpose

Enables asynchronous faults.

Syntax

#include <idl/c/base.h>
#include <idl/c/pfm.h>

void
pfm_$enable (*void***)**

Description

The **pfm_$enable** routine enables asynchronous faults after they have been inhibited by a call to the **pfm_$inhibit** routine. The **pfm_$enable** routine causes the operating system to pass asynchronous faults on to the calling process.

While faults are inhibited, the operating system holds at most one asynchronous fault. Consequently, when **pfm_$enable** returns, there can be at most one fault waiting on the process. If more than one fault was received between calls to the **pfm_$inhibit** and **pfm_$enable** routines, the process receives the first asynchronous fault received while faults were inhibited.

Example

1. To enable asynchronous interrupts to occur after a call to the **pfm_$inhibit** routine, use the following:

```
pfm_$enable( );
```

Implementation Specifics

This Library Routine is part of Network Computing System in Network Support Facilities in Base Operating System (BOS) Runtime.

Related Information

The **pfm_$enable_faults** routine, **pfm_$inhibit** routine.

pfm_$enable_faults Library Routine (NCS)

Purpose

Enables asynchronous faults.

Syntax

#include <idl/c/base.h>
#include <idl/c/pfm.h>

void
pfm_$enable_faults (*void*)

Description

The **pfm_$enable_faults** routine enables asynchronous faults after they have been inhibited by a call to the **pfm_$inhibit_faults** routine. The **pfm_$enable_faults** routine causes the operating system to pass asynchronous faults on to the calling process.

While faults are inhibited, the operating system holds at most one asynchronous fault. Consequently, when **pfm_$enable_faults** returns, there can be at most one fault waiting on the process. If more than one fault was received between calls to the **pfm_$inhibit_faults** and **pfm_$enable_faults** routines, the process receives the first asynchronous fault received while faults were inhibited.

Example

1. To enable faults to occur after a call to **pfm_$inhibit_faults**, use the following:

```
pfm_$enable_faults( );
```

Implementation Specifics

This Library Routine is part of Network Computing System in Network Support Facilities in Base Operating System (BOS) Runtime.

Related Information

The **pfm_$enable** routine, **pfm_$inhibit_faults** routine.

pfm_$inhibit Library Routine (NCS)

Purpose

Inhibits asynchronous faults.

Syntax

#include <idl/c/base.h>
#include <idl/c/pfm.h>

void
pfm_$inhibit (*void*)

Description

The **pfm_$inhibit** routine prevents asynchronous faults from being passed to the calling process. While faults are inhibited, the operating system holds at most one asynchronous fault. Consequently, a call to the **pfm_$inhibit** routine can result in the loss of some signals. For that and other reasons, it is good practice to inhibit faults only when absolutely necessary.

Note: This routine has no effect on the processing of synchronous faults, such as access violations or floating-point and overflow exceptions.

Example

1. To prevent asynchronous interrupts from occurring in a critical portion of a routine, use the following:

```
pfm_$inhibit( );
```

Implementation Specifics

This Library Routine is part of Network Computing System in Network Support Facilities in Base Operating System (BOS) Runtime.

Related Information

The **pfm_$enable** routine, **pfm_$inhibit_faults** routine.

pfm_$inhibit_faults Library Routine (NCS)

Purpose

Inhibits asynchronous faults, but allows task switching.

Syntax

#include <idl/c/base.h>
#include <idl/c/pfm.h>

void
pfm_$inhibit_faults (*void*)

Description

The **pfm_$inhibit** routine prevents asynchronous faults, except for time-sliced task switching, from being passed to the calling process. While faults are inhibited, the operating system holds at most one asynchronous fault. Consequently, a call to the **pfm_$inhibit_faults** routine can result in the loss of some signals. For that and other reasons, it is good practice to inhibit faults only when absolutely necessary.

Note: This routine has no effect on the processing of synchronous faults, such as access violations or floating-point and overflow exceptions.

Example

1. To prevent faults from occurring in a critical portion of a routine, use the following:

```
pfm_$inhibit_faults( );
```

Implementation Specifics

This Library Routine is part of Network Computing System in Network Support Facilities in Base Operating System (BOS) Runtime.

Related Information

The **pfm_$enable_faults** routine, **pfm_$inhibit** routine.

pfm_$init Library Routine (NCS)

Purpose

Initializes the program fault management (PFM) package.

Syntax

#include <idl/c/base.h>
#include <idl/c/pfm.h>

void
pfm_$init (*flags*)
unsigned long *flags*;

Parameters

Input

flags Indicates which initialization activities to perform. Currently only one value is valid: **pfm_$init_signal_handlers**. This causes C signals to be intercepted and converted to PFM signals. The signals intercepted are **SIGINT**, **SIGILL**, **SIGFPE**, **SIGTERM**, **SIGHUP**, **SIGQUIT**, **SIGTRAP**, **SIGBUS**, **SIGSEGV**, and **SIGSYS**.

Description

The **pfm_$init** routine initializes the PFM package. Applications that use the PFM package should invoke the **pfm_$init** routine before invoking any other NCS routines.

Example

1. To initialize the PFM subsystem, use the following:

```
pfm_$init(pfm_$init_signal_handlers);
```

Implementation Specifics

This Library Routine is part of Network Computing System in Network Support Facilities in Base Operating System (BOS) Runtime.

pfm_$reset_cleanup Library Routine (NCS)

Purpose

Resets a cleanup handler.

Syntax

#include <idl/c/base.h>
#include <idl/c/pfm.h>

void
pfm_$reset_cleanup (*cleanup_record*, *status*)
pfm_$cleanup_rec **cleanup_record*;
status_$t **status*;

Parameters

Input

cleanup_ record	A record of the context at the cleanup handler entry point. It is supplied by the **pfm_$cleanup** routine when the cleanup handler is first established.

Output

status	Points to the completion status.

Description

The **pfm_$reset_cleanup** routine re-establishes the cleanup handler last entered so that any subsequent errors enter it first. This procedure should only be used within cleanup handler code.

Example

1. To re-establish a cleanup handler, use the following:

```
pfm_$reset_cleanup(crec, st);
```

where the `crec` cleanup record is a valid cleanup handler.

Implementation Specifics

This Library Routine is part of Network Computing System in Network Support Facilities in Base Operating System (BOS) Runtime.

pfm_$rls_cleanup Library Routine (NCS)

Purpose

Releases cleanup handlers.

Syntax

#include <idl/c/base.h>
#include <idl/c/pfm.h>

void
pfm_$rls_cleanup(*cleanup_record*, *status*)
pfm_$cleanup_rec **cleanup_record*;
status_$t **status*;

Parameters

Input

cleanup_record — The cleanup record for the first cleanup handler to release.

Output

status — Points to the completion status. If the *status* parameter has a value of **pfm_$bad_rls_order**, it means that the caller attempted to release a cleanup handler before releasing all handlers established after it. This status is only a warning. The intended cleanup handler is released, along with all cleanup handlers established after it.

Description

The **pfm_$rls_cleanup** routine releases the cleanup handler associated with the *cleanup_record* parameter and all cleanup handlers established after it.

Example

1. To release an established cleanup handler, use the following:

```
pfm_$rls_cleanup(crec, st);
```

where `crec` is a valid cleanup record established by the **pfm_$cleanup** routine.

Implementation Specifics

This Library Routine is part of Network Computing System in Network Support Facilities in Base Operating System (BOS) Runtime.

pfm_$signal Library Routine

Purpose

Signals the calling process.

Syntax

#include <idl/c/base.h>
#include <idl/c/pfm.h>

void
pfm_$signal (*fault_signal***)**
status_$t **fault_signal*;

Parameters

Input

fault_ signal	A fault code.

Description

The **pfm_$signal** routine signals the fault specified by the *fault_signal* parameter to the calling process. It is usually called to leave cleanup handlers.

Note: This routine does not return when successful.

Example

1. To send the calling process a fault signal, use the following:

   ```
   pfm_$signal(fst);
   ```

   where `fst` is a valid PFM fault.

Implementation Specifics

This Library Routine is part of Network Computing System in Network Support Facilities in Base Operating System (BOS) Runtime.

rpc_$alloc_handle Library Routine (NCS)

Purpose

Creates an RPC handle.

Syntax

handle_t rpc_$alloc_handle (*object_id*, *family*, *status*)

uuid_$t **object_id*;
unsigned long *family*;
status_$t **status*;

Parameters

Input

object_id — Points to the UUID of the object to be accessed. If there is no specific object, specify **uuid_$nil** as the value.

family — Specifies the address family to use in communications to access the object.

Output

status — Points to the completion status.

Description

The **rpc_$alloc_handle** routine creates an unbound RPC handle that identifies a particular object but not a particular server or host. A remote procedure call made using an unbound handle is broadcast to all Local Location Brokers (LLBs) on the local network. If the call's interface and the object identified by the handle are both registered with any LLB, that LLB forwards the request to the registering server. The client RPC runtime library returns the first response that it receives and binds the handle to the server.

Note: This routine is used by clients only.

Return Value

Upon successful completion, the **rpc_$alloc_handle** routine returns an RPC handle identifying the remote object in the form **handle_t**. This handle is used as the first input parameter to remote procedure calls with explicit handles.

Example

The following statement allocates a handle that identifies the Acme company's payroll database object:

```
handle = rpc_$alloc_handle (&acme_pay_id, socket_$dds, &st);
```

Implementation Specifics

This Library Routine is part of Network Computing System in Network Support Facilities in Base Operating System (BOS) Runtime.

rpc_$bind Library Routine (NCS)

Purpose

Allocates an RPC handle and sets its binding to a server.

Syntax

handle_t rpc_$bind (*object_id*, *sockaddr*, *slength*, *status*)
uuid_$t **object_id*;
socket_$addr_t **sockaddr*;
unsigned long *slength*;
us_$t **status*;

Parameters

Input

object_id Points to the UUID of the object to be accessed. If there is no specific object, specify **uuid_$nil** as the value.

sockaddr Points to the socket address of the server.

slength Specifies the length, in bytes, of the socket address (*sockaddr*).

Output

status Points to the completion status.

Description

The **rpc_$bind** function creates a fully bound RPC handle that identifies a particular object and server. This routine is equivalent to an **rpc_$alloc_handle** routine followed by an **rpc_$set_binding** routine.

Note: This routine is used by clients only.

Return Value

Upon successful completion, this routine returns an RPC handle (**handle_t**) that identifies the remote object. This handle is used as the first input parameter to remote procedure calls with explicit handles.

Example

The following example binds a banking client program to the specified object and socket address:

```
h = rpc_$bind(&bank_id, &bank_loc[0].saddr, bank_loc[0].saddr_len,
    &st);
```

The **bank_loc** structure is the *results* parameter of a previous Location Broker lookup call.

Implementation Specifics

This Library Routine is part of Network Computing System in Network Support Facilities in Base Operating System (BOS) Runtime.

Related Information

The **rpc_$alloc_handle** routine, **rpc_$set_binding** routine.

rpc_$clear_binding Library Routine (NCS)

Purpose

Unsets the binding between an RPC handle and a host and server.

Syntax

void rpc_$clear_binding (*handle*, *status*)
handle_t *handle*;
status_$t **status*;

Parameters

Input

handle Specifies the RPC handle from which the binding is being cleared.

Output

status Points to the completion status.

Description

The **rpc_$clear_binding** routine removes any association between an RPC handle and a particular server and host, but does not remove the association between the handle and an object. This routine saves the RPC handle so that it can be reused to access the same object, either by broadcasting or after resetting the binding to another server.

A remote procedure call made using an unbound handle is broadcast to all Local Location Brokers (LLBs) on the local network. If the call's interface and the object identified by the handle are both registered with any LLB, that LLB forwards the request to the registering server. The client RPC runtime library returns the first response that it receives and binds the handle to the server.

The **rpc_$clear_binding** routine reverses an **rpc_$set_binding** routine.

Note: This routine is used by clients only.

Example

To clear the binding represented in a handle, enter the following:

```
rpc_$clear_binding(handle, &st);
```

Implementation Specifics

This Library Routine is part of Network Computing System in Network Support Facilities in Base Operating System (BOS) Runtime.

Related Information

The **rpc_$set_binding** routine.

rpc_$clear_server_binding Library Routine (NCS)

Purpose

Unsets the binding between an RPC handle and a server.

Syntax

void rpc_$clear_server_binding (*handle*, *status*)
handle_t *handle*;
status_$t **status*;

Parameters

Input

handle Specifies the RPC handle from which the server binding is being cleared.

Output

status Points to the completion status.

Description

The **rpc_$clear_server_binding** routine removes the association between an RPC handle and a particular server (which is a particular port number), but does not remove the associations with an object and a host. For example, the routine unmaps the handle to the port number, but it leaves the object and host associated through a network address.

This routine replaces a fully bound handle with a bound-to-host handle. A bound-to-host handle identifies an object located on a particular host, but does not identify a server exporting an interface to the object.

If a client uses a bound-to-host handle to make a remote procedure call, the call is sent to the Local Location Broker (LLB) forwarding port at the host identified by the handle. If the call's interface and the object identified by the handle are both registered with the host's LLB, the LLB forwards the request to the registering server. When the client RPC runtime library receives a response, it binds the handle to the server. Subsequent remote procedure calls that use this handle are then sent directly to the bound server's port.

The **rpc_$clear_server_binding** routine is used for client error recovery when a server dies. The port that a server uses when it restarts is not necessarily the same port that it used previously. Therefore, the binding that the client was using may not be correct. This routine enables the client to unbind from the dead server while retaining the binding to the host. When the client sends a request, the binding is automatically set to the server's new port.

Note: This routine is used by clients only.

Example

To clear the server binding represented in a handle, enter the following:

```
rpc_$clear_server_binding(handle, &st);
```

Implementation Specifics

This Library Routine is part of Network Computing System in Network Support Facilities in Base Operating System (BOS) Runtime.

rpc_$dup_handle Library Routine (NCS)

Purpose

Makes a copy of an RPC handle.

Syntax

handle_t rpc_$dup_handle (*handle*, *status*)
handle_t *handle*;
status_$t **status*;

Parameters

Input

handle Specifies the RPC handle to be copied.

Output

status Points to the completion status.

Description

The **rpc_$dup_handle** routine returns a copy of an existing RPC handle. Both handles can then be used in the client program for concurrent multiple accesses to a binding. Because all duplicates of a handle reference the same data, a call to the **rpc_$set_binding**, **rpc_$clear_binding**, or **rpc_$clear_server_binding** routine made on any one duplicate affects all duplicates. However, an RPC handle is not freed until the **rpc_$free_handle** routine is called on all copies of the handle.

Note: This routine is used by clients only.

Return Value

Upon successful completion, this routine returns the duplicate handle (**handle_t**).

Example

1. To create as **thread_2_handle** a copy of a handle, enter the following:

```
thread_2_handle = rpc_$dup_handle(handle, &st);
```

Implementation Specifics

This Library Routine is part of Network Computing System in Network Support Facilities in Base Operating System (BOS) Runtime.

rpc_$free_handle Library Routine (NCS)

Purpose

Frees an RPC handle.

Syntax

void rpc_$free_handle (*handle*, *status*)
handle_t *handle*;
status_$t **status*;

Parameters

Input

handle Specifies the RPC handle to be freed.

Output

status Points to the completion status.

Description

The **rpc_$free_handle** routine frees an RPC handle by clearing the association between the handle and a server or an object, and then releasing the resources identified by the RPC handle. The client program cannot use a handle after it is freed.

To make multiple RPC calls using the same interface but different socket addresses, replace the binding in an existing handle with the **rpc_$set_binding** routine instead of creating a new handle with the **rpc_$free_handle** and **rpc_$bind** routines.

To free copies of RPC handles created by the **rpc_$dup_handle** routine, use the **rpc_$free_handle** routine once for each copy of the handle. However, the RPC runtime library does not differentiate between calling the **rpc_$free_handle** routine several times on one copy of a handle and calling it one time for each of several copies of a handle. Therefore, if you use duplicate handles, you must ensure that no thread inadvertently makes multiple **rpc_$free_handle** calls on a single handle.

Note: This routine is used by clients only.

Example

1. To free two copies of a handle, enter the following:

```
rpc_$free_handle(handle, &st);
rpc_$free_handle(thread_2_handle, &st);
```

Implementation Specifics

This Library Routine is part of Network Computing System in Network Support Facilities in Base Operating System (BOS) Runtime.

Related Information

The **rpc_$set_binding** routine, **rpc_$dup_handle** routine.

rpc_$inq_binding Library Routine (NCS)

Purpose

Returns the socket address represented by an RPC handle.

Syntax

void rpc_$inq_binding (*handle*, *sockaddr*, *slength*, *status*)
handle_t *handle*;
socket_$addr_t **sockaddr*;
unsigned long **slength*;
status_$t **status*;

Parameters

Input

handle Specifies an RPC handle.

Output

sockaddr Points to the socket address represented by the *handle* parameter.

slength Points to the length, in bytes, of the socket address (*sockaddr*).

status Points to the completion status.

Description

The **rpc_$inq_binding** routine enables a client to determine the socket address, and therefore the server, identified by an RPC handle. It can be used to determine which server is responding to a remote procedure call when a client uses an unbound handle in the call.

Note: This routine is used by clients only.

Diagnostics

The **rpc_$inq_binding** routine fails if the following is true:

rpc_$unbound_handle The handle is not bound and does not represent a specific host address.

Example

1. The Location Broker administrative tool, **lb_admin**, uses the following statement to determine the particular GLB that responded to a lookup request:

```
rpc_$inq_binding(glb_$handle, &global_broker_addr,
     &global_broker_addr_len, &status);
```

Implementation Specifics

This Library Routine is part of Network Computing System in Network Support Facilities in Base Operating System (BOS) Runtime.

rpc_$inq_object Library Routine (NCS)

Purpose

Returns the object UUID represented by an RPC handle.

Syntax

void rpc_$inq_object (*handle*, *object_id*, *status*)
handle_t *handle*;
uuid_$t **object_id*;
status_$t **status*;

Parameters

Input

handle Specifies an RPC handle.

Output

object_id Points to the UUID of the object identified by the *handle* parameter.

status Points to the completion status.

Description

The **rpc_$inq_object** routine enables a server to determine the particular object that a client is accessing. A server must use **rpc_$inq_object** if it exports an interface through which multiple objects may be accessed.

A server can make this call only if the interface uses explicit handles (that is, if each operation in the interface has a handle argument). If the interface uses an implicit handle, the handle identifier is not passed to the server.

Note: This routine is used by servers only.

Example

1. A database server that manages multiple databases must determine the particular database to be accessed whenever it receives a remote procedure call. Each manager routine therefore makes the following call:

   ```
   rpc_$inq_object(handle, &db_uuid, &st);
   ```

   The routine then uses the returned UUID to identify the database to be accessed.

Implementation Specifics

This Library Routine is part of Network Computing System in Network Support Facilities in Base Operating System (BOS) Runtime.

rpc_$listen Library Routine (NCS)

Purpose

Listens for and handles remote procedure call packets.

Syntax

void rpc_$listen (*max_calls*, *status*)
unsigned long *max_calls*;
status_$t **status*;

Parameters

Input

max_calls Specifies the maximum number of calls (in the range 1 through 10) that the server is allowed to process concurrently.

Output

status Points to the completion status.

Description

The **rpc_$listen** routine dispatches incoming remote procedure call requests to manager procedures and returns the responses to the client. You must issue an **rpc_$use_family** or **rpc_$use_family_wk** routine before you use the **rpc_$listen** routine.

If the value of the *max_calls* parameter is greater than 1, the server RPC runtime library uses Concurrent Programming Support (CPS) to handle multiple calls simultaneously. As a result, the manager routines must be re-entrant. This means they must maintain concurrency controls on any nonlocal variables to prevent conflicts among the various threads of execution.

Note: This routine is used by servers only.

Return Value

This routine normally does not return.

Example

1. To have a server listen for incoming remote procedure call requests, handling up to five concurrently, enter the folllowing:

```
rpc_$listen(5, &status);
```

Implementation Specifics

This Library Routine is part of Network Computing System in Network Support Facilities in Base Operating System (BOS) Runtime.

Related Information

The **rpc_$use_family** routine, **rpc_$use_family_wk** routine.

rpc_$name_to_sockaddr Library Routine (NCS)

Purpose

Converts a host name and port number to a socket address.

Syntax

void rpc_$name_to_sockaddr (*name*, *nlength*, *port*, *family*, *sockaddr*, *slength*, *status*)
char **name*;
unsigned long *nlength*;
unsigned long *port*;
unsigned long *family*;
socket_$addr_t **sockaddr*;
unsigned long **slength*;
status_$t **status*;

Parameters

Input

name
: Points to a host name, and optionally, a port and an address family, in the form: *family*:*host*[*port*]. The *family*: and [*port*] parameters are optional. If you specify a *family* variable as part of the *name* parameter, you must specify **socket_$unspec** in the *family* parameter. The only supported value for the *family* variable is **ip**. The *host* parameter specifies the host name, and *port* specifies a port number in integer form.

nlength
: Specifies the number of characters in the *name* parameter.

port
: Specifies the socket port number. If you are not specifying a well-known port, this parameter should have the value **socket_$unspec_port**. The returned socket address will specify the Local Location Broker (LLB) forwarding port at the host. If you specify the port number in the *name* parameter, this parameter is ignored.

family
: Specifies the address family to use for the socket address. This value corresponds to the communications protocol used to access the socket and determines how the socket address (*sockaddr)* is expressed. If you specify the address family in the *name* parameter, this parameter must have the value **socket_$unspec**.

Output

sockaddr
: Points to the socket address corresponding to the *name*, *port*, and *family* parameters.

slength
: Points to the length, in bytes, of the socket address (specified by the *sockaddr* parameter).

status
: Points to the completion status.

Description

The **rpc_$name_to_sockaddr** routine provides the socket address for a socket, given the host name, the port number, and the address family.

You can specify the socket address information either as one text string in the *name* parameter, or by passing each of the three elements as a separate parameter. When three separate elements are passed, the *name* parameter should contain only the host name.

Example

1. To place in the **sockaddr** structure a socket address that specifies the LLB forwarding port at the host identified by **host_name**, enter the following:

```
rpc_$name_to_sockaddr(host_name, strlen(host_name),
    socket_$unspec_port,socket_$dds, &sockaddr, &slen, &st);
```

Implementation Specifics

This Library Routine is part of Network Computing System in Network Support Facilities in Base Operating System (BOS) Runtime.

rpc_$register Library Routine (NCS)

Purpose

Registers an interface at a server.

Syntax

void rpc_$register (*if_spec*, *epv*, *status*)
rpc_$if_spec_t **if_spec*;
rpc_$epv_t *epv*;
status_$t **status*;

Parameters

Input

if_spec	Points to the interface being registered.
epv	Specifies the entry point vector (EPV) for the operations in the interface. The EPV is normally defined in the server stub that is generated by the NIDL compiler from an interface definition.

Output

status	Points to the completion status.

Description

The **rpc_$register** routine registers an interface with the RPC runtime library. After an interface is registered, the RPC runtime library passes requests for that interface to the server.

You can call **rpc_$register** multiple times with the same interface (for example, from various subroutines of the same server), but each call must specify the same EPV. Each registration increments a reference count for the registered interface. An equal number of calls to the **rpc_$unregister** routine are then required to unregister the interface.

Note: This routine is used by servers only.

Diagnostics

The **rpc_$register** routine fails if one or more of the following is true:

rpc_$too_many_ifs	The maximum number of interfaces is already registered with the server.
rpc_$illegal_register	You are trying to register an interface that is already registered, and you are using an EPV different from the one used when the interface was first registered.

Example

1. To register a **bank** interface with the bank server host's RPC runtime library, enter the following:

```
rpc_$register(&bank_$if_spec, bank_$server_epv, &st);
```

Implementation Specifics

This Library Routine is part of Network Computing System in Network Support Facilities in Base Operating System (BOS) Runtime.

Related Information

The **rpc_$unregister** routine.

rpc_$set_binding Library Routine (NCS)

Purpose

Associates an RPC handle with a server.

Syntax

rpc_$set_binding (*handle*, *sockaddr*, *slength*, *status*)
struct handle_t **handle*;
struct socket_$addr_t **sockaddr*;
int *slength*;
struct status_$t **status*;

Parameters

Input

handle	Specifies an RPC handle.
sockaddr	Specifies the socket address of the server with which the handle is being associated.
slength	Specifies the length, in bytes, of the socket address (*sockaddr*).

Output

status	Specifies the completion status.

Description

The **rpc_$set_binding** routine sets the binding of an RPC handle to the specified server. The handle then identifies a specific object at a specific server. Any subsequent remote procedure calls that a client makes using the handle are sent to this destination. This routine can also replace an existing binding in a fully bound handle, or set the binding in an unbound handle.

Note: This routine is used by clients only.

Example

1. To set the binding on the **m_handle** handle to the first server in the **results** array, which was returned by a previous Location Broker lookup call, enter the following:

```
rpc_$set_binding(m_handle, &lb_reslts[0].saddr,
    lb_reslts[0].saddr_len, &st);
```

Implementation Specifics

This Library Routine is part of Network Computing System in Network Support Facilities in Base Operating System (BOS) Runtime.

rpc_$sockaddr_to_name Library Routine (NCS)

Purpose

Converts a socket address to a host name and port number.

Syntax

void rpc_$sockaddr_to_name (*sockaddr*, *slength*, *name*, *nlength*, *port*, *status*)

socket_$addr_t **sockaddr*;
unsigned long *slength*;
unsigned long **nlength*;
char **name*;
unsigned long **port*;
status_$t **status*;

Parameters

Input

sockaddr Points to a socket address.

slength Specifies the length, in bytes, of socket address (*sockaddr*).

Input/Output

nlength On input, points to the length of the *name* parameter in the buffer. On output, points to the number of characters returned in the *name* parameter.

Output

name Points to a character string that contains the host name and the address family in the format: *family*:*host*. The value of the *family* parameter must be **ip**.

port Points to the socket port number.

status Points to the completion status.

Description

The **rpc_$sockaddr_to_name** routine provides the address family, the host name, and the port number identified by the specified socket address.

Example

1. To take the bank server's socket address, return the server's host name and port, and then print the information, enter the following:

```
rpc_$sockaddr_to_name(&saddr, slen, name, &namelen, &port, &st);
   printf("(bankd) name=\"%.*s\", port=%d\n", name, namelen, port
);
```

Implementation Specifics

This Library Routine is part of Network Computing System in Network Support Facilities in Base Operating System (BOS) Runtime.

rpc_$unregister Library Routine (NCS)

Purpose

Unregisters an interface.

Syntax

void rpc_$unregister (*if_spec*, *status*)
rpc_$if_spec_t **if_spec*;
status_$t **status*;

Parameters

Input

if_spec Points to the interface being unregistered.

Output

status Points to the completion status.

Description

The **rpc_$unregister** routine unregisters an interface that the server previously registered with the RPC runtime library. After an interface is unregistered, the RPC runtime library does not pass requests for that interface to the server.

If a server uses multiple calls to the **rpc_$register** routine to register an interface more than once, then the server must call the **rpc_$unregister** routine an equal number of times to unregister the interface.

Note: This routine is used by servers only.

Example

1. To unregister a matrix arithmetic interface, use the following:

```
rpc_$unregister (&matrix_$if_spec, &st);
```

Implementation Specifics

This Library Routine is part of Network Computing System in Network Support Facilities in Base Operating System (BOS) Runtime.

Related Information

The **rpc_$register** routine.

rpc_$use_family Library Routine (NCS)

Purpose

Creates a socket of a specified address family for an RPC server.

Syntax

void rpc_$use_family (*family*, *sockaddr*, *slength*, *status*)
unsigned long *family*;
socket_$addr_t **sockaddr*;
unsigned long **slength*;
status_$t **status*;

Parameters

Input

family Specifies the address family of the socket to be created. This value corresponds to the communications protocol used to access the socket and determines how the socket address (*sockaddr*) is expressed.

Output

sockaddr Points to the socket address of the socket on which the server listens.

slength Points to the length, in bytes, of the socket address (*sockaddr*).

status Points to the completion status.

Description

The **rpc_$use_family** routine creates a socket for a server without specifying its port number. (The RPC runtime software assigns the port number.) Use this routine to create the server socket unless the server must listen on a particular well-known port. If the socket must listen on a specific well-known port, use the **rpc_$use_family_wk** routine to create the socket.

A server can listen on more than one socket. However, a server normally does not listen on more than one socket for each address family, regardless of the number of interfaces that it exports. Therefore, most servers should make this call once for each supported address family.

Note: This routine is used by servers only.

Diagnostics

The **rpc_$use_family** routine can fail if one or more of the following is true:

rpc_$cant_create_sock
The RPC runtime library is unable to create a socket.

rpc_$cant_bind_sock
The RPC runtime library created a socket but is unable to bind it to a socket address.

rpc_$too_many_sockets
The server is trying to use more than the maximum number of sockets

allowed. The server has called the **rpc_$use_family** or **rpc_$use_family_wk** routines too many times.

Example

1. To create the bank server's socket, enter the following:

   ```
   rpc_$use_family(atoi(argv[1]), &saddr, &slen, &st);
   ```

   The numeric value of the address family to be used is supplied as an argument to the program.

Implementation Specifics

This Library Routine is part of Network Computing System in Network Support Facilities in Base Operating System (BOS) Runtime.

Related Information

The **rpc_$use_family_wk** routine.

rpc_$use_family_wk Library Routine (NCS)

Purpose

Creates a socket with a well-known port for an RPC server.

Syntax

void rpc_$use_family_wk (*family*, *if_spec*, *sockaddr*, *slength*, *status*)

unsigned long *family*;
rpc_$if_spec_t **if_spec*;
socket_$addr_t **sockaddr*;
unsigned long **slength*;
status_$t **status*;

Parameters

Input

family — Specifies the address family of the socket to be created. This value corresponds to the communications protocol used to access the socket and determines how the socket address (*sockaddr*) is expressed.

if_spec — Points to the interface that will be registered by the server. Typically, this parameter is the **$if_spec** interface generated by the NIDL compiler from the interface definition. The well-known port is specified as an interface attribute.

Output

sockaddr — Points to the socket address of the socket on which the server listens.

slength — Points to the length, in bytes, of the socket address (*sockaddr*).

status — Points to the completion status.

Description

The **rpc_$use_family_wk** routine creates a socket that uses the port specified with the *if_spec* parameter. Use this routine to create a socket if a server must listen on a particular well-known port. Otherwise, create the socket with the **rpc_$use_family** routine.

A server can listen on more than one socket. However, a server normally does not listen on more than one socket for each address family, regardless of the number of interfaces that it exports. Therefore, most servers that use well-known ports should make this call once for each supported address family.

Note: This routine is used by servers only.

Diagnostics

The **rpc_$use_family_wk** routine fails if one or more of the following is true:

rpc_$cant_create_sock
The RPC runtime library is unable to create a socket.

rpc_$cant_bind_sock
The RPC runtime library created a socket but is unable to bind it to a socket address.

rpc_$too_many_sockets
The server is trying to use more than the maximum number of sockets allowed. The server has called the **rpc_$use_family** or **rpc_$use_family_wk** routines too many times.

rpc_$addr_in_use
The specified address and port are already in use. This is caused by multiple calls to the **rpc_$use_family_wk** routine with the same well-known port.

Example

1. To create a well-known socket for an array processor server, use the following:

```
rpc_$use_family_wk (socket_$internet, &matrix_$if_spec,
&sockaddr, slen, &st);
```

Implementation Specifics

This Library Routine is part of Network Computing System in Network Support Facilities in Base Operating System (BOS) Runtime.

Related Information

The **rpc_$use_family** routine.

uuid_$decode Library Routine (NCS)

Purpose

Converts a character-string representation of a UUID into a UUID.

Syntax

void uuid_$decode (*uuid_string*, *uuid*, *status*)
char **uuid_string*;
uuid_$t **uuid*;
status_$t **status*;

Parameters

Input

uuid_string	Points to the character-string representation of a UUID in the form **uuid_$string_t**.

Output

uuid	Points to the UUID that corresponds to the character string represented in the *uuid_string* parameter.
status	Points to the completion status.

Description

The **uuid_$decode** routine returns the UUID corresponding to a valid character-string representation of a UUID.

Example

1. The following call returns as `my_uuid` the UUID corresponding to the character-string representation in `my_uuid_rep`:

```
uuid_$decode (my_uuid_rep, &my_uuid, &status);
```

Implementation Specifics

This Library Routine is part of Network Computing System in Network Support Facilities in Base Operating System (BOS) Runtime.

uuid_$encode Library Routine (NCS)

Purpose

Converts a UUID into its character-string representation.

Syntax

void uuid_$encode (*uuid*, *uuid_string*)
uuid_$t **uuid*;
char **uuid_string*;

Parameters

Input

uuid Points to the UUID.

Output

uuid_string Points to the character-string representation of a UUID, in the form **uuid_$string_t**.

Description

The **uuid_$encode** call returns the character-string representation of a UUID.

Example

1. The following call returns as `my_uuid_rep` the character-string representation for the UUID `my_uuid`:

```
uuid_$encode (&my_uuid, my_uuid_rep);
```

Implementation Specifics

This Library Routine is part of Network Computing System in Network Support Facilities in Base Operating System (BOS) Runtime.

uuid_$gen Library Routine (NCS)

Purpose

Generates a new UUID.

Syntax

void uuid_$gen (*uuid*)
uuid_$t **uuid*;

Parameters

Output

uuid Points to the new UUID in the form of **uuid_$t**.

Description

The **uuid_$gen** routine returns a new UUID.

Example

1. The following call returns as `my_uuid` a new UUID:

```
uuid_$gen (&my_uuid);
```

Implementation Specifics

This Library Routine is part of Network Computing System in Network Support Facilities in Base Operating System (BOS) Runtime.

Remote Procedure Calls (RPC)

authdes_create Subroutine

Purpose

Enables the use of DES from the client side.

Library

C Library (**libc.a**)

Syntax

#include <rpc/rpc.h>

AUTH *
authdes_create (*name*, *window*, *syncaddr*, *ckey***)**
**char ****name*;
u_int *window*;
**struct sockaddr ****syncaddr*;
**des_block ****ckey*;

Description

The **authdes_create** subroutine interfaces to the secure authentication system, known as Data Encryption Standard (DES). This subroutine, used from the client side, returns the authentication handle that allows use of the secure authentication system.

Note: The **keyserv** daemon must be running for the DES authentication system to work.

Parameters

name Specifies the network name (or netname) of the server process owner. The *name* parameter can be either the host name derived from the **host2netname** subroutine or the user name derived from the **user2netname** subroutine.

window Specifies the confirmation of the client credentials, given in seconds. A small value for the *window* parameter is more secure than a large one. Yet, choosing too small a value for the *window* parameter increases the frequency of resynchronizations due to clock drift.

syncaddr Identifies clock synchronization. If the *syncaddr* parameter has a NULL value, then the authentication system assumes that the local clock is always in sync with the server's clock. The authentication system will not attempt resynchronizations. However, if an address is supplied, the system uses the address for consulting the remote time service whenever resynchronization is required. This parameter usually contains the address of the RPC server itself.

ckey Specifies the DES key. If the value of the *ckey* parameter is NULL, the authentication system generates a random DES key to be used for the encryption of credentials. However, if a DES key is supplied, the supplied key is used.

Return Values

This subroutine returns a pointer to a DES authentication object.

Implementation Specifics

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

authdes_getucred Subroutine

Purpose

Maps a DES credential into a UNIX credential.

Library

C Library (**libc.a**)

Syntax

#include <rpc/rpc.h>

authdes_getucred (*adc*, *uid*, *gid*, *grouplen*, *groups*)
struct authdes_cred **adc*;
short **uid*;
short **gid*;
short **grouplen*;
int **groups*;

Description

The **authdes_getucred** subroutine interfaces to the secure authentication system known as Data Encryption Standard (DES). The server uses this subroutine to convert a DES credential, which is the independent operating system, into a UNIX credential. The **authdes_getucred** subroutine retrieves necessary information from a cache, instead of using the network information service (NIS).

Note: The **keyserv** daemon must be running for the DES authentication system to work.

Parameters

adc	Points to the DES credential structure.
uid	Specifies the caller's effective user ID (UID).
gid	Specifies the caller's effective group ID (GID).
grouplen	Specifies the group's length.
groups	Points to the group's array.

Return Values

Upon successful completion, this subroutine returns a value of 1. If unsuccessful, it returns a value of 0.

Implementation Specifics

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

Related Information

The **keyserv** daemon.

Network Information Service (NIS) Overview for System Management in *Communication Concepts and Procedures*.

auth_destroy Macro

Purpose

Destroys authentication information.

Library

C Library (**libc.a**)

Syntax

#include <rpc/rpc.h>

void
auth_destroy (*auth*)
auth **auth*;

Description

The **auth_destroy** macro destroys the authentication information structure pointed to by the *auth* parameter. Destroying the structure deallocates private data structures. The use of the *auth* parameter is undefined after calling this macro.

Parameter

auth	Points to the authentication information structure to be destroyed.

Implementation Specifics

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

authnone_create Subroutine

Purpose

Creates NULL authentication.

Library

C Library (**libc.a**)

Syntax

#include <rpc/rpc.h>

AUTH *
authnone_create ( )

Description

The **authnone_create** subroutine creates and returns a default Remote Procedure Call (RPC) authentication handle that passes NULL authentication information with each remote procedure call.

Return Values

This subroutine returns a pointer to an RPC authentication handle.

Implementation Specifics

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

Related Information

The **auth_destroy** macro.

The **authunix_create** subroutine, **authunix_create_default** subroutine, **svcerr_auth** subroutine.

authunix_create Subroutine

Purpose

Creates an authentication handle with AIX permissions.

Library

C Library (**libc.a**)

Syntax

#include <rpc/rpc.h>

AUTH *
authunix_create (*host*, *uid*, *gid*, *len*, *aupgids*)
char **host*;
int *uid*, *gid*
int *len*, **aupgids*;

Description

The **authunix_create** subroutine creates and returns a Remote Procedure Call (RPC) authentication handle with AIX permissions.

Parameters

host	Points to the name of the machine on which the permissions were created.
uid	Specifies the caller's effective user ID (UID).
gid	Specifies the caller's effective group ID (GID).
len	Specifies the length of the groups array.
aupgids	Points to the counted array of groups to which the user belongs.

Return Values

This subroutine returns an RPC authentication handle.

Implementation Specifics

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

Related Information

The **auth_destroy** macro.

The **authnone_create** subroutine, **authunix_create_default** subroutine, **svcerr_auth** subroutine.

authunix_create_default Subroutine

Purpose

Sets the authentication to default.

Library

C Library (**libc.a**)

Syntax

#include <rpc/rpc.h>

AUTH *
authunix_create_default( )

Description

The **authunix_create_default** subroutine calls the **authunix_create** subroutine to create and return the default AIX authentication handle.

Parameters

This subroutine contains no parameters.

Return Values

Upon successful completion, this subroutine returns an authentication handle.

Implementation Specifics

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

Related Information

The **auth_destroy** macro.

The **authnone_create** subroutine, **authunix_create** subroutine, **svcerr_auth** subroutine.

callrpc Subroutine

Purpose

Calls the remote procedure on the machine specified by the *host* parameter.

Library

C Library (**libc.a**)

Syntax

#include <rpc/rpc.h>

callrpc (*host*, *prognum*, *versnum*, *procnum*,
inproc, *in*, *outproc*, *out*)
char **host*;
u_long *prognum*, *versnum*, *procnum*;
xdrproc_t *inproc*;
char * *in*;
xdrproc_t *outproc*;
char **out*;

Description

The **callrpc** subroutine calls a remote procedure identified by the *prognum* parameter, the *versnum* parameter, and the *procnum* parameter on the machine pointed to by the *host* parameter.

This subroutine uses User Datagram Protocol/Internet Protocol (UDP/IP) as a transport to call a remote procedure. No connection will be made if the server is supported by Transmission Control Protocol/Internet Protocol (TCP/IP). This subroutine does not control time outs or authentication.

Parameters

host	Points to the program name of the remote machine.
prognum	Specifies the number of the remote program.
versnum	Specifies the version number of the remote program.
procnum	Specifies the number of the procedure associated with the remote program being called.
inproc	Specifies the name of the XDR procedure that encodes the procedure parameters.
in	Specifies the address of the procedure arguments.
outproc	Specifies the name of the XDR procedure that decodes the procedure results.
out	Specifies the address where results are placed.

Return Values

This subroutine returns a value of **enum clnt_stat**. Use the **clnt_perrno** subroutine to translate this failure status into a displayed message.

Implementation Specifics

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

Related information

The **clnt_call** macro.

The **clnt_broadcast** subroutine, **clnttcp_create** subroutine, **clntudp_create** subroutine, **clnt_perrno** subroutine, **registerrpc** subroutine, **svc_run** subroutine.

Using the callrpc Routine in *Communications Programming Concepts*.

Understanding Protocols for TCP/IP, User Datagram Protocol in *Communication Concepts and Procedures*.

clnt_broadcast Subroutine

Purpose

Broadcasts a remote procedure call to all locally connected networks.

Library

C Library (**libc.a**)

Syntax

#include <rpc/rpc.h>

enum clnt_stat
clnt_broadcast (*prognum*, *versnum*, *procnum*,
inproc, *in*, *outproc*, *out*, *eachresult*)
u_long *prognum*, *versnum*, *procnum*;
xdrproc_t *inproc*;
char **in*;
xdrproc_t *outproc*;
char **out*;
resultproc_t *eachresult*;

Description

The **clnt_broadcast** subroutine broadcasts a remote procedure call to all locally connected networks. The remote procedure is identified by the *prognum*, *versnum*, and *procnum* parameters on the workstation identified by the *host* parameter.

Broadcast sockets are limited in size to the maximum transfer unit of the data link. For Ethernet, this value is 1500 bytes.

When a client broadcasts a remote procedure call over the network, a number of server processes respond. Each time the client receives a response, the **clnt_broadcast** subroutine calls the **eachresult** routine. The **eachresult** routine takes the following form:

eachresult (*out*, **addr*)
char **out*;
struct sockaddr_in **addr*;

Parameters

prognum	Specifies the number of the remote program.
versnum	Specifies the version number of the remote program.
procnum	Identifies the procedure to be called.
inproc	Specifies the procedure that encodes the procedure's parameters.
in	Specifies the address of the procedure's arguments.
outproc	Specifies the procedure that decodes the procedure results.
out	Specifies the address where results are placed.
eachresult	Specifies the procedure to call when clients respond.
addr	Specifies the address of the workstation that sent the results.

Return Values

If the **eachresult** subroutine returns a value of 0, the **clnt_broadcast** subroutine waits for more replies. Otherwise, the **clnt_broadcast** subroutine returns with the appropriate results.

Implementation Specifics

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

Related Information

The **callrpc** subroutine.

Sockets Overview in *Communications Programming Concepts.*

clnt_call Macro

Purpose

Calls the remote procedure associated with the *clnt* parameter.

Library

C Library (**libc.a**)

Syntax

#include <rpc/rpc.h>
enum clnt_stat

clnt_call (*clnt*, *procnum*, *inproc*, *in*, *outproc*, *out*, *tout***)**
CLIENT **clnt*;
u_long *procnum*;
xdrproc_t *inproc*;
char **in*;
xdrproc_t *outproc*;
char **out*;
struct timeval *tout*;

Description

The **clnt_call** macro calls the remote procedure associated with the client handle pointed to by the *clnt* parameter.

Parameters

clnt	Points to the structure of the client handle that results from a Remote Procedure Call (RPC) client creation subroutine, such as the **clntudp_create** subroutine that opens a User Datagram Protocol/Internet Protocol (UDP/IP) socket.
procnum	Identifies the remote procedure on the host machine.
inproc	Specifies the procedure that encodes the procedure's parameters.
in	Specifies the address of the procedure's arguments.
outproc	Specifies the procedure that decodes the procedure's results.
out	Specifies the address where results are placed.
tout	Sets the time allowed for results to return.

Implementation Specifics

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

Related Information

The **callrpc** subroutine, **clnt_perror** subroutine, **clnttcp_create** subroutine, **clntudp_create** subroutine.

Sockets Overview in *Communications Programming Concepts*.

User Datagram Protocol (UDP) in *Communication Concepts and Procedures*.

clnt_control Macro

Purpose

Changes or retrieves various information about a client object.

Library

C Library (**libc.a**)

Syntax

#include <rpc/rpc.h>

bool_t
clnt_control (*cl*, *req*, *info*)
CLIENT **cl*;
int *req*
char **info*;

Description

The **clnt_control** macro is used to change or retrieve various information about a client object.

User Datagram Protocol (UDP) and Transmission Control Protocol (TCP) have the following supported values for the *req* parameter's argument types and functions:

Values for the req Parameter	Argument Type	Function
CLSET_TIMEOUT	struct timeval	Sets total time out
CLGET_TIMEOUT	struct timeval	Gets total time out

Note: If the time out is set using the **clnt_control** subroutine, the timeout parameter passed to the **clnt_call** subroutine will be ignored in all future calls.

CLGET_SERVER_ADDR	struct sockaddr	Gets server's address

The following operations are valid for UDP only:

CLSET_RETRY_TIMEOUT	struct timeval	Sets the retry time out
CLGET_RETRY_TIMEOUT	struct timeval	Gets the retry time out

Note: The retry time out is the time that User Datagram Protocol/Remote Procedure Call (UDP/RPC) waits for the server to reply before retransmitting the request.

Parameters

cl Points to the structure of the client handle.

req Indicates the type of operation.

info Points to the information for request type.

Return Values

Upon successful completion, this subroutine returns a value of 1. If unsuccessful, it returns a value of 0.

Implementation Specifics

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

Related Information

The **clnt_call** macro.

The **clnttcp_create** subroutine, **clntudp_create** subroutine.

Understanding Protocols for TCP/IP, User Datagram Protocol (UDP) in *Communication Concepts and Procedures*.

clnt_create Subroutine

Purpose

Creates and returns a generic client handle.

Library

C Library (**libc.a**)

Syntax

#include <rpc/rpc.h>

CLIENT *
clnt_create (*host*, *prognum*, *versnum*, *protocol*)
char **host*;
unsigned *prognum, versnum*;
char **protocol*;

Description

Creates and returns a generic client handle.

RPC messages transported by UDP/IP can hold up to 8K bytes of encoded data. Use this transport for procedures that take arguments or return results of less than 8K bytes.

Parameters

host	Identifies the name of the remote host where the server is located.
prognum	Specifies the program number of the remote program.
versnum	Specifies the version number of the remote program.
protocol	Identifies which data transport protocol the program is using (UDP or TCP).

Return Values

Upon successful completion, this subroutine returns a client handle.

Implementation Specifics

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

Related Information

The **clnt_control** macro, **clnt_destroy** macro.

The **clnttcp_create** subroutine, **clntudp_create** subroutine.

Understanding Protocols for TCP/IP, User Datagram Protocol (UDP) in *Communication Concepts and Procedures.*

clnt_destroy Macro

Purpose

Destroys the client's RPC handle.

Library

C Library (**libc.a**)

Syntax

#include <rpc/rpc.h>

void
clnt_destroy (*clnt*)
CLIENT **clnt*;

Description

The **clnt_destroy** macro destroys the client's Remote Procedure Call (RPC) handle. Destroying the client's RPC handle deallocates private data structures, including the *clnt* parameter itself. The use of the *clnt* parameter becomes undefined upon calling the **clnt_destroy** macro.

Parameter

clnt	Points to the structure of the client handle.

Implementation Specifics

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

Related Information

The **clntudp_create** subroutine, **clnt_create** subroutine.

Sockets Overview in *Communications Programming Concepts.*

clnt_freeres Macro

Purpose

Frees data that was allocated by the RPC/XDR system.

Library

C Library (**libc.a**)

Syntax

#include <rpc/rpc.h>

clnt_freeres (*clnt*, *outproc*, *out*)
CLIENT **clnt*;
xdrpoc_t *outproc*;
char **out*;

Description

The **clnt_freeres** macro frees data allocated by the Remote Procedure Call/eXternal Data Representation (RPC/XDR) system. This data was allocated when the RPC/XDR system decoded the results of an RPC call.

Parameters

clnt — Points to the structure of the client handle.

outproc — Specifies the XDR subroutine that describes the results in simple decoding primitives.

out — Specifies the address where the results are placed.

Implementation Specifics

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

Related Information

eXternal Data Representation (XDR) Overview for Programming in *Communications Programming Concepts.*

clnt_geterr Macro

Purpose

Copies error information from a client handle.

Library

C Library (**libc.a**)

Syntax

#include <rpc/rpc.h>

void
clnt_geterr (*clnt*, *errp***)**
**CLIENT ****clnt*;
**struct rpc_err ****errp*;

Description

The **clnt_geterr** macro copies error information from a client handle to an error structure.

Parameters

clnt	Points to the structure of the client handle.
errp	Specifies the address of the error structure.

Implementation Specifics

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

clnt_pcreateerror Subroutine

Purpose

Indicates why a client RPC handle was not created.

Library

C Library (**libc.a**)

Syntax

#include <rpc/rpc.h>

void
clnt_pcreateerror (*s*)
char **s*;

Description

The **clnt_pcreateerror** subroutine writes a message to standard error output, indicating why a client Remote Procedure Call (RPC) handle could not be created. The message is preceded by the string pointed to by the *s* parameter and a colon.

Use this subroutine if one of the following calls fails: the **clntraw_create** subroutine, **clnttcp_create** subroutine, or **clntudp_create** subroutine.

Parameters

s Points to a character string that represents the error text.

Implementation Specifics

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

Related Information

The **clnt_create** subroutine, **clntraw_create** subroutine, **clnttcp_create** subroutine, **clntudp_create** subroutine, **clnt_spcreateerror** subroutine.

clnt_perrno Subroutine

Purpose

Specifies the condition of the *stat* parameter.

Library

C Library (**libc.a**)

Syntax

#include <rpc/rpc.h>

void
clnt_perrno (*stat***)**
enum clnt_stat *stat*;

Description

The **clnt_perrno** subroutine writes a message to standard error output, corresponding to the condition specified by the *stat* parameter.

This subroutine is used after a **callrpc** subroutine fails. The **clnt_perrno** subroutine translates the failure status (the **enum clnt_stat** subroutine) into a message.

If the program does not have a standard error output, or the programmer does not want the message to be output with the **printf** subroutine, or the message format used is different from that supported by the **clnt_perrno** subroutine, then the **clnt_sperrno** subroutine is used instead of the **clnt_perrno** subroutine.

Parameters

stat	Specifies the client error status of the remote procedure call.

Return Values

The **clnt_perrno** subroutine translates and displays the following **enum clnt_stat** error status codes:

RPC_SUCCESS = 0	Call succeeded.
RPC_CANTENCODEARGS = 1	Cannot decode arguments.
RPC_CANTDECODERES = 2	Cannot decode results.
RPC_CANTSEND = 3	Failure in sending call.
RPC_CANTRECV = 4	Failure in receiving result.
RPC_TIMEDOUT = 5	Call timed out.

Implementation Specifics

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

Related Information

The **callrpc** subroutine, **clnt_sperrno** subroutine.

clnt_perror Subroutine

Purpose

Indicates why a remote procedure call failed.

Library

C Library (**libc.a**)

Syntax

#include <rpc/rpc.h>

clnt_perror (*clnt, s*)
CLIENT **clnt*;
char **s*;

Description

The **clnt_perror** subroutine writes a message to standard error output indicating why a remote procedure call failed. The message is prepended with the string pointed to by the *s* parameter and a colon.

This subroutine is used after the **clnt_call** macro.

Parameters

clnt	Points to the structure of the client handle.
s	Points to a character string that represents the error text.

Return Values

This subroutine returns an error string to standard error output.

Implementation Specifics

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

Related Information

The **clnt_call** macro.

The **clnt_sperror** subroutine.

clnt_spcreateerror Subroutine

Purpose

Indicates why a client RPC handle was not created.

Library

C Library (**libc.a**)

Syntax

#include <rpc/rpc.h>

char *
clnt_spcreateerror (*s*)
**char ****s*;

Description

The **clnt_spcreateerror** subroutine returns a string indicating why a client Remote Procedure Call (RPC) handle was not created.

Note: This subroutine returns the pointer to static data that is overwritten on each call.

Parameters

s	Points to a character string that represents the error text.

Implementation Specifics

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

Related Information

The **clnt_pcreateerror** subroutine.

clnt_sperrno Subroutine

Purpose

Specifies the condition of the *stat* parameter by returning a pointer to a string containing a status message.

Library

C Library (**libc.a**)

Syntax

#include <rpc/rpc.h>

char *
clnt_sperrno (*stat***)**
enum clnt_stat *stat***;**

Description

The **clnt_sperrno** subroutine specifies the condition of the *stat* parameter by returning a pointer to a string containing a status message. The string ends with a new-line character.

Whenever one of the following conditions exists, the **clnt_sperrno** subroutine is used instead of the **clnt_perrno** subroutine when a **callrpc** routine fails:

- The program does not have a standard error output. This is common for programs running as servers.
- The programmer does not want the message to be output with the **printf** subroutine.
- A message format differing from that supported by the **clnt_perrno** subroutine is being used.

Note: The **clnt_sperrno** subroutine does not return the pointer to static data, so the result is not overwritten on each call.

Parameters

stat	Specifies the client error status of the remote procedure call.

Return Values

The **clnt_sperrno** subroutine translates and displays the following **enum clnt_stat** error status messages:

RPC_SUCCESS = 0	Call succeeded.
RPC_CANTENCODEARGS = 1	Cannot decode arguments.
RPC_CANTDECODERES = 2	Cannot decode results.
RPC_CANTSEND = 3	Failure in sending call.
RPC_CANTRECV = 4	Failure in receiving result.
RPC_TIMEDOUT = 5	Call timed out.

Implementation Specifics

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

Related Information

The **clnt_perrno** subroutine.

clnt_sperror Subroutine

Purpose

Indicates why a remote procedure call failed.

Library

C Library (**libc.a**)

Syntax

#include <rpc/rpc.h>

char *
clnt_sperror (*cl*,*s*)
CLIENT **cl*;
char **s*;

Description

The **clnt_sperror** subroutine returns a string to standard error output indicating why a Remote Procedure Call (RPC) call failed. This subroutine also returns the pointer to static data overwritten on each call.

Parameters

cl	Points to the structure of the client handle.
s	Points to a character string that represents the error text.

Return Values

This subroutine returns an error string to standard error output.

Implementation Specifics

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

Related Information

The **clnt_perror** subroutine.

clntraw_create Subroutine

Purpose

Creates a toy RPC client for simulation.

Library

C Library (**libc.a**)

Syntax

#include <rpc/rpc.h>

CLIENT *
clntraw_create (*prognum*, *versnum***)**
u_long *prognum*, *versnum*;

Description

The **clntraw_create** subroutine creates a toy Remote Procedure Call (RPC) client for simulation of a remote program. This toy client uses a buffer located within the address space of the process for the transport to pass messages to the service. If the corresponding RPC server lives in the same address space, simulation of RPC and acquisition of RPC overheads, such as round-trip times, are done without kernel interference.

Parameters

prognum Specifies the program number of the remote program.

versnum Specifies the version number of the remote program.

Return Values

Upon successful completion, this subroutine returns a pointer to a valid RPC client. If unsuccessful, it returns a value of NULL.

Implementation Specifics

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

Related Information

The **clnt_pcreateerror** subroutine, **svcraw_create** subroutine.

clnttcp_create Subroutine

Purpose

Creates a TCP/IP client transport handle.

Library

C Library (**libc.a**)

Syntax

CLIENT *
clnttcp_create (*addr*, *prognum*, *versnum*, *sockp*, *sendsz*, *recvsz*)
struct sockaddr_in **addr*;
u_long *prognum*, *versnum*;
int **sockp*;
u_int *sendsz*, *recvsz*;

Description

The **clnttcp_create** subroutine creates a Remote Procedure Call (RPC) client transport handle for a remote program. This client uses Transmission Control Protocol/Internet Protocol (TCP/IP) as the transport to pass messages to the service.

The TCP/IP remote procedure calls use buffered input/output (I/O). Users can set the size of the send and receive buffers with the *sendsz* and *recvsz* parameters. If the size of either buffer is set to a value of 0, the **svctcp_create** subroutine picks suitable default values.

Parameters

addr	Points to the Internet address of the remote program. If the port number for this Internet address (**addr->sin_port**) is a value of 0, then the *addr* parameter is set to the actual port on which the remote program is listening. The client making the remote procedure call consults the remote **portmap** daemon to obtain the port information.
prognum	Specifies the program number of the remote program.
versnum	Specifies the version number of the remote program.
sockp	Specifies a pointer to a socket. If the value of the *sockp* parameter is RPC_ANYSOCK, the **clnttcp_create** subroutine opens a new socket and sets the *sockp* pointer to the new socket.
sendsz	Sets the size of the send buffer.
recvsz	Sets the size of the receive buffer.

Return Values

Upon successful completion, this routine returns a valid TCP/IP client handle. If unsuccessful, it returns a value of NULL.

Implementation Specifics

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

Related Information

The **portmap** daemon.

The **clnt_call** macro.

The **callrpc** subroutine, **clntudp_create** subroutine, **svctcp_create** subroutine, **clnt_pcreateerror** subroutine.

Sockets Overview in *Communications Programming Concepts.*

Understanding Protocols for TCP/IP in *Communication Concepts and Procedures.*

clntudp_create Subroutine

Purpose

Creates a UDP/IP client transport handle.

Library

C Library (**libc.a**)

Syntax

#include <rpc/rpc.h>

CLIENT *
clntudp_create (*addr*, *prognum*, *versnum*, *wait*, *sockp* **)**
struct sockaddr_in **addr*;
u_long *prognum*, *versnum*;
struct timeval *wait*;
int **sockp*;

Description

The **clntudp_create** subroutine creates a Remote Procedure Call (RPC) client transport handle for a remote program. The client uses User Datagram Protocol/Internet Protocol (UDP/IP) as the transport to pass messages to the service.

RPC messages transported by UDP/IP can hold up to 8K bytes of encoded data. Use this subroutine for procedures that take arguments or return results of less than 8K bytes.

Parameters

addr — Points to the Internet address of the remote program. If the port number for this Internet address (**addr->sin_port**) is 0, then the value of the *addr* parameter is set to the port that the remote program is listening on. The **clntudp_create** subroutine consults the remote **portmap** daemon for this information.

prognum — Specifies the program number of the remote program.

versnum — Specifies the version number of the remote program.

wait — Sets the amount of time that the UDP/IP transport waits to receive a response before the transport sends another remote procedure call or the remote procedure call times out. The total time for the call to time out is set by the **clnt_call** macro.

sockp — Specifies a pointer to a socket. If the value of the *sockp* parameter is RPC_ANYSOCK, the **clntudp_create** subroutine opens a new socket and sets the *sockp* pointer to that new socket.

Return Values

Upon successful completion, this subroutine returns a valid UDP client handle. If unsuccessful, it returns a value of NULL.

Implementation Specifics

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

Related Information

The **portmap** daemon.

The **clnt_call** macro.

The **callrpc** subroutine, **clnt_pcreateerror** subroutine, **clnttcp_create** subroutine, **svcudp_create** subroutine.

Sockets Overview in *Communications Programming Concepts.*

User Datagram Protocol (UDP) in *Communication Concepts and Procedures.*

dbm_close Subroutine

Purpose

Closes a database.

Library

C Library **(libc.a)**

Syntax

#include <ndbm.h>
void dbm_close (*db*)
DBM **db*;

Description

The **dbm_close** subroutine closes a database.

Parameter

db	Specifies the database to close.

Implementation Specifics

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

Related Information

The **dbmclose** subroutine.

dbm_delete Subroutine

Purpose

Deletes a key and its associated contents.

Library

C Library **(libc.a)**

Syntax

#include <ndbm.h>
int dbm_delete (*db*, *key***)**
DBM **db*;
datum *key*;

Description

The **dbm_delete** subroutine deletes a key and its associated contents.

Parameters

db	Specifies a database.
key	Specifies the key to delete.

Return Values

Upon successful completion, this subroutine returns a value of 0 (zero). If unsuccessful, it returns a negative value.

Implementation Specifics

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

Related Information

The **delete** subroutine.

dbm_fetch Subroutine

Purpose

Accesses data stored under a key.

Library

C Library **(libc.a)**

Syntax

#include <ndbm.h>
datum dbm_fetch(*db*, *key***)**
DBM ****db*;**
datum *key***;**

Description

The **dbm_fetch** subroutine accesses data stored under a key.

Parameters

db	Specifies the database to access.
key	Specifies the input key.

Return Values

Upon successful completion, this subroutine returns a **datum** structure containing the value returned for the specified key. If it is unsuccessful, the dptr field of the **datum** structure is set to NULL.

Implementation Specifics

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

Related Information

The **fetch** subroutine.

dbm_firstkey Subroutine

Purpose

Returns the first key in a database.

Library

C Library **(libc.a)**

Syntax

#include <ndbm.h>
datum dbm_firstkey (*db***)**
DBM **db*;

Description

The **dbm_firstkey** subroutine returns the first key in a database.

Parameter

db	Specifies the database to access.

Return Values

Upon successful completion, this subroutine returns a **datum** structure containing the value for the first key. If it is unsuccessful, the dptr field of the **datum** structure is set to NULL.

Implementation Specifics

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

Related Information

The **firstkey** subroutine.

dbm_nextkey Subroutine

Purpose

Returns the next key in a database.

Library

C Library **(libc.a)**

Syntax

#include <ndbm.h>
datum dbm_nextkey (*db*)
**DBM ****db*;

Description

The **dbm_nextkey** subroutine returns the next key in a database.

Parameter

db	Specifies the database to access.

Return Values

Upon successful completion, this subroutine returns a **datum** structure containing the value for the next key. If it is unsuccessful, the dptr field of the **datum** structure is set to NULL.

Implementation Specifics

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

Related Information

The **nextkey** subroutine.

dbm_open Subroutine

Purpose

Opens a database for access.

Library

C Library (**libc.a**)

Syntax

#include <ndbm.h>
DBM *dbm_open (*file*, *flags*, *mode***)**
**char ****file*;
int *flags*, *mode*;

Description

The **dbm_open** subroutine opens a database for access. This opens and/or creates the **file.dir** and **file.pag** files, depending on the flags parameter. The returned DBM structure is used as input to other NDBM routines.

Parameters

file	Specifies the path to open a database.
flags	Specifies the flags required to open a subroutine.
mode	Specifies the mode required to open a subroutine.

Return Values

Upon successful completion, this subroutine returns a pointer to the DBM structure. If unsuccessful, it returns a value of NULL.

Implementation Specifics

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

Related Information

The **dbminit** subroutine.

dbm_store Subroutine

Purpose

Places data under a key.

Library

C Library **(libc.a)**

Syntax

#include <ndbm.h>
int dbm_store (*db*, *key*, *content*, *flags*)
DBM **db*;
datum *key*, *content*;
int *flags*;

Description

The **dbm_store** subroutine places data under a key.

Parameters

db	Specifies the database to store.
key	Specifies the input key.
content	Specifies the value associated with the key to store.
flags	Contains either DBM_INSERT or DBM_REPLACE. If the **dbm_store** subroutine is called with *flags* set to DBM_INSERT, and if an entry for the key already exists, then the **dbm_store** subroutine returns a value of 1. If the *flags* parameter is set to DBM_REPLACE then the entry will be replaced if it already exists.

Return Values

Upon successful completion, this subroutine returns a value of 0 (zero). If unsuccessful, it returns a negative value. If the **dbm_store** subroutine is called with the *flags* parameter set to DBM_INSERT and an existing entry is found, then it returns a value of 1.

Implementation Specifics

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

Related Information

The **store** subroutine.

dbmclose Subroutine

Purpose

Closes a database.

Library

DBM Library **(libdbm.a)**

Syntax

#include <dbm.h>
void dbmclose (*db*)
DBM **db*;

Description

The **dbmclose** subroutine closes a database.

Parameter

db	Specifies the database to close.

Implementation Specifics

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

Related Information

The **dbm_close** subroutine.

dbminit Subroutine

Purpose

Opens a database for access.

Library

DBM Library **(libdbm.a)**

Syntax

#include <dbm.h>
dbminit (*file*)
char **file*;

Description

The **dbminit** subroutine opens a database for access. At the time of the call, the **file.dir** and **file.pag** files must exist.

Note: To build an empty database, create zero-length **.dir** and **.pag** files.

Parameter

file	Specifies the path name of the database to open.

Return Values

Upon successful completion, this subroutine returns a value of 0 (zero). If unsuccessful, it returns a negative value.

Implementation Specifics

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

Related Information

The **dbm_open** subroutine.

delete Subroutine

Purpose

Deletes a key and its associated contents.

Library

DBM Library **(libdbm.a)**

Syntax

#include <dbm.h>
delete (*key*)
datum *key*;

Description

The **delete** subroutine deletes a key and its associated contents.

Parameter

key	Specifies the key to delete.

Return Values

Upon successful completion, this subroutine returns a value of 0 (zero). If unsuccessful, it returns a negative value.

Implementation Specifics

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

Related Information

The **dbm_delete** subroutine.

fetch Subroutine

Purpose

Accesses data stored under a key.

Library

DBM Library **(libdbm.a)**

Syntax

#include <dbm.h>
datum fetch (*key***)**
datum *key*;

Description

The **fetch** subroutine accesses data stored under a key.

Parameter

key	Specifies the input key.

Return Values

Upon successful completion, this subroutine returns data corresponding to the specified key. If it is unsuccessful, a NULL value is indicated in the dptr field of the **datum** structure.

Implementation Specifics

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

Related Information

The **dbm_fetch** subroutine.

firstkey Subroutine

Purpose

Returns the first key in the database.

Library

DBM Library **(libdbm.a)**

Syntax

#include <dbm.h>
datum firstkey ()

Description

The **firstkey** subroutine returns the first key in the database.

Parameters

This subroutine contains no parameters.

Return Values

Returns a **datum** structure containing the first key value pair.

Implementation Specifics

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

Related Information

The **dbm_firstkey** subroutine.

get_myaddress Subroutine

Purpose

Gets the user's IP address.

Library

C Library (**libc.a**)

Syntax

#include <rpc/rpc.h>

void
get_myaddress (*addr*)
struct sockaddr_in **addr*;

Description

The **get_myaddress** subroutine gets the machine's Internet Protocol (IP) address without consulting the library routines that access the **/etc/hosts** file.

Parameter

addr — Specifies the address where the machine's IP address is placed. The port number is set to a value of `htons` (PMAPPORT).

Implementation Specifics

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

Related Information

The **/etc/hosts** file.

Internet Protocol (IP) Overview in *Communication Concepts and Procedures*.

getnetname Subroutine

Purpose

Installs the network name of the caller in the array specified by the *name* parameter.

Library

C Library (**libc.a**)

Syntax

#include <rpc/rpc.h>

getnetname (*name*)
char *name* [**MAXNETNAMELEN**];

Description

The **getnetname** subroutine installs the caller's unique, operating-system-independent network name in the fixed-length array specified by the *name* parameter.

Parameter

name	Specifies the network name (or netname) of the server process owner. The *name* parameter can be either the host name derived from the **host2netname** subroutine or the user name derived from the **user2netname** subroutine.

Return Values

Upon successful completion, this subroutine returns a value of 1. If unsuccessful, it returns a value of 0.

Implementation Specifics

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

Related Information

The **host2netname** subroutine, **user2netname** subroutine.

host2netname Subroutine

Purpose

Converts a domain-specific host name to an operating-system-independent network name.

Library

C Library (**libc.a**)

Syntax

#include <rpc/rpc.h>

host2netname (*name*, *host*, *domain*)
char **name*;
char **host*
char **domain*

Description

The **host2netname** subroutine converts a domain-specific host name to an operating-system-independent network name.

This subroutine is the inverse of the **netname2host** subroutine.

Parameters

name	Points to the network name (or netname) of the server process owner. The *name* parameter can be either the host name derived from the **host2netname** subroutine or the user name derived from the **user2netname** subroutine.
host	Points to the name of the machine on which the permissions were created.
domain	Points to the domain name.

Return Values

Upon successful completion, this subroutine returns a value of 1. If unsuccessful, it returns a value of 0.

Implementation Specifics

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

Related Information

The **netname2host** subroutine, **user2netname** subroutine.

key_decryptsession Subroutine

Purpose

Decrypts a server network name and a DES key.

Library

C Library (**libc.a**)

Syntax

key_decryptsession (*remotename*, *deskey*)
char **remotename*;
des_block **deskey*;

Description

The **key_decryptsession** subroutine interfaces to the **keyserv** daemon, which is associated with the secure authentication system known as Data Encryption Standard (DES). The subroutine takes a server network name and a DES key and decrypts the DES key by using the public key of the server and the secret key associated with the effective user number (UID) of the calling process. User programs rarely need to call this subroutine. System commands such as **keylogin** and the Remote Procedure Call (RPC) library are the main clients.

This subroutine is the inverse of the **key_encryptsession** subroutine.

Parameters

remotename	Points to the remote host name.
deskey	Points to the **des_block** structure.

Return Values

Upon successful completion, this subroutine returns a value of 0. If unsuccessful, it returns a value of –1.

Implementation Specifics

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

Related Information

The **keylogin** command.

The **keyserv** daemon.

The **key_encryptsession** subroutine.

key_encryptsession Subroutine

Purpose

Encrypts a server network name and a DES key.

Library

C Library (**libc.a**)

Syntax

#include <rpc/rpc.h>

key_encryptsession (*remotename*, *deskey*)
char **remotename*;
des_block **deskey*;

Description

The **key_encryptsession** subroutine interfaces to the **keyserv** daemon, which is associated with the secure authentication system know as Data Encryption Standard (DES). This subroutine encrypts a server network name and a DES key. To do so, the routine uses the public key of the server and the secret key associated with the effective user number (UID) of the calling process. System commands such as **keylogin** and the Remote Procedure Call (RPC) library are the main clients. User programs rarely need to call this subroutine.

This subroutine is the inverse of the **key_decryptsession** subroutine.

Parameters

remotename	Points to the remote host name.
deskey	Points to the **des_block** structure.

Return Values

Upon successful completion, this subroutine returns a value of 0. If unsuccessful, it returns a value of –1.

Implementation Specifics

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

Related Information

The **keylogin** command.

The **keyserv** daemon.

The **key_decryptsession** subroutine.

key_gendes Subroutine

Purpose

Asks the **keyserv** daemon for a secure conversation key.

Library

C Library (**libc.a**)

Syntax

#include <rpc/rpc.h>

key_gendes (*deskey*)
des_block **deskey*;

Description

The **key_gendes** subroutine interfaces to the **keyserv** daemon, which is associated with the secure authentication system know as Data Encryption Standard (DES). This subroutine asks the **keyserv** daemon for a secure conversation key. Choosing a key at random is not recommended because the common ways of choosing random numbers, such as the current time, are easy to guess. User programs rarely need to call this subroutine. System commands such as **keylogin** and the Remote Procedure Call (RPC) library are the main clients.

Parameters

deskey	Points to the **des_block** structure.

Return Values

Upon successful completion, this subroutine returns a value of 0. If unsuccessful, it returns a value of –1.

Implementation Specifics

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

Related Information

The **keylogin** command.

The **keyserv** daemon.

key_setsecret Subroutine

Purpose

Sets the key for the effective UID of the calling process.

Library

C Library (**libc.a**)

Syntax

#include <rpc/rpc.h>

key_setsecret (*key*)
char **key*;

Description

The **key_setsecret** subroutine interfaces to the **keyserv** daemon, which is associated with the secure authentication system know as Data Encryption Standard (DES). This subroutine is used to set the key for the effective user number (UID) of the calling process. User programs rarely need to call this subroutine. System commands such as **keylogin** and the Remote Procedure Call (RPC) library are the main clients.

Parameters

key	Points to the key name.

Return Values

Upon successful completion, this subroutine returns a value of 0. If unsuccessful, it returns a value of –1.

Implementation Specifics

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

Related Information

The **keylogin** command.

The **keyserv** daemon.

netname2host Subroutine

Purpose

Converts an operating-system-independent network name to a domain-specific host name.

Library

C Library (**libc.a**)

Syntax

#include <rpc/rpc.h>

netname2host (*name*, *host*, *hostlen*)
char **name*;
char **host*;
int *hostlen*;

Description

The **netname2host** subroutine converts an operating-system-independent network name to a domain-specific host name.

This subroutine is the inverse of the **host2netname** subroutine.

Parameters

name	Specifies the network name (or netname) of the server process owner. The *name* parameter can be either the host name derived from the **host2netname** subroutine or the user name derived from the **user2netname** subroutine.
host	Points to the name of the machine on which the permissions were created.
hostlen	Specifies the size of the host name.

Return Values

Upon successful completion, this subroutine returns a value of 1. If unsuccessful, it returns a value of 0.

Implementation Specifics

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

Related Information

The **host2netname** subroutine, **user2netname** subroutine.

netname2user Subroutine

Purpose

Converts from an operating-system-independent network name to a domain-specific UID.

Library

C Library (**libc.a**)

Syntax

#include <rpc/rpc.h>

netname2user (*name*, *uidp*, *gidp*, *gidlenp*, *gidlist*)
char **name*;
int **uidp*;
int **gidp*;
int **gidlenp*;
int **gidlist*;

Description

The **netname2user** subroutine converts from an operating-system-independent network name to a domain-specific user number (UID). This subroutine is the inverse of the **user2netname** subroutine.

Parameters

name	Points to the network name (or netname) of the server process owner. The *name* parameter can be either the host name derived from the **host2netname** subroutine or the user name derived from the **user2netname** subroutine.
uidp	Points to the user ID.
gidp	Points to the group ID.
gidlenp	Points to the size of the group ID.
gidlist	Points to the group list.

Return Values

Upon successful completion, this subroutine returns a value of 1. If unsuccessful, it returns a value of 0.

Implementation Specifics

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

Related Information

The **host2netname** subroutine, **user2netname** subroutine.

nextkey Subroutine

Purpose

Returns the next key in a database.

Library

DBM Library **(libdbm.a)**

Syntax

#include <dbm.h>
datum nextkey (*key*)
datum *key*;

Description

The **nextkey** subroutine returns the next key in a database.

Parameters

key Specifies the input key. This value has no effect on the return value but must be present.

Return Values

Returns a **datum** structure containing the next key-value pair.

Implementation Specifics

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

Related Information

The **dbm_nextkey** subroutine.

pmap_getmaps Subroutine

Purpose

Returns a list of the current RPC program to port mappings on the host.

Library

C Library (**libc.a**)

Syntax

#include <rpc/rpc.h>

struct pmaplist *
pmap_getmaps (*addr*)
struct sockaddr_in **addr*;

Description

The **pmap_getmaps** subroutine acts as a user interface to the **portmap** daemon. The subroutine returns a list of the current Remote Procedure Call (RPC) program to port mappings on the host located at the Internet Protocol (IP) address pointed to by the *addr* parameter.

Note: The **rpcinfo –p** command calls this subroutine.

Parameter

addr	Specifies the address where the machine's IP address is placed.

Return Value

If there is no list of current RPC programs, this procedure returns a value of NULL.

Implementation Specifics

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

Related Information

The **rpcinfo** command.

The **portmap** daemon.

The **pmap_set** subroutine, **pmap_unset** subroutine, **svc_register** subroutine.

pmap_getport Subroutine

Purpose

Requests the port number on which a service waits.

Library

C Library (**libc.a**)

Syntax

#include <rpc/rpc.h>

u_short
pmap_getport (*addr*, *prognum*, *versnum*, *protocol***)**
**struct sockaddr_in ****addr*;
u_long *prognum*, *versnum*, *protocol*;

Description

The **pmap_getport** subroutine acts as a user interface to the **portmap** daemon in order to return the port number on which a service waits.

Parameters

addr	Points to the Internet Protocol (IP) address of the host where the remote program that supports the waiting service resides.
prognum	Specifies the program number of the remote program.
versnum	Specifies the version number of the remote program.
protocol	Specifies the transport protocol that the service recognizes.

Return Values

If the mapping does not exist or the Remote Procedure Call (RPC) system could not contact the remote **portmap** daemon, this subroutine returns a value of 0. If the remote **portmap** daemon could not be contacted, the **rpc_createerr** subroutine contains the RPC status.

Implementation Specifics

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

Related Information

The **portmap** daemon.

Internet Protocol (IP) in *Communication Concepts and Procedures.*

pmap_rmtcall Subroutine

Purpose

Instructs the **portmap** daemon to make a remote procedure call.

Library

C Library (**libc.a**)

Syntax

#include <rpc/rpc.h>

enum clnt_stat
pmap_rmtcall (*addr*, *prognum*, *versnum*, *procnum*,
inproc, *in*, *outproc*, *out*, *tout*, *portp*)
struct sockaddr_in **addr*;
u_long *prognum*, *versnum*, *procnum*;
xdrproc_t *inproc*;
char **in*;
xdrproc_t *outproc*;
char **out*;
struct timeval *tout*;
u_long **portp*;

Description

The **pmap_rmtcall** subroutine is a user interface to the **portmap** daemon. The routine instructs the host **portmap** daemon to make a remote procedure call. Clients consult the **portmap** daemon when sending out Remote Procedure Call (RPC) calls for given program numbers. The **portmap** daemon tells the client the ports to which to send the calls.

Parameters

addr	Points to the Internet Protocol (IP) address of the host where the remote program that supports the waiting service resides.
prognum	Specifies the program number of the remote program.
versnum	Specifies the version number of the remote program.
procnum	Identifies the procedure to be called.
inproc	Specifies the eXternal Data Representation (XDR) routine that encodes the remote procedure parameters.
in	Points to the address of the procedure arguments.
outproc	Specifies the XDR routine that decodes the remote procedure results.
out	Points to the address where the results are placed.
tout	Sets the time the routine waits for the results to return before sending the call again.
portp	Points to the program port number if the procedure succeeds.

Implementation Specifics

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

Related Information

The **portmap** daemon.

The **clnt_broadcast** subroutine.

eXternal Data Representation (XDR) Overview for Programming in *Communications Programming Concepts*.

Internet Protocol (IP) in *Communication Concepts and Procedures*.

pmap_set Subroutine

Purpose

Maps a remote procedure call to a port.

Library

C Library (**libc.a**)

Syntax

#include <rpc/rpc.h>

pmap_set (*prognum*, *versnum*, *protocol*, *port*)
u_long *prognum*, *versnum*, *protocol*;
u_short *port*;

Description

The **pmap_set** subroutine acts as a user interface to the **portmap** daemon to map the program number, version number, and protocol of a remote procedure call to a port on the machine **portmap** daemon.

Note: The **pmap_set** subroutine is called by the **svc_register** subroutine.

Parameters

prognum	Specifies the program number of the remote program.
versnum	Specifies the version number of the remote program.
protocol	Specifies the transport protocol that the service recognizes. The values for this parameter can be IPPROTO_UDP or IPPROTO_TCP.
port	Specifies the port on the machine's **portmap** daemon.

Return Values

Upon successful completion, this routine returns a value of 1. If unsuccessful, it returns a value of 0.

Implementation Specifics

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

Related Information

The **portmap** daemon.

The **pmap_getmaps** subroutine, **pmap_unset** subroutine, **svc_register** subroutine.

Understanding Protocols for TCP/IP in *Communication Concepts and Procedures.*

pmap_unset Subroutine

Purpose

Destroys the mappings between a remote procedure call and the port.

Library

C Library (**libc.a**)

Syntax

#include <rpc/rpc.h>

pmap_unset (*prognum*, *versnum*)
u_long *prognum*, *versnum*;

Description

The **pmap_unset** subroutine destroys mappings between the program number and version number of a remote procedure call and the ports on the host **portmap** daemon.

Parameters

prognum	Specifies the program number of the remote program.
versnum	Specifies the version number of the remote program.

Implementation Specifics

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

Related Information

The **portmap** daemon.

The **pmap_getmaps** subroutine, **pmap_set** subroutine, **svc_unregister** subroutine.

registerrpc Subroutine

Purpose

Registers a procedure with the RPC service package.

Library

C Library (**libc.a**)

Syntax

#include <rpc/rpc.h>
registerrpc (*prognum*, *versnum*, *procnum*, *procname*, *inproc*, *outproc*)
u_long *prognum*, *versnum*, *procnum*;
char * (**procname*) ();
xdrproc_t *inproc*, *outproc*;

Description

The **registerrpc** subroutine registers a procedure with the Remote Procedure Call (RPC) service package.

If a request arrives that matches the values of the *prognum* parameter, the *versnum* parameter, and the *procnum* parameter, then the *procname* parameter is called with a pointer to its parameters, after which it returns a pointer to its static results.

Note: Remote procedures registered in this form are accessed using the User Datagram Protocol/Internet Protocol (UDP/IP) transport protocol only.

Parameters

prognum	Specifies the program number of the remote program.
versnum	Specifies the version number of the remote program.
procnum	Identifies the procedure number to be called.
procname	Identifies the procedure name.
inproc	Specifies the eXternal Data Representation (XDR) subroutine that decodes the procedure parameters.
outproc	Specifies the XDR subroutine that encodes the procedure results.

Return Values

Upon successful completion, this subroutine returns a value of 1. If unsuccessful, it returns a value of −1.

Implementation Specifics

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

Related Information

The **callrpc** subroutine, **svcudp_create** subroutine.

eXternal Data Representation (XDR) Overview for Programming in *Communications Programming Concepts*.

User Datagram Protocol (UDP) in *Communication Concepts and Procedures*.

rtime Subroutine

Purpose

Gets remote time.

Library

C Library (**libc.a**)

Syntax

#include <rpc/rpc.h>

#include <sys/types.h>
#include <sys/time.h>
#include <netinet/in.h>
int rtime (*addrp*, *timep*, *timeout*)
struct sockaddr_in **addrp*;
struct timeval **timep*;
struct timeval **timeout*;

Description

The **rtime** subroutine consults the Internet Time Server (TIME) at the address pointed to by the *addrp* parameter and returns the remote time in the **timeval** structure pointed to by the *timep* parameter. Normally, the User Datagram Protocol (UDP) protocol is used when consulting the time server. If the *timeout* parameter is specified as NULL, however, the routine instead uses Transmission Control Protocol (TCP) and blocks until a reply is received from the time server.

Parameters

addrp	Points to the Internet Time Server.
timep	Points to the **timeval** structure.
timeout	Specifies how long the routine waits for a reply before terminating.

Return Values

Upon successful completion, this subroutine returns a value of 0. If unsuccessful, it returns a value of –1, and the error number parameter (*errno*) is set to reflect the cause of the error.

Implementation Specifics

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

Related Information

Understanding Protocols for TCP/IP, User Datagram Protocol (UDP) in *Communication Concepts and Procedures.*

store Subroutine

Purpose

Places data under a key.

Library

DBM Library **(libdbm.a)**

Syntax

#include <dbm.h>
store (*key, content***)**
datum *key, content*;

Description

The **store** subroutine places data under a key.

Parameters

key	Specifies the input key.
content	Specifies the value associated with the key to store.

Return Values

Upon successful completion, this subroutine returns a value of 0 (zero). If unsuccessful, it returns a negative value.

Implementation Specifics

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

Related Information

The **dbm_store** subroutine.

svc_destroy Macro

Purpose

Destroys an RPC service transport handle.

Library

C Library (**libc.a**)

Syntax

#include <rpc/rpc.h>

void
svc_destroy (*xprt***)**
SVCXPRT **xprt*;

Description

The **svc_destroy** macro destroys a Remote Procedure Call (RPC) service transport handle. Destroying the service transport handle deallocates the private data structures, including the handle itself. After the **svc_destroy** macro is used, the handle pointed to by the **xprt** parameter is no longer defined.

Parameter

xprt	Points to the RPC service transport handle.

Implementation Specifics

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

Related Information

The **clnt_destroy** macro, **svc_freeargs** macro.

svc_freeargs Macro

Purpose

Frees data allocated by the RPC/XDR system.

Library

C Library (**libc.a**)

Syntax

#include <rpc/rpc.h>

svc_freeargs (*xprt*, *inproc*, *in*)
SVCXPRT **xprt*;
xdrproc_t *inproc*;
char **in*;

Description

The **svc_freeargs** macro frees data allocated by the Remote Procedure Call/eXternal Data Representation (RPC/XDR) system. This data is allocated when the RPC/XDR system decodes the arguments to a service procedure with the **svc_getargs** macro.

Parameters

xprt — Points to the RPC service transport handle.

inproc — Specifies the XDR routine that decodes the arguments.

in — Specifies the address where the procedure arguments are placed.

Implementation Specifics

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

Related Information

The **svc_getargs** macro, **svc_destroy** macro.

eXternal Data Representation (XDR) Overview for Programming in *Communications Programming Concepts*.

svc_getargs Macro

Purpose

Decodes the arguments of an RPC request.

Library

C Library (**libc.a**)

Syntax

#include <rpc/rpc.h>

svc_getargs (*xprt*, *inproc*, *in*)
SVCXPRT **xprt*;
xdrproc_t *inproc*;
char **in*;

Description

The **svc_getargs** macro decodes the arguments of a Remote Procedure Call (RPC) request associated with the RPC service transport handle.

Parameters

xprt	Points to the RPC service transport handle.
inproc	Specifies the eXternal Data Representation (XDR) routine that decodes the arguments.
in	Specifies the address where the arguments are placed.

Return Values

Upon successful completion, this subroutine returns a value of 1. If unsuccessful, it returns a value of 0.

Implementation Specifics

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

Related Information

The **svc_freeargs** macro.

eXternal Data Representation (XDR) Overview for Programming in *Communications Programming Concepts*.

svc_getcaller Macro

Purpose

Gets the network address of the caller of a procedure.

Library

C Library (**libc.a**)

Syntax

#include <rpc/rpc.h>

struct sockaddr_in *
svc_getcaller (*xprt*)
**SVCXPRT ****xprt*;

Description

The **svc_getcaller** macro retrieves the network address of the caller of a procedure associated with the Remote Procedure Call (RPC) service transport handle.

Parameters

xprt	Points to the RPC service transport handle.

Implementation Specifics

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

Related Information

The **svc_register** subroutine, **svc_run** subroutine.

svc_getreqset Subroutine

Purpose

Services an RPC request.

Library

C Library (**libc.a**)

Syntax

#include <rpc/rpc.h>

void
svc__getreqset (*rdfds*)
fd_set **rdfds*;

Description

The **svc__getreqset** subroutine is only used if a service implementor does not call the **svc_run** subroutine, but instead implements custom asynchronous event processing. The subroutine is called when the **select** subroutine has determined that a Remote Procedure Call (RPC) request has arrived on any RPC sockets. The **svc_getreqset** subroutine returns when all sockets associated with the value specified by the *rdfds* parameter have been serviced.

Parameters

rdfds	Specifies the resultant read-file descriptor bit mask.

Implementation Specifics

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

Related Information

The **select** subroutine, **svc_run** subroutine.

Sockets Overview in *Communications Programming Concepts.*

svc_register Subroutine

Purpose

Maps a remote procedure.

Library

C Library (**libc.a**)

Syntax

#include <rpc/rpc.h>

svc_register (*xprt*, *prognum*, *versnum*, *dispatch*, *protocol*)
SVCXPRT **xprt*;
u_long *prognum*, *versnum*;
void (**dispatch*) ();
int *protocol*;

Description

The **svc_register** subroutine maps a remote procedure with a service dispatch procedure pointed to by the *dispatch* parameter. If the *protocol* parameter has a value of 0, the service is not registered with the **portmap** daemon. If the *protocol* parameter does not have a value of 0 (or if it is IPPROTO_UDP or IPPROTO_TCP), the remote procedure triple *(prognum*, *versnum*, and *protocol* parameters) is mapped to the **xprt–>xp_port** port.

The dispatch procedure takes the following form:

dispatch (*request, xprt*)
struct svc_req **request*;
SVCXPRT **xprt*;

Parameters

xprt	Points to a Remote Procedure Call (RPC) service transport handle.
prognum	Specifies the program number of the remote program.
versnum	Specifies the version number of the remote program.
dispatch	Points to the service dispatch procedure.
protocol	Specifies the data transport used by the service.

Return Values

Upon successful completion, this subroutine returns a value of 1. If unsuccessful, it returns a value of 0.

Implementation Specifics

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

Related Information

The **portmap** daemon.

The **pmap_set** subroutine, **pmap_getmaps** subroutine, **svc_unregister** subroutine.

Understanding Protocols for TCP/IP, User Datagram Protocol (UDP) in *Communication Concepts and Procedures.*

svc_run Subroutine

Purpose

Waits for a Remote Procedure Call (RPC) service request to arrive.

Library

C Library (**libc.a**)

Syntax

#include <rpc/rpc.h>

void
svc_run (*xprt*);
SCVXPRT **xprt*;

Description

The **svc_run** subroutine waits for an RPC service request to arrive. When a request arrives, the **svc_run** subroutine calls the appropriate service procedure with the **svc_getreqset** subroutine. This procedure is usually waiting for a **select** subroutine to return.

Parameters

xprt	Points to an RPC service transport handle.

Implementation Specifics

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

Related Information

The **callrpc** subroutine, **registerrpc** subroutine, **select** subroutine, **svc_getreqset** subroutine.

Using the Intermediate Layer of RPC, Using the registerrpc Routine in *Communications Programming Concepts.*

svc_sendreply Subroutine

Purpose

Sends back the results of a remote procedure call.

Library

C Library (**libc.a**)

Syntax

#include <rpc/rpc.h>

svc_sendreply (*xprt*, *outproc*, *out*)
SVCXPRT **xprt*;
xdrproc_t *outproc*;
char **out*;

Description

The **svc_sendreply** subroutine sends back the results of a remote procedure call. This subroutine is called by a Remote Procedure Call (RPC) service dispatch subroutine.

Parameters

xprt	Points to the RPC service transport handle of the caller.
outproc	Specifies the eXternal Data Representation (XDR) routine that encodes the results.
out	Points to the address where results are placed.

Return Values

Upon successful completion, this subroutine returns a value of 1. If unsuccessful, it returns a value of 0.

Implementation Specifics

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

Related Information

eXternal Data Representation (XDR) Overview for Programming in *Communications Programming Concepts*.

svc_unregister Subroutine

Purpose

Removes mappings between procedures and objects.

Library

C Library (**libc.a**)

Syntax

#include <rpc/rpc.h>

void
svc_unregister (*prognum*, *versnum*)
u_long *prognum*, *versnum*

Description

The **svc_unregister** subroutine removes mappings between dispatch subroutines and the service procedure identified by the *prognum* parameter and the *versnum* parameter. It also removes the mapping between the port number and the service procedure which is identified by the *prognum* parameter and the *versnum* parameter.

Parameters

prognum	Specifies the program number of the remote program.
versnum	Specifies the version number of the remote program.

Implementation Specifics

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

Related Information

The **pmap_unset** subroutine, **svc_register** subroutine.

svcerr_auth Subroutine

Purpose

Indicates that the service dispatch routine cannot complete a remote procedure call due to an authentication error.

Library

RPC Library (**libcrpc.a**)

Syntax

#include <rpc/rpc.h>

void
svcerr_auth (*xprt*, *why***)**
SVCXPRT **xprt*;
enum auth_stat *why*;

Description

The **svcerr_auth** subroutine is called by a service dispatch subroutine that refuses to perform a remote procedure call because of an authentication error. This subroutine sets the status of the RPC reply message to AUTH_ERROR.

Parameters

xprt	Points to the Remote Procedure Call (RPC) service transport handle.
why	Specifies the authentication error.

Implementation Specifics

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

svcerr_decode Subroutine

Purpose

Indicates that the service dispatch routine cannot decode the parameters of a request.

Library

C Library (**libc.a**)

Syntax

#include <rpc/rpc.h>

void
svcerr_decode (*xprt***)**
SVCXPRT **xprt*;

Description

The **svcerr_decode** subroutine is called by a service dispatch subroutine that cannot decode the parameters specified in a request. This subroutine sets the status of the RPC reply message to the GARBAGE_ARGS condition.

Parameter

xprt	Points to the Remote Procedure Call (RPC) service transport handle.

Implementation Specifics

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

Related Information

The **svc_getargs** macro.

svcerr_noproc Subroutine

Purpose

Indicates that the service dispatch routine cannot complete a remote procedure call because the program cannot support the requested procedure.

Library

C Library (**libc.a**)

Syntax

#include <rpc/rpc.h>

void
svcerr_noproc (*xprt*)
SVCXPRT **xprt*;

Description

The **svcerr_noproc** subroutine is called by a service dispatch routine that does not implement the procedure number the caller has requested. This subroutine sets the status of the RPC reply message to the PROC_UNAVAIL condition, which indicates that the program cannot support the requested procedure.

Note: Service implementors do not usually need this subroutine.

Parameter

xprt Points to the Remote Procedure Call (RPC) service transport handle.

Implementation Specifics

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

svcerr_noprog Subroutine

Purpose

Indicates that the service dispatch routine cannot complete a remote procedure call because the requested program is not registered.

Library

C Library (**libc.a**)

Syntax

#include <rpc/rpc.h>

void
svcerr_noprog (*xprt*)
SVCXPRT **xprt*;

Description

The **svcerr_noprog** subroutine is called by a service dispatch routine when the requested program is not registered with the Remote Procedure Call (RPC) package. This subroutine sets the status of the RPC reply message to the PROG_UNAVAIL condition, which indicates that the remote server has not exported the program.

Note: Service implementors do not usually need this subroutine.

Parameter

xprt Points to the RPC service transport handle.

Implementation Specifics

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

svcerr_progvers Subroutine

Purpose

Indicates that the service dispatch routine cannot complete the remote procedure call because the requested program version is not registered.

Library

C Library (**libc.a**)

Syntax

#include <rpc/rpc.h>

void
svcerr_progvers (*xprt***)**
SVCXPRT **xprt*;
u_long

Description

The **svcerr_progvers** subroutine is called by a service dispatch routine when the requested version of a program is not registered with the Remote Procedure Call (RPC) package. This subroutine sets the status of the RPC reply message to the PROG_ MISMATCH condition, which indicates that the remote server cannot support the client's version number.

Note: Service implementors do not usually need this subroutine.

Parameter

xprt	Points to the RPC service transport handle.

Implementation Specifics

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

svcerr_systemerr Subroutine

Purpose

Indicates that the service dispatch routine cannot complete the remote procedure call due to an error that is not covered by a protocol.

Library

C Library (**libc.a**)

Syntax

#include <rpc/rpc.h>

void
svcerr_systemerr (*xprt*)
SVCXPRT **xprt*;

Description

The **svcerr_systemerr** subroutine is called by a service dispatch subroutine that detects a system error not covered by a protocol. For example, a service dispatch subroutine calls the **svcerr_systemerr** subroutine if the first subroutine can no longer allocate storage. The routine sets the status of the RPC reply message to the SYSTEM_ERR condition.

Parameter

xprt	Points to the Remote Procedure Call (RPC) service transport handle.

Implementation Specifics

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

svcerr_weakauth Subroutine

Purpose

Indicates that the service dispatch routine cannot complete the remote procedure call due to insufficient authentication security parameters.

Library

C Library (**libc.a**)

Syntax

#include <rpc/rpc.h>

void
svcerr_weakauth (*xprt*)
SVCXPRT **xprt*;

Description

The **svcerr_weakauth** subroutine is called by a service dispatch routine that cannot make the remote procedure call because the supplied authentication parameters are insufficient for security reasons.

The **svcerr_weakauth** subroutine calls the **svcerr_auth** subroutine with the correct Remote Procedure Call (RPC) service transport handle (the *xprt* parameter). The subroutine also sets the status of the RPC reply message to the AUTH_TOOWEAK condition as the authentication error (AUTH_ERR).

Parameter

xprt	Points to the RPC service transport handle.

Implementation Specifics

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

Related Information

The **svcerr_auth** subroutine, **svcerr_decode** subroutine.

svcfd_create Subroutine

Purpose

Creates a service on any open file descriptor.

Library

C Library (**libc.a**)

Syntax

#include <rpc/rpc.h>

SVCXPRT *
svcfd_create (*fd*, *sendsize*, *recvsize***)**
int *fd*;
u_int *sendsize*;
u_int *recvsize*;

Description

The **svcfd_create** subroutine creates a service on any open file descriptor. Typically, this descriptor is a connected socket for a stream protocol such as Transmission Control Protocol (TCP).

Parameters

fd	Identifies the descriptor.
sendsize	Specifies the size of the send buffer.
recvsize	Specifies the size of the receive buffer.

Return Values

Upon successful completion, this subroutine returns a TCP-based transport handle. If unsuccessful, it returns a value of NULL.

Implementation Specifics

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

Related Information

Sockets Overview in *Communications Programming Concepts.*

Understanding Protocols for TCP/IP in *Communication Concepts and Procedures.*

svcraw_create Subroutine

Purpose

Creates a toy RPC service transport handle for simulation.

Library

C Library (**libc.a**)

Syntax

#include <rpc/rpc.h>

SVCXPRT *
svcraw_create ( )

Description

The **svcraw_create** subroutine creates a toy Remote Procedure Call (RPC) service transport handle. The service transport handle is located within the address space of the process. If the corresponding RPC server resides in the same address space, then simulation of RPC and acquisition of RPC overheads, such as round-trip times, are done without kernel interference.

Parameters

This subroutine contains no parameters.

Return Values

Upon successful completion, this subroutine returns a pointer to a valid RPC transport handle. If unsuccessful, it returns a value of NULL.

Implementation Specifics

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

Related Information

The **clntraw_create** subroutine.

svctcp_create Subroutine

Purpose

Creates a TCP/IP service transport handle.

Library

C Library (**libc.a**)

Syntax

#include <rpc/rpc.h>

SVCXPRT *
svctcp_create (*sock*, *sendsz*, *recvsz*)
int *sock*;
u_int *sendsz*, *rcvcsz*;

Description

The **svctcp_create** subroutine creates a Remote Procedure Call (RPC) service transport handle based on Transmission Control Protocol/Internet Protocol (TCP/IP) and returns a pointer to it.

Since TCP/IP remote procedure calls use buffered I/O, users can set the size of the send and receive buffers with the *sendsz* and *recvsz* parameters, respectively. If the size of either buffer is set to a value of 0, the **svctcp_create** subroutine picks suitable default values.

Parameters

sock — Specifies the socket associated with the transport. If the value of the *sock* parameter is RPC_ANYSOCK, the **svctcp_create** subroutine creates a new socket. The service transport handle socket number is set to **xprt–>xp_sock**. If the socket is not bound to a local TCP/IP port, then this routine binds the socket to an arbitrary port. Its port number is set to **xprt–>xp_port**.

sendsz — Specifies the size of the send buffer.

recvsz — Specifies the size of the receive buffer.

Return Values

Upon successful completion, this subroutine returns a valid RPC service transport handle. If unsuccessful, it returns a value of NULL.

Implementation Specifics

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

Related Information

The **registerrpc** subroutine, **svcudp_create** subroutine.

Sockets Overview in *Communications Programming Concepts*.

Understanding Protocols for TCP/IP in *Communication Concepts and Procedures*.

svcudp_create Subroutine

Purpose

Creates a UDP/IP service transport handle.

Library

C Library (**libc.a**)

Syntax

#include <rpc/rpc.h>

SVCXPRT *
svcudp_create (*sock***)**
int *sock***;**

Description

The **svcudp_create** subroutine creates a Remote Procedure Call (RPC) service transport handle based on User Datagram Protocol/Internet Protocol (UDP/IP) and returns a pointer to it.

The UDP/IP service transport handle is used only for procedures that take up to 8K bytes of encoded arguments or results.

Parameter

sock Specifies the socket associated with the service transport handle. If the value specified by the *sock* parameter is RPC_ANYSOCK, the **svcudp_create** subroutine creates a new socket and sets the service transport handle socket number to **xprt–>xp_sock**. If the socket is not bound to a local UDP/IP port, then the **svcudp_create** subroutine binds the socket to an arbitrary port. The port number is set to **xprt–>xp_port.**

Return Values

Upon successful completion, this subroutine returns a valid RPC service transport. If unsuccessful, it returns a value of NULL.

Implementation Specifics

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

Related Information

The **registerrpc** subroutine, **svctcp_create** subroutine.

Understanding Protocols for TCP/IP, User Datagram Protocol (UDP) in *Communication Concepts and Procedures.*

user2netname Subroutine

Purpose

Converts from a domain-specific user ID to an operating-system-independent network name.

Library

C Library (**libc.a**)

Syntax

#include <rpc/rpc.h>

user2netname (*name*, *uid*, *domain*)
char **name*;
int *uid*;
char **domain*;

Description

The **user2netname** subroutine converts from a domain-specific user ID to an operating system-independent-network name.

This subroutine is the inverse of the **netname2user** subroutine.

Parameters

name	Points to the network name (or netname) of the server process owner.
uid	Points to the caller's effective user ID (UID).
domain	Points to the domain name.

Return Values

Upon successful completion, this subroutine returns a value of 1. If unsuccessful, it returns a value of 0.

Implementation Specifics

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

Related Information

The **host2netname** subroutine, **netname2user** subroutine.

xdr_accepted_reply Subroutine

Purpose

Encodes RPC reply messages.

Library

C Library (**libc.a**)

Syntax

#include <rpc/rpc.h>

xdr_accepted_reply (*xdrs*, *ar*)
XDR **xdrs*;
struct accepted_reply **ar*;

Description

The **xdr_accepted_reply** subroutine encodes Remote Procedure Call (RPC) reply messages. The routine generates message replies similar to RPC message replies without using the RPC program.

Parameters

xdrs	Points to the eXternal Data Representation (XDR) stream handle.
ar	Specifies the address of the structure that contains the RPC reply.

Return Values

Upon successful completion, this subroutine returns a value of 1. If unsuccessful, it returns a value of 0.

Implementation Specifics

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

Related Information

eXternal Data Representation (XDR) Overview for Programming in *Communications Programming Concepts.*

xdr_array Subroutine

Purpose

Translates between variable-length arrays and their corresponding external representations.

Library

C Library (**libc.a**)

Syntax

#include <rpc/xdr.h>

xdr_array (*xdrs*, *arrp*, *sizep*, *maxsize*, *elsize*, *elproc*)
XDR **xdrs*;
char ***arrp*;
u_int **sizep*;
u_int *maxsize*;
u_int *elsize*;
xdrproc_t *elproc*;

Description

The **xdr_array** subroutine is a filter primitive that translates between variable-length arrays and their corresponding external representations. This subroutine is called to encode or decode each element of the array.

Parameters

xdrs	Points to the eXternal Data Representation (XDR) stream handle.
arrp	Specifies the address of the pointer to the array. If the *arrp* parameter is NULL when the array is being deserialized, XDR allocates an array of the appropriate size and sets the parameter to that array.
sizep	Specifies the address of the element count of the array. The element count cannot exceed the value for the *maxsize* parameter.
maxsize	Specifies the maximum number of array elements.
elsize	Specifies the byte size of each of the array elements.
elproc	Translates between the C form of the array elements and their external representations. This parameter is an XDR filter.

Return Values

Upon successful completion, this routine returns a value of 1. If unsuccessful, it returns a value of 0.

Implementation Specifics

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

Related Information

Understanding XDR Library Filter Primitives in *Communications Programming Concepts*.

xdr_authunix_parms Subroutine

Purpose

Describes UNIX-style credentials.

Library

C Library (**libc.a**)

Syntax

#include <rpc/rpc.h>

xdr_authunix_parms (*xdrs*, *app*)
XDR **xdrs*;
struct authunix_parms **app*;

Description

The **xdr_authunix_parms** subroutine describes UNIX-style credentials. This subroutine generates credentials without using the Remote Procedure Call (RPC) authentication program.

Parameters

xdrs	Points to the eXternal Data Representation (XDR) stream handle.
app	Points to the structure that contains the UNIX-style authentication credentials.

Return Values

Upon successful completion, this subroutine returns a value of 1. If unsuccessful, it returns a value of 0.

Implementation Specifics

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

Related Information

eXternal Data Representation (XDR) Overview for Programming in *Communications Programming Concepts*.

xdr_bytes Subroutine

Purpose

Translates between internal counted byte arrays and their external representations.

Library

C Library (**libc.a**)

Syntax

#include <rpc/xdr.h>

xdr_bytes (*xdrs*, *sp*, *sizep*, *maxsize*)
XDR **xdrs*;
char ***sp*;
u_int **sizep*;
u_int *maxsize*;

Description

The **xdr_bytes** subroutine is a filter primitive that translates between counted byte arrays and their external representations. This subroutine treats a subset of generic arrays, in which the size of array elements is known to be 1 (one), and the external description of each element is built-in. The length of the byte array is explicitly located in an unsigned integer. The byte sequence is not terminated by a null character. The external representation of the bytes is the same as their internal representation.

Parameters

xdrs	Points to the eXternal Data Representation (XDR) stream handle.
sp	Specifies the address of the pointer to the byte array.
sizep	Points to the length of the byte area. The value of this parameter cannot exceed the value of the *maxsize* parameter.
maxsize	Specifies the maximum number of bytes allowed when XDR encodes or decodes messages.

Return Values

Upon successful completion, this routine returns a value of 1. If unsuccessful, it returns a value of 0.

Implementation Specifics

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

Related Information

Understanding XDR Library Filter Primitives in *Communications Programming Concepts*.

xdr_callhdr Subroutine

Purpose

Describes RPC call header messages.

Library

C Library (**libc.a**)

Syntax

#include <rpc/rpc.h>

xdr_callhdr (*xdrs*, *chdr*)
XDR **xdrs*;
struct rpc_msg **chdr*;

Description

The **xdr_callhdr** subroutine describes Remote Procedure Call (RPC) call-header messages. This subroutine generates call headers that are similar to RPC call headers without using the RPC program.

Parameters

xdrs	Points to the eXternal Data Representation (XDR) stream handle.
chdr	Points to the structure that contains the header for the call message.

Return Values

Upon successful completion, this subroutine returns a value of 1. If unsuccessful, it returns a value of 0.

Implementation Specifics

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

Related Information

eXternal Data Representation (XDR) Overview for Programming in *Communications Programming Concepts.*

xdr_callmsg Subroutine

Purpose

Describes RPC call messages.

Library

C Library (**libc.a**)

Syntax

#include <rpc/rpc.h>

xdr_callmsg (*xdrs*, *cmsg*)
XDR **xdrs*;
struct rpc_msg **cmsg*;

Description

The **xdr_callmsg** subroutine describes Remote Procedure Call (RPC) call messages. This subroutine generates messages similar to RPC messages without using the RPC program.

Parameters

xdrs	Points to the eXternal Data Representatiion (XDR) stream handle.
cmsg	Points to the structure that contains the text of the call message.

Return Values

Upon successful completion, this subroutine returns a value of 1. If unsuccessful, it returns a value of 0.

Implementation Specifics

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

Related Information

eXternal Data Representation (XDR) Overview for Programming in *Communications Programming Concepts*.

xdr_char Subroutine

Purpose

Translates between C language characters and their external representations.

Library

C Library (**libc.a**)

Syntax

#include <rpc/xdr.h>

xdr_char (*xdrs*, *cp*)
XDR **xdrs*;
char **cp*;

Description

The **xdr_char** subroutine is a filter primitive that translates between C characters and their external representations.

Note: Encoded characters are not packed and occupy 4 bytes each. For arrays of characters, the programmer should consider using the **xdr_bytes**, **xdr_opaque**, or **xdr_string** routine.

Parameters

xdrs	Points to the eXternal Data Representation (XDR) stream handle.
cp	Points to the character.

Return Values

Upon successful completion, this routine returns a value of 1. If unsuccessful, it returns a value of 0.

Implementation Specifics

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

Related Information

Understanding XDR Library Filter Primitives in *Communications Programming Concepts*.

xdr_destroy Macro

Purpose

Destroys the XDR stream pointed to by the *xdrs* parameter.

Library

C Library (**libc.a**)

Syntax

#include <rpc/xdr.h>

void
xdr_destroy (*xdrs*)
XDR **xdrs*;

Description

The **xdr_destroy** macro invokes the destroy routine associated with the eXternal Data Representation (XDR) stream pointed to by the *xdrs* parameter and frees the private data structures allocated to the stream. The use of the XDR stream handle is undefined after it is destroyed.

Parameter

xdrs	Points to the XDR stream handle.

Implementation Specifics

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

Related Information

Understanding XDR Non-Filter Primitives in *Communications Programming Concepts*.

xdr_double Subroutine

Purpose

Translates between C language double-precision numbers and their external representations.

Library

C Library (**libc.a**)

Syntax

#include <rpc/xdr.h>

xdr_double (*xdrs*, *dp*)
XDR **xdrs*;
double **dp*;

Description

The **xdr_double** subroutine is a filter primitive that translates between C double-precision numbers and their external representations.

Parameters

xdrs	Points to the eXternal Data Representation (XDR) stream handle.
dp	Specifies the address of the double-precision number.

Return Values

Upon successful completion, this subroutine returns a value of 1. If unsuccessful, it returns a value of 0.

Implementation Specifics

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

Related Information

Understanding XDR Library Filter Primitives in *Communications Programming Concepts*.

xdr_enum Subroutine

Purpose

Translates between a C language enumeration (enum) and its external representation.

Library

C Library (**libc.a**)

Syntax

#include <rpc/xdr.h>

xdr_enum (*xdrs*, *ep*)
XDR **xdrs*;
enum_t **ep*;

Description

The **xdr_enum** subroutine is a filter primitive that translates between a C language enumeration (enum) and its external representation.

Parameters

xdrs	Points to the eXternal Data Representation (XDR) stream handle.
ep	Specifies the address of the enumeration data.

Return Values

Upon successful completion, this routine returns a value of 1. If unsuccessful, it returns a value of 0.

Implementation Specifics

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

Related Information

Understanding XDR Library Filter Primitives in *Communications Programming Concepts*.

xdr_float Subroutine

Purpose

Translates between C language floats and their external representations.

Library

C Library (**libc.a**)

Syntax

#include <rpc/xdr.h>

xdr_float (*xdrs*, *fp*)
XDR **xdrs*;
float **fp*;

Description

The **xdr_float** subroutine is a filter primitive that translates between C floats (normalized single floating-point numbers) and their external representations.

Parameters

xdrs	Points to the eXternal Data Representation (XDR) stream handle.
fp	Specifies the address of the float.

Return Values

Upon successful completion, this subroutine returns a value of 1. If unsuccessful, it returns a value of 0.

Implementation Specifics

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

Related Information

Understanding XDR Library Filter Primitives in *Communications Programming Concepts*.

xdr_free Subroutine

Purpose

Deallocates, or frees, memory.

Library

C Library (**libc.a**)

Syntax

#include <rpc/xdr.h>

void
xdr_free (*proc*, *objp*)
xdrproc_t *proc*;
char **objp*;

Description

The **xdr_free** subroutine is a generic freeing routine that deallocates memory. The first argument is the eXternal Data Representation (XDR) routine for the object being freed. The second argument is a pointer to the object itself.

Note: The pointer passed to this routine is *not* freed, but what it points to *is* freed (recursively).

Parameters

proc Points to the XDR stream handle.

objp Points to the object being freed.

Implementation Specifics

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

Related Information

Understanding XDR Non-Filter Primitives in *Communications Programming Concepts*.

xdr_getpos Macro

Purpose

Returns an unsigned integer that describes the current position in the data stream.

Library

C Library (**libc.a**)

Syntax

#include <rpc/xdr.h>

u_int
xdr_getpos (*xdrs*)
XDR **xdrs*;

Description

The **xdr_getpos** macro invokes the get-position routine associated with the eXternal Data Representation (XDR) stream pointed to by the *xdrs* parameter. This routine returns an unsigned integer that describes the current position in the data stream.

Parameter

xdrs	Points to the XDR stream handle.

Return Values

This macro returns an unsigned integer describing the current position in the stream. In some XDR streams, this routine returns a value of –1, even though the value has no meaning.

Implementation Specifics

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

Related Information

Understanding XDR Non-Filter Primitives in *Communications Programming Concepts*.

xdr_inline Macro

Purpose

Returns a pointer to the buffer of a stream pointed to by the *xdrs* parameter.

Library

C Library (**libc.a**)

Syntax

#include <rpc/xdr.h>

long *
x_inline (*xdrs*, *len*)
XDR **xdrs*;
int *len*;

Description

The **xdr_inline** macro invokes the inline routine associated with the eXternal Data Representation (XDR) stream pointed to by the *xdrs* parameter. The routine returns a pointer to a contiguous piece of the stream's buffer, whose size is specified by the *len* parameter. The buffer can be used for any purpose, but it is not data-portable. This routine may return a value of NULL if it cannot return a buffer segment of the requested size.

Parameters

xdrs	Points to the XDR stream handle.
len	Specifies the size, in bytes, of the internal buffer.

Return Values

This macro returns a pointer to a piece of the stream's buffer.

Implementation Specifics

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

Related Information

Understanding XDR Non-Filter Primitives in *Communications Programming Concepts*.

xdr_int Subroutine

Purpose

Translates between C language integers and their external representations.

Library

C Library (**libc.a**)

Syntax

#include <rpc/xdr.h>

xdr_int (*xdrs*, *ip*)
XDR ******xdrs*;
int ******ip*;

Description

The **xdr_int** subroutine is a filter primitive that translates between C language integers and their external representations.

Parameters

xdrs	Points to the eXternal Data Representation (XDR) stream handle.
ip	Specifies the address of the integer.

Return Values

Upon successful completion, this routine returns a value of 1. If unsuccessful, it returns a value of 0.

Implementation Specifics

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

Related Information

Understanding XDR Library Filter Primitives in *Communications Programming Concepts*.

xdr_long Subroutine

Purpose

Translates between C language long integers and their external representations.

Library

C Library (**libc.a**)

Syntax

#include <rpc/xdr.h>

xdr_long (*xdrs*, *lp*)
XDR **xdrs*;
long **lp*;

Description

The **xdr_long** filter primitive translates between C language long integers and their external representations. This primitive is characteristic of most eXternal Data Representation (XDR) library primitives and all client XDR routines.

Parameters

xdrs — Points to the XDR stream handle. This parameter can be treated as an opaque handler and passed to the primitive routines.

lp — Specifies the address of the number.

Return Values

Upon successful completion, this routine returns a value of 1. If unsuccessful, it returns a value of 0.

Implementation Specifics

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

Related Information

Understanding XDR Library Filter Primitives in *Communications Programming Concepts*.

xdr_opaque Subroutine

Purpose

Translates between fixed-size opaque data and its external representation.

Library

C Library (**libc.a**)

Syntax

#include <rpc/xdr.h>

xdr_opaque (*xdrs*, *cp*, *cnt*)
XDR **xdrs*;
char **cp*;
u_int *cnt*;

Description

The **xdr_opaque** subroutine is a filter primitive that translates between fixed-size opaque data and its external representation.

Parameters

xdrs	Points to the eXternal Data Representation (XDR) stream handle.
cp	Specifies the address of the opaque object.
cnt	Specifies the size, in bytes, of the object. By definition, the actual data contained in the opaque object is not machine-portable.

Return Values

Upon successful completion, this subroutine returns a value of 1. If unsuccessful, it returns a value of 0.

Implementation Specifics

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

Related Information

Understanding XDR Library Filter Primitives in *Communications Programming Concepts*.

xdr_opaque_auth Subroutine

Purpose

Describes RPC authentication messages.

Library

C Library (**libc.a**)

Syntax

#include <rpc/rpc.h>

xdr_opaque_auth (*xdrs*, *ap*)
XDR **xdrs*;
struct opaque_auth **ap*;

Description

The **xdr_opaque_auth** subroutine describes Remote Procedure Call (RPC) authentication information messages. It generates RPC authentication message data without using the RPC program.

Parameters

xdrs	Points to the eXternal Data Representation (XDR) stream handle.
ap	Points to the structure that contains the authentication information.

Return Values

Upon successful completion, this subroutine returns a value of 1. If unsuccessful, it returns a value of 0.

Implementation Specifics

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

Related Information

eXternal Data Representation (XDR) Overview for Programming in *Communications Programming Concepts.*

xdr_pmap Subroutine

Purpose

Describes parameters for **portmap** procedures.

Library

C Library (**libc.a**)

Syntax

#include <rpc/rpc.h>

xdr_pmap (*xdrs*, *regs*)
XDR **xdrs*;
struct pmap **regs*;

Description

The **xdr_pmap** subroutine describes parameters for **portmap** procedures. This subroutine generates **portmap** parameters without using the **portmap** interface.

Parameters

xdrs Points to the eXternal Data Representation (XDR) stream handle.

regs Points to the buffer or register where the **portmap** daemon stores information.

Return Values

Upon successful completion, this subroutine returns a value of 1. If unsuccessful, it returns a value of 0.

Implementation Specifics

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

Related Information

The **portmap** daemon.

eXternal Data Representation (XDR) Overview for Programming in *Communications Programming Concepts*.

xdr_pmaplist Subroutine

Purpose

Describes a list of port mappings externally.

Library

C Library (**libc.a**)

Syntax

#include <rpc/rpc.h>

xdr_pmaplist (*xdrs*, *rp*)
XDR **xdrs*;
struct pmaplist ***rp*;

Description

The **xdr_pmaplist** subroutine describes a list of port mappings externally. This subroutine generates the port mappings to Remote Procedure Call (RPC) ports without using the **portmap** interface.

Parameters

xdrs	Points to the eXternal Data Representation (XDR) stream handle.
rp	Points to the structure that contains the **portmap** listings.

Return Values

Upon successful completion, this subroutine returns a value of 1. If unsuccessful, it returns a value of 0.

Implementation Specifics

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

Related Information

The **portmap** daemon.

eXternal Data Representation (XDR) Overview for Programming in *Communications Programming Concepts*.

xdr_pointer Subroutine

Purpose

Provides pointer chasing within structures and serializes NULL pointers.

Library

C Library (**libc.a**)

Syntax

#include <rpc/xdr.h>

xdr_pointer (*xdrs*, *objpp*, *objsize*, *xdrobj*)
XDR **xdrs*;
char ***objpp*;
u_int *objsize*;
xdrproc_t *xdrobj*;

Description

The **xdr_pointer** subroutine provides pointer chasing within structures and serializes NULL pointers. This subroutine can represent recursive data structures, such as binary trees or linked lists.

Parameters

xdrs	Points to the eXternal Data Representation (XDR) stream handle.
objpp	Points to the character pointer of the data structure.
objsize	Specifies to the size of the structure.
xdrobj	Specifies the XDR filter for the object.

Return Values

Upon successful completion, this subroutine returns a value of 1. If unsuccessful, it returns a value of 0.

Implementation Specifics

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

Related Information

Understanding XDR Non-Filter Primitives in *Communications Programming Concepts.*

xdr_reference Subroutine

Purpose

Provides pointer chasing within structures.

Library

C Library (**libc.a**)

Syntax

#include <rpc/xdr.h>

xdr_reference (*xdrs*, *pp*, *size*, *proc*)
XDR **xdrs*;
char ***pp*;
u_int *size*;
xdrproc_t *proc*;

Description

The **xdr_reference** subroutine is a filter primitive that provides pointer chasing within structures. This primitive allows the serializing, deserializing, and freeing of pointers within one structure that are referenced by another structure.

The **xdr_reference** subroutine does not attach any special meaning to a null-value pointer during serialization. Attempting to pass the address of a NULL pointer can cause a memory error. The programmer must describe data with a two–armed discriminated union. One arm is used when the pointer is valid; the other arm is used when the pointer is NULL.

Parameters

xdrs	Points to the eXternal Data Representation (XDR) stream handle.
pp	Specifies the address of the pointer to the structure. When decoding data, XDR allocates storage if the pointer is NULL.
size	Specifies the byte size of the structure pointed to by the *pp* parameter.
proc	Filters the structure between its C form and its external representation. This parameter is the XDR procedure that describes the structure.

Return Values

Upon successful completion, this routine returns a value of 1. If unsuccessful, it returns a value of 0.

Implementation Specifics

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

Related Information

Understanding XDR Library Filter Primitives in *Communications Programming Concepts.*

xdr_rejected_reply Subroutine

Purpose

Describes RPC message rejection replies.

Library

C Library (**libc.a**)

Syntax

#include <rpc/rpc.h>

xdr_rejected_reply (*xdrs*, *rr*)
XDR **xdrs*;
struct rejected_reply **rr*;

Description

The **xdr_rejected_reply** subroutine describes Remote Procedure Call (RPC) message rejection replies. This subroutine can be used to generate rejection replies similar to RPC rejection replies without using the RPC program.

Parameters

xdrs	Points to the eXternal Data Representation (XDR) stream handle.
rr	Points to the structure that contains the rejected reply.

Return Values

Upon successful completion, this subroutine returns a value of 1. If unsuccessful, it returns a value of 0.

Implementation Specifics

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

Related Information

eXternal Data Representation (XDR) Overview for Programming in *Communications Programming Concepts.*

xdr_replymsg Subroutine

Purpose

Describes RPC message replies.

Library

C Library (**libc.a**)

Syntax

#include <rpc/rpc.h>

xdr_replymsg (*xdrs*, *rmsg*)
XDR **xdrs*;
struct rpc_msg **rmsg*;

Description

The **xdr_replymsg** subroutine describes Remote Procedure Call (RPC) message replies. Use this subroutine to generate message replies similar to RPC message replies without using the RPC program.

Parameters

xdrs	Points to the eXternal Data Representation (XDR) stream handle.
rmsg	Points to the structure containing the parameters of the reply message.

Return Values

Upon successful completion, this subroutine returns a value of 1. If unsuccessful, it returns a value of 0.

Implementation Specifics

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

Related Information

eXternal Data Representation (XDR) Overview for Programming in *Communications Programming Concepts.*

xdr_setpos Macro

Purpose

Changes the current position in the XDR stream.

Library

C Library (**libc.a**)

Syntax

#include <rpc/xdr.h>

xdr_setpos (*xdrs*, *pos*)
XDR **xdrs*;
u_int *pos*;

Description

The **xdr_setpos** macro invokes the set-position routine associated with the eXternal Data Representation (XDR) stream pointed to by the *xdrs* parameter. The new position setting is obtained from the **xdr_getpos** routine. This routine returns a value of FALSE if the set position is impossible or if the requested position is out of bounds.

A position cannot be set in some XDR streams. Trying to set a position in such streams causes the routine to fail. This routine also fails if the programmer requests a position that is not within the stream's boundaries.

Parameters

xdrs	Points to the XDR stream handle.
pos	Specifies a position value obtained from the **xdr_getpos** macro.

Return Values

Upon successful completion (if the stream is positioned successfully), this routine returns a value of 1. If unsuccessful, the routine returns a value of 0.

Implementation Specifics

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

Related Information

The **xdr_getpos** macro.

Understanding XDR Non-Filter Primitives in *Communications Programming Concepts*.

xdr_short Subroutine

Purpose

Translates between C language short integers and their external representations.

Library

C Library (**libc.a**)

Syntax

#include <rpc/xdr.h
xdr_short (*xdrs*, *sp*)
XDR **xdrs*;
short **sp*;

Description

The **xdr_short** subroutine is a filter primitive that translates between C language short integers and their external representations.

Parameters

xdrs Points to the eXternal Data Representation (XDR) stream handle.

sp Specifies the address of the short integer.

Return Values

Upon successful completion, this routine returns a value of 1. If unsuccessful, it returns a value of 0.

Implementation Specifics

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

Related Information

Understanding XDR Library Filter Primitives in *Communications Programming Concepts.*

xdr_string Subroutine

Purpose

Translates between C language strings and their external representations.

Library

C Library (**libc.a**)

Syntax

#include <rpc/xdr.h>

xdr_string (*xdrs*, *sp*, *maxsize*)
XDR **xdrs*;
char ***sp*;
u_int *maxsize*;

Description

The **xdr_string** subroutine is a filter primitive that translates between C language strings and their corresponding external representations. Externally, strings are represented as sequences of ASCII characters, while internally, they are represented with character pointers.

Parameters

xdrs
: Points to the eXternal Data Representation (XDR) stream handle.

sp
: Specifies the address of the pointer to the string.

maxsize
: Specifies the maximum length of the string allowed during encoding or decoding. This value is set in a protocol. For example, if a protocol specifies that a file name cannot be longer than 255 characters, then a string cannot exceed 255 characters.

Return Values

Upon successful completion, this routine returns a value of 1. If unsuccessful, it returns a value of 0.

Implementation Specifics

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

Related Information

Understanding XDR Library Filter Primitives in *Communications Programming Concepts.*

xdr_u_char Subroutine

Purpose

Translates between unsigned C language characters and their external representations.

Library

C Library (**libc.a**)

Syntax

#include <rpc/xdr.h>

xdr_u_char (*xdrs*, *ucp*)
XDR **xdrs*;
char **ucp*;

Description

The **xdr_u_char** subroutine is a filter primitive that translates between unsigned C language characters and their external representations.

Parameters

xdrs	Points to the eXternal Data Representation (XDR) stream handle.
ucp	Points to an unsigned integer.

Return Values

Upon successful completion, this routine returns a value of 1. If unsuccessful, it returns a value of 0.

Implementation Specifics

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

Related Information

Understanding XDR Library Filter Primitives in *Communications Programming Concepts*.

xdr_u_int Subroutine

Purpose

Translates between C language unsigned integers and their external representations.

Library

C Library (**libc.a**)

Syntax

#include <rpc/xdr.h>

xdr_u_int (*xdrs*, *up*)
XDR **xdrs*;
u_int **up*;

Description

The **xdr_u_int** subroutine is a filter primitive that translates between C language unsigned integers and their external representations.

Parameters

xdrs	Points to the eXternal Data Representation (XDR) stream handle.
up	Specifies the address of the unsigned long integer number.

Return Values

Upon successful completion, this routine returns a value of 1. If unsuccessful, it returns a value of 0.

Implementation Specifics

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

Related Information

Understanding XDR Library Filter Primitives in *Communications Programming Concepts.*

xdr_u_long Subroutine

Purpose

Translates between C language unsigned long integers and their external representations.

Library

C Library (**libc.a**)

Syntax

#include <rpc/xdr.h>

xdr_u_long (*xdrs*, *ulp*)
XDR **xdrs*;
u_long **ulp*;

Description

The **xdr_u_long** subroutine is a filter primitive that translates between C language unsigned long integers and their external representations.

Parameters

xdrs	Points to the eXternal Data Representation (XDR) stream handle.
ulp	Specifies the address of the unsigned long integer.

Return Values

Upon successful completion, this subroutine returns a value of 1. If unsuccessful, it returns a value of 0.

Implementation Specifics

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

Related Information

Understanding XDR Library Filter Primitives in *Communications Programming Concepts*.

xdr_u_short Subroutine

Purpose

Translates between C language unsigned short integers and their external representations.

Library

C Library (**libc.a**)

Syntax

#include <rpc/xdr.h>

xdr_u_short (*xdrs*, *usp*)
XDR **xdrs*;
u_short **usp*;

Description

The **xdr_u_short** subroutine is a filter primitive that translates between C language unsigned short integers and their external representations.

Parameters

xdrs	Points to the eXternal Data Representation (XDR) stream handle.
usp	Specifies the address of the unsigned short integer.

Return Values

Upon successful completion, this routine returns a value of 1. If unsuccessful, it returns a value of 0.

Implementation Specifics

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

Related Information

Understanding XDR Library Filter Primitives in *Communications Programming Concepts.*

xdr_union Subroutine

Purpose

Translates between discriminated unions and their external representations.

Library

C Library (**libc.a**)

Syntax

#include <rpc/xdr.h>

xdr_union (*xdrs*, *dscmp*, *unp*, *armchoices*, *defaultarm*)
XDR **xdrs*;
enum_t **dscmp*;
char **unp*;
struct xdr_discrim **armchoices*;
xdrproc_t (**defaultarm*) ; /* **may equal NULL** */

Description

The **xdr_union** subroutine is a filter primitive that translates between a discriminated C union and its corresponding external representations. It first translates the discriminant of the union located at the address pointed to by the *dscmp* parameter. This discriminant is always an enum_t value. Next, the union located at the address pointed to by the *unp* parameter is translated. The *armchoices* parameter is a pointer to an array of **xdr_discrim** structures. Each structure contains an ordered pair of parameters[*value, proc*]. If the union's discriminant is equal to the associated *value*, then the *proc* is called to translate the union. The end of the **xdr_discrim** structure array is denoted by a routine of value NULL. If the discriminant is not found in the choices array, then the *defaultarm* procedure is called (if it is not NULL).

Parameters

xdrs	Points to the eXternal Data Representation (XDR) stream handle.
dscmp	Specifies the address of the union's discriminant. The discriminant is an enum_t value.
unp	Specifies the address of the union.
armchoices	Points to an array of **xdr_discrim** structures.
defaultarm	A structure provided in case no discriminants are found.

Return Values

Upon successful completion, this routine returns a value of 1. If unsuccessful, it returns a value of 0.

Implementation Specifics

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

Related Information

Understanding XDR Library Filter Primitives in *Communications Programming Concepts.*

xdr_vector Subroutine

Purpose

Translates between fixed-length arrays and their corresponding external representations.

Library

C Library (**libc.a**)

Syntax

#include <rpc/xdr.h>

xdr_vector (*xdrs*, *arrp*, *size*, *elsize*, *elproc*)
XDR **xdrs*;
char **arrp*;
u_int *size, elsize*;
xdrproc_t *elproc*;

Description

The **xdr_vector** subroutine is a filter primitive that translates between fixed-length arrays and their corresponding external representations.

Parameters

xdrs	Points to the eXternal Data Representation (XDR) stream handle.
arrp	Specifies the the pointer to the array.
size	Specifies the element count of the array.
elsize	Specifies the size of each of the array elements.
elproc	Translates between the C form of the array elements and their external representation. This is an XDR filter.

Return Values

Upon successful completion, this routine returns a value of 1. If unsuccessful, it returns a value of 0.

Implementation Specifics

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

Related Information

Understanding XDR Library Filter Primitives in *Communications Programming Concepts.*

xdr_void Subroutine

Purpose

Supplies an XDR subroutine to the RPC system without transmitting data.

Library

C Library (**libc.a**)

Syntax

#include <rpc/xdr.h>

xdr_void ()

Description

The **xdr_void** subroutine has no function parameters. It may be passed to other Remote Procedure Call (RPC) routines that require a function parameter, where nothing is to be done.

Parameters

This subroutine contains no parameters.

Return Values

This subroutine always returns a value of 1.

Implementation Specifics

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

Related Information

Understanding XDR Library Filter Primitives, Remote Procedure Call (RPC) Overview for Programming in *Communications Programming Concepts.*

xdr_wrapstring Subroutine

Purpose

Calls the **xdr_string** subroutine.

Library

C Library (**libc.a**)

Syntax

#include <rpc/xdr.h>

xdr_wrapstring (*xdrs*, *sp***)**
XDR **xdrs*
char ***sp*

Description

The **xdr_wrapstring** subroutine is a primitive that calls the **xdr_string** subroutine *(xdrs, sp, MAXUN.UNSIGNED);* where the *MAXUN.UNSIGNED* value is the maximum value of an unsigned integer. The **xdr_wrapstring** subroutine is useful because the Remote Procedure Call (RPC) package passes a maximum of two eXternal Data Representation (XDR) routines as parameters, and **xdr_string** requires three.

Parameters

xdrs	Points to the XDR stream handle.
sp	Specifies the address of the pointer to the string.

Return Values

Upon successful completion, this routine returns a value of 1. If unsuccessful, it returns a value of 0.

Implementation Specifics

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

Related Information

The **xdr_string** subroutine.

Understanding XDR Library Filter Primitives in *Communications Programming Concepts.*

xdrmem_create Subroutine

Purpose

Initializes in local memory the XDR stream pointed to by the *xdrs* parameter.

Library

C Library (**libc.a**)

Syntax

#include <rpc/xdr.h>
void
xdrmem_create (*xdrs*, *addr*, *size*, *op***)**
XDR **xdrs*;
char **addr*;
u_int *size*;
enum xdr_op *op*;

Description

The **xdrmem_create** subroutine initializes in local memory the eXternal Data Representation (XDR) stream pointed to by the *xdrs* parameter. The XDR stream data is written to or read from a chunk of memory at the location specified by the *addr* parameter.

Parameters

xdrs	Points to the XDR stream handle.
addr	Points to the memory where the XDR stream data is written to or read from.
size	Specifies the length of the memory in bytes.
op	Specifies the XDR direction. The possible choices are XDR_ENCODE, XDR_DECODE, or XDR_FREE.

Implementation Specifics

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

Related Information

Understanding XDR Non-Filter Primitives in *Communications Programming Concepts*.

xdrrec_create Subroutine

Purpose

Provides an XDR stream that can contain long sequences of records.

Library

C Library (**libc.a**)

Syntax

#include <rpc/xdr.h>

void
xdrrec_create (*xdrs*, *sendsize*, *recvsize*, *handle*, *readit*, *writeit*)
XDR **xdrs*;
u_int *sendsize*;
u_int *recvsize*;
char **handle*;
int (**readit*) (), (**writeit*) ();

Description

The **xdrrec_create** subroutine provides an eXternal Data Representation (XDR) stream that can contain long sequences of records and handle them in both the encoding and decoding directions. The record contents contain data in XDR form. The routine initializes the XDR stream object pointed to by the *xdrs* parameter.

Note: This XDR stream implements an intermediate record stream. Therefore, there are additional bytes in the stream to provide record boundary information.

Parameters

xdrs Points to the XDR stream handle.

sendsize Sets the size of the input buffer where data is written to. If 0 is specified, the buffers are set to the system defaults.

recvsize Sets the size of the output buffer where data is read from. If 0 is specified, the buffers are set to the system defaults.

handle Points to the input/output buffer's handle, which is opaque.

readit Points to the subroutine to call when a buffer needs to be filled. Similar to the **read** system call.

writeit Points to the subroutine to call when a buffer needs to be flushed. Similar to the **write** system call.

Implementation Specifics

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

Related Information

Understanding XDR Non-Filter Primitives in *Communications Programming Concepts.*

xdrrec_endofrecord Subroutine

Purpose

Causes the current outgoing data to be marked as a record.

Library

C Library (**libc.a**)

Syntax

#include <rpc/xdr.h>

xdrrec_endofrecord (*xdrs*, *sendnow*)
XDR **xdrs*;
bool_t *sendnow*;

Description

The **xdrrec_endofrecord** subroutine causes the current outgoing data to be marked as a record and can only be invoked on streams created by the **xdrrec_create** subroutine. The data in the output buffer is marked as a completed record, and the output buffer is optionally written out if the value of the *sendnow* parameter is nonzero.

Parameters

xdrs	Points to the eXternal Data Representation (XDR) stream handle.
sendnow	Specifies whether the record should be flushed to the output **tcp** stream.

Return Values

Upon successful completion, this routine returns a value of 1. If unsuccessful, it returns a value of 0.

Implementation Specifics

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

Related Information

The **xdrrec_create** subroutine.

Understanding XDR Non-Filter Primitives in *Communications Programming Concepts.*

xdrrec_eof Subroutine

Purpose

Checks the buffer for an input stream that indicates the end of file (EOF).

Library

C Library (**libc.a**)

Syntax

#include <rpc/xdr.h>

xdrrec_eof (*xdrs*)
XDR **xdrs*;

Description

The **xdrrec_eof** subroutine checks the buffer for an input stream to see if it reached the end of the file. This routine can only be invoked on streams created by the **xdrrec_create** subroutine.

Parameter

xdrs	Points to the eXternal Data Representation (XDR) stream handle.

Return Values

After consuming the rest of the current record in the stream, this routine returns a value of 1 if the stream has no more input and a value of 0 otherwise.

Implementation Specifics

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

Related Information

The **xdrrec_create** subroutine.

Understanding XDR Non-Filter Primitives in *Communications Programming Concepts*.

xdrrec_skiprecord Subroutine

Purpose

Causes the position of an input stream to move to the beginning of the next record.

Library

C Library (**libc.a**)

Syntax

#include <rpc/xdr.h>

xdrrec_skiprecord (*xdrs*)
XDR **xdrs*;

Description

The **xdrrec_skiprecord** subroutine causes the position of an input stream to move past the current record boundary and onto the beginning of the next record of the stream. This subroutine can only be invoked on streams created by the **xdrrec_create** subroutine. The **xdrrec_skiprecord** subroutine tells the eXternal Data Representation (XDR) implementation that the rest of the current record in the stream's input buffer should be discarded.

Parameter

xdrs	Points to the XDR stream handle.

Return Values

Upon successful completion, this routine returns a value of 1. If unsuccessful, it returns a value of 0.

Implementation Specifics

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

Related Information

The **xdrrec_create** subroutine.

Understanding XDR Non-Filter Primitives in *Communications Programming Concepts*.

xdrstdio_create Subroutine

Purpose

Initializes the XDR data stream pointed to by the *xdrs* parameter.

Library

C Library (**libc.a**)

Syntax

#include <stdio.h>
#include <rpc/xdr.h>

void
xdrstdio_create (*xdrs*, *file*, *op*)
XDR **xdrs*;
FILE **file*;
enum xdr_op *op*;

Description

The **xdrstdio_create** subroutine initializes the eXternal Data Representation (XDR) data stream pointed to by the *xdrs* parameter. The XDR stream data is written to or read from the standard input/output stream pointed to by the *file* parameter.

Note: The destroy routine associated with such XDR stream calls the **fflush** function on the *file* stream, but never calls the **fclose** function.

Parameters

xdrs	Points to the XDR stream handle to initialize.
file	Points to the standard I/O device which data is written to or read from.
op	Specifies an XDR direction. The possible choices are XDR_ENCODE, XDR_DECODE, or XDR_FREE.

Implementation Specifics

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

Related Information

Understanding XDR Non-Filter Primitives in *Communications Programming Concepts*.

xprt_register Subroutine

Purpose

Registers an RPC service transport handle.

Library

C Library (**libc.a**)

Syntax

void
xprt_register (*xprt*)
SVCXPRT **xprt*;

Description

The **xprt_register** subroutine registers a Remote Procedure Call (RPC) service transport handle with the RPC program after the transport has been created. This subroutine modifies the **svc_fds** global variable.

Note: Service implementors do not usually need this subroutine.

Parameter

xprt	Points to the newly created RPC service transport handle.

Implementation Specifics

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

Related Information

eXternal Data Representation (XDR) Overview for Programming in *Communications Programming Concepts*.

xprt_unregister Subroutine

Purpose

Removes an RPC service transport handle.

Library

C Library (**libc.a**)

Syntax

void
xprt_unregister (*xprt*)
SVCXPRT **xprt*;

Description

The **xprt_unregister** subroutine removes a Remote Procedure Call (RPC) service transport handle from the RPC service program before the transport handle can be destroyed. This subroutine modifies the **svc_fds** global variable.

Note: Service implementors do not usually need this subroutine.

Parameter

xprt	Points to the RPC service transport handle to be destroyed.

Implementation Specifics

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

Related Information

eXternal Data Representation (XDR) Overview for Programming in *Communications Programming Concepts*.

yp_all Subroutine

Purpose

Transfers all of the key-value pairs from the network information service (NIS) server to the client as the entire map.

Library

C Library (**libc.a**)

Syntax

#include <rpcsvc/ypclnt.h>
#include <rpcsvc/yp_prot.h>

yp_all (*indomain*, *inmap*, *incallback*)
char **indomain*;
char **inmap*;
struct ypall_CallBack **incallback* {
    int (* foreach) ();
    char **data*;
};
    foreach (*instatus*, *inkey*, *inkeylen*, *inval*, *invallen*, *indata*)

    int *instatus*;
    char **inkey*;
    int *inkeylen*;
    char **inval*;
    int *invallen*;
    char **indata*;

Description

The **yp_all** subroutine provides a way to transfer an entire map from the server to the client in a single request. The routine uses Transmission Control Protocol (TCP) rather than User Datagram Protocol (UDP) as with other functions in this package. This entire transaction takes place as a single Remote Procedure Call (RPC) request and response. The **yp_all** subroutine is used like any other NIS procedure to identify a subroutine and the map in the normal manner. The subroutine is supplied to process each key-value pair within the map.

Note: The remote procedure call is returned to the **yp_all** subroutine only after the transaction is completed (successfully or unsuccessfully), or the **foreach** function decides that it does not want to see any more key-value pairs.

Parameters

indomain Points to the name of the domain used as input to the subroutine.

inmap Points to the name of the map used as input to the subroutine.

incallback Specifies the structure containing the user-defined **foreach** function, which is called for each key-value pair transfered.

instatus Specifies either a return status value of the form NIS_TRUE or an error code. The error codes are defined in the **<rpcsvc/yp_prot.h>** header file.

inkey Points to memory that is private to the **yp_all** subroutine and is overwritten when each new key-value pair arrives. The **foreach** function can use the contents of the memory but does not own the memory itself. Key and value objects presented to the **foreach** function look exactly as they do in the server's map. Objects not terminated by NEWLINE or NULL in the server's map are not terminated by NEWLINE or NULL in the client's map.

inkeylen Returns the length of the *inkey* parameter in bytes.

inval Specifies the value as returned from the server's database.

invallen Specifies the size of the value in bytes.

indata Specifies the contents of the **incallback->data** element passed to the **yp_all** subroutine. The **data** element shares state information between the **foreach** function and the mainline code. It is an optional parameter because no part of the NIS client package inspects its contents.

Return Values

Since the **foreach** subroutine is a Boolean, it returns a value of 0 (zero) to indicate that it is ready to be called again for additional received key-value pairs. It returns a nonzero value to stop the flow of key-value pairs. If the **foreach** function returns a nonzero value, it is not called again, and the **yp_all** subroutine returns a value of 0 (zero).

Implementation Specifics

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

Related Information

Remote Procedure Call (RPC) Overview for Programming in *Communications Programming Concepts*, Understanding Protocols for TCP/IP in *Communication Concepts and Procedures*.

yp_bind Subroutine

Purpose

Used in programs to call the **ypbind** daemon directly for processes that use backup strategies when NIS is not available.

Library

C Library **(libc.a)**

Syntax

#include <rpcsvc/ypclnt.h>
#include <rpcsvc/yp_prot.h>

yp_bind (*indomain*)
char **indomain*;

Description

In order to use network information service (NIS), the client process must be bound to an NIS server that serves the appropriate domain. That is, the client must be associated with a specific NIS server that services the client's requests for NIS information. The NIS lookup processes automatically use the **ypbind** daemon to bind the client, but the **yp_bind** subroutine can be used in programs to call the daemon directly for processes that use backup strategies (for example, a local file) when NIS is not available.

Each NIS binding allocates, or uses up, one client process socket descriptor, and each bound domain uses one socket descriptor. Multiple requests to the same domain use the same descriptor.

Note: If a Remote Procedure Call (RPC) failure status returns from the use of the **yp_bind** subroutine, the domain is unbound automatically. When this occurs, the NIS client tries to complete the operation if the **ypbind** daemon is running and either of the following is true:

- The client process cannot bind a server for the proper domain.
- Remote procedure calls to the server fail.

Parameter

indomain Points to the name of the domain for which to attempt the bind.

Return Values

The NIS client returns control to the user with either an error or a success code if any of the following occurs:

- The error is not related to RPC.
- The **ypbind** daemon is not running.
- The **ypserv** daemon returns the answer.

Implementation Specifics

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

Related Information

The **ypbind** daemon, **ypserv** daemon.

Remote Procedure Call (RPC) Overview for Programming in *Communications Programming Concepts*.

yp_first Subroutine

Purpose

Returns the first key-value pair from the named network information service (NIS) map in the named domain.

Library

C Library **(libc.a)**

Syntax

#include <rpcsvc/ypclnt.h>
#include <rpcsvc/yp_prot.h>

yp_first (*indomain*, *inmap*, *outkey*, *outkeylen*, *outval*, *outvallen***)**
char **indomain*;
char **inmap*;
char ***outkey*;
int **outkeylen*;
char ***outval*;
int **outvallen*;

Description

The **yp_first** routine returns the first key-value pair from the named NIS map in the named domain.

Parameters

indomain Points to the name of the domain used as input to the subroutine.

inmap Points to the name of the map used as input to the subroutine.

outkey Specifies the address of the uninitialized string pointer where the first key is returned. Memory is allocated by the NIS client using the **malloc** subroutine, and may be freed by the application.

outkeylen Returns the length of the *outkey* parameter in bytes.

outval Specifies the address of the uninitialized string pointer where the value associated with the key is returned. Memory is allocated by the NIS client using the **malloc** subroutine, and may be freed by the application.

outvallen Returns the length of the *outval* parameter in bytes.

Return Values

Upon successful completion, this routine returns a value of 0 (zero). If unsuccessful, it returns an error as described in the **<rpcsvc/yp_prot.h>** header file.

Implementation Specifics

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

Related Information

The **malloc** subroutine.

Remote Procedure Call (RPC) Overview for Programming in *Communications Programming Concepts*.

yp_get_default_domain Subroutine

Purpose

Gets the default domain of the node.

Library

C Library **(libc.a)**

Syntax

#include <rpcsvc/ypclnt.h>
#include <rpcsvc/yp_prot.h>

yp_get_default_domain (*outdomain*)
char ***outdomain*;

Description

Network information service (NIS) look-up calls require a map name and a domain name. The client processes can get the default domain of the node by calling the **yp_get_default_domain** routine and using the value returned in the *outdomain* parameter as the input domain *(indomain*) parameter for its NIS remote procedure calls.

Parameter

outdomain Specifies the address of the uninitialized string pointer where the default domain is returned. Memory is allocated by the NIS client using the **malloc** subroutine, and may be freed by the application.

Return Values

Upon successful completion, this routine returns a value of 0 (zero). If unsuccessful, it returns an error as described in the **<rpcsvc/yp_prot.h>** header file.

Implementation Specifics

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

Related Information

The **malloc** subroutine.

Remote Procedure Call (RPC) Overview for Programming in *Communications Programming Concepts*.

yp_master Subroutine

Purpose

Returns the machine name of the NIS master server for a map.

Library

C Library **(libc.a)**

Syntax

#include <rpcsvc/ypclnt.h>
#include <rpcsvc/yp_prot.h>

yp_master (*indomain*, *inmap*,*outname*)
char **indomain*;
char **inmap*;
char ***outname*;

Description

The **yp_master** subroutine returns the machine name of the network information service (NIS) master server for a map.

Parameters

indomain	Points to the name of the domain used as input to the subroutine.
inmap	Points to the name of the map used as input to the subroutine.
outname	Specifies the address of the unitialized string pointer where the name of the domain's **yp_master** server is returned. Memory is allocated by the NIS client using the **malloc** subroutine, and may be freed by the application.

Return Values

Upon successful completion, this routine returns a value of 0 (zero). If unsuccessful, it returns an error as described in the **<rpcsvc/yp_prot.h>** header file.

Implementation Specifics

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

Related Information

The **malloc** subroutine.

Remote Procedure Call (RPC) Overview for Programming in *Communications Programming Concepts*.

yp_match Subroutine

Purpose

Searches for the value associated with a key.

Library

C Library **(libc.a)**

Syntax

#include <rpcsvc/ypclnt.h>
#include <rpcsvc/yp_prot.h>

yp_match (*indomain*, *inmap*, *inkey*, *inkeylen*, *outval*, *outvallen*)
char **indomain*;
char **inmap*;
char **inkey*;
int *inkeylen*;
char ***outval*;
int **outvallen*;

Description

The **yp_match** subroutine searches for the value associated with a key. The input character string entered as the key must match a key in the network information service (NIS) map exactly because pattern matching is not available in NIS.

Parameters

indomain	Points to the name of the domain used as input to the subroutine.
inmap	Points to the name of the map used as input to the subroutine.
inkey	Points to the name of the key used as input to the subroutine.
inkeylen	Specifies the length of the key in bytes.
outval	Specifies the address of the unitialized string pointer where the values associated with the key are returned. Memory is allocated by the NIS client using the **malloc** subroutine, and may be freed by the application.
outvallen	Returns the length of the *outval* parameter in bytes.

Return Values

Upon successful completion, this routine returns a value of 0 (zero). If unsuccessful, it returns an error as described in the **<rpcsvc/yp_prot.h>** header file.

Implementation Specifics

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

Related Information

The **mailoc** subroutine.

Remote Procedure Call (RPC) Overview for Programming in *Communications Programming Concepts.*

yp_next Subroutine

Key Concepts

Purpose

Returns each subsequent value it finds in the named network information service (NIS) map until it reaches the end of the list.

Library

C Library **(libc.a)**

Syntax

#include <rpcsvc/ypclnt.h>
#include <rpcsvc/yp_prot.h>

yp_next (*indomain*, *inmap*, *inkey*, *inkeylen*, *outkey*, *outkeylen*, *outval*, *outvallen*)
char **indomain*;
char **inmap*;
char **inkey*
int *inkeylen*
char ***outkey*;
int **outkeylen*;
char ***outval*;
int **outvallen*;

Description

The **yp_next** subroutine returns each subsequent value it finds in the named NIS map until it reaches the end of the list.

The **yp_next** routine must be preceded by an initial **yp_first** subroutine. Use the *outkey* parameter value returned from the initial **yp_first** subroutine as the value of the *inkey* parameter for the **yp_next** subroutine. The *inkey* parameter values for subsequent calls are retrieved as the nth + second key-value pair. That is, each time the routine returns a key-value to use as the next *inkey* parameter.

The concepts of *first* and *next* depend on the structure of the NIS map being processed. The routines do not retrieve the information in a specific order, such as the lexical order from the original database information files or the numerical sorting order of the keys, values, or key-value pairs. They do show every entry in the NIS map if the **yp_first** subroutine is called on a specific map with the **yp_next** subroutine called repeatedly. The process returns the YPERR_NOMORE message to the user to indicate that every entry in the NIS map has been seen once. If the same sequence of operations is performed on the same map at the same server, the entries are seen in the same order.

Note: If a server operates under a heavy load or fails, the domain can become unbound and then bound again while a client is running. If it binds itself to a different server, it can cause entries to be seen twice or not be seen at all. The domain rebinds itself to protect the enumeration process from being interrupted before it completes. Avoid this situation by returning all of the keys and values with the **yp_all** subroutine.

Parameters

indomain Points to the name of the domain used as input to the subroutine.

inmap Points to the name of the map used as input to the subroutine.

inkey Points to the key that is used as input to the subroutine.

inkeylen Returns the length of the *inkey* parameter in bytes.

outkey Specifies the address of the unitialized string pointer where the first key is returned. Memory is allocated by the NIS client using the **malloc** subroutine, and may be freed by the application.

outkeylen Returns the length of *outkey* in bytes.

outval Specifies the address of the unitialized string pointer where the values associated with the key are returned. Memory is allocated by the NIS client using the **malloc** subroutine, and may be freed by the application.

outvallen Returns the length of the *outval* parameter in bytes.

Return Values

Upon successful completion, this routine returns a value of 0. If unsuccessful, it returns an error as described in the **<rpcsvc/yp_prot.h>** header file.

Implementation Specifics

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

Related Information

The **malloc** subroutine, **yp_all** subroutine, **yp_first** subroutine.

Remote Procedure Call (RPC) Overview for Programming in *Communications Programming Concepts.*

yp_order Subroutine

Purpose

Returns the order number for a network information service (NIS) map that identifies when the map was built.

Library

C Library **(libc.a)**

Syntax

#include <rpcsvc/ypclnt.h>
#include <rpcsvc/yp_prot.h>

yp_order (*indomain*, *inmap*, *outorder*)
char **indomain*;
char **inmap*;
int **outorder*;

Description

The **yp_order** subroutine returns the order number for an NIS map that identifies when the map was built. The number determines whether the local map is the most current version or the master NIS database has a more current one.

Parameters

indomain	Points to the name of the domain used as input to the subroutine.
inmap	Points to the name of the map used as input to the subroutine.
outorder	Points to the returned order number, which is a ten-digit ASCII integer that represents the AIX time, in seconds, when the map was built.

Return Values

Upon successful completion, this routine returns a value of 0 (zero). If unsuccessful, it returns an error as described in the **<rpcsvc/yp_prot.h>** header file.

Implementation Specifics

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

Related Information

Remote Procedure Call (RPC) Overview for Programming in *Communications Programming Concepts*.

yp_unbind Subroutine

Purpose

Manages socket descriptors for processes that access multiple domains.

Library

C Library **(libc.a)**

Syntax

#include <rpcsvc/ypclnt.h>
#include <rpcsvc/yp_prot.h>

void yp_unbind (*indomain*)
char **indomain*;

Description

The **yp_unbind** subroutine is available to manage socket descriptors for processes that access multiple domains. When the **yp_unbind** subroutine is used to free a domain, all per-process and per-node resources that were used to bind it are also freed.

Parameter

indomain Points to the name of the domain used as input to the subroutine.

Return Values

Upon successful completion, this routine returns a value of 0 (zero). If unsuccessful, it returns an error as described in the **<rpcsvc/yp_prot.h>** header file.

Implementation Specifics

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

Related Information

The **ypbind** daemon.

The **yp_bind** subroutine.

Sockets Overview in *Communications Programming Concepts.*

yp_update Subroutine

Purpose

Used to make changes to the network information service (NIS) map.

Library

C Library **(libc.a)**

Syntax

#include <rpcsvc/ypclnt.h>
#include <rpcsvc/yp_prot.h>

yp_update (*indomain, inmap, ypop, inkey, inkeylen, indata, indatalen*)
char **indomain*;
char **inmap*;
unsigned *ypop*;
char **inkey*;
int *inkeylen*;
char **indata*;
int *indatalen*;

Description

The **yp_update** subroutine is used to make changes to the NIS map. The syntax is the same as that of the **yp_match** subroutine except for the extra parameter *ypop* which may take on one of the following four values:

ypop _INSERT	Inserts the key-value pair into the map. If the key already exists in the map, the **yp_update** subroutine returns a value of YPERR_KEY.
ypop_CHANGE	Changes the data associated with the key to the new value. If the key is not found in the map, the **yp_update** subroutine returns a value of YPERR_KEY.
ypop_STORE	Stores an item in the map whether or not it already exists. No error will be returned if the key exists already or does not exist.
ypop_DELETE	Deletes an entry from the map.

Note: This routine depends upon secure Remote Procedure Call (RPC), and will not work unless the network is running secure RPC.

Parameters

indomain	Points to the name of the domain used as input to the subroutine.
inmap	Points to the name of the map used as input to the subroutine.
ypop	Specifies the update operation to be used as input to the subroutine.
inkey	Points to the input key to be used as input to the subroutine.
inkeylen	Specifies the length of the *inkey* parameter in bytes.

indata	Points to the data used as input to the subroutine.
indatalen	Specifies the length of the data in bytes used as input to the subroutine.

Return Values

Upon successful completion, this routine returns a value of 0. If unsuccessful, it returns an error as described in the **<rpcsvc/yp_prot.h>** header file.

Implementation Specifics

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

Related Information

The **/etc/yp/updaters** file.

The **yp_match** subroutine.

yperr_string Subroutine

Purpose

Returns a pointer to an error message string.

Library

C Library **(libc.a)**

Syntax

#include <rpcsvc/ypclnt.h>
#include <rpcsvc/yp_prot.h>

char *yperr_string (*incode***)**
int *incode*;

Description

The **yperr_string** routine returns a pointer to an error message string. The error message string is null-terminated but contains no period or new-line escape characters.

Parameter

incode	Contains network information service (NIS) error code as described in the **<rpcsvc/yp_prot.h>** header file.

Return Values

This routine returns a pointer to an error message string corresponding to the *incode* parameter.

Implementation Specifics

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

Related Information

Remote Procedure Call (RPC) Overview for Programming in *Communications Programming Concepts.*

ypprot_err Subroutine

Purpose

Takes a network information service (NIS) protocol error code as input, and returns an error code to be used as input to a **yperr_string** subroutine.

Library

C Library **(libc.a)**

Syntax

#include <rpcsvc/ypclnt.h>
#include <rpcsvc/yp_prot.h>

ypprot_err (*incode*)
u_int *incode*;

Description

The **ypprot_err** subroutine takes an NIS protocol error code as input, and returns an error code to be used as input to a **yperr_string** subroutine.

Parameter

incode	Specifies an NIS protocol error used as input to the subroutine.

Return Values

This routine returns a corresponding error code to be passed to the **yperr_string** subroutine.

Implementation Specifics

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

Related Information

The **yperr_string** subroutine.

Remote Procedure Call (RPC) Overview for Programming in *Communications Programming Concepts.*

Simple Network Management Protocol (SNMP)

aix_exec Function

Purpose

Executes AIX programs and commands from within a virtual G machine environment.

Syntax

(int) aix_exec (*Command*)
string *Command*;

Description

The **aix_exec** function uses the AIX **exec** subroutine to execute programs and commands. These execute as separate processes outside the VGM environment. The standard input file is not opened for the spawned processes, so they cannot read input. The VGM does not wait for the spawned process to terminate.

Parameter

Command Specifies the AIX command to be invoked. This parameter must be a string.

Return Values

The **aix_exec** function returns the process ID of the spawned process. There are no error return codes.

Implementation Specifics

This intrinsic function is part of Simple Network Management Protocol (SNMP) Manager in AIX Network Management/6000.

Related Information

Alphabetic List of Intrinsic Functions, Functional List of Intrinsic Functions, How to Create **xgmon** Intrinsic Functions, How to Create **xgmon** Library Commands in *Communications Programming Concepts*.

Using Intrinsic Functions in *Communications Programming Concepts*.

alloc Function

Purpose

Makes a specified amount of storage space available and returns a pointer to the newly allocated space.

Syntax

(pointer) alloc (*Words***)**
int *Words*;

Description

The **alloc** function dynamically allocates a specified amount of storage space in memory. This space is measured in units of 32 bits. These units are also referred to as *words.* Once it allocates space, the **alloc** function returns a pointer that points to the newly allocated space.

Note: This is useful for building arrays.

Parameter

Words Specifies the amount of space, in units of 32 bits, to be made available. This parameter must be an integer data type.

Return Value

The **alloc** function returns a pointer that points to the allocated space.

Implementation Specifics

This intrinsic function is part of Simple Network Management Protocol (SNMP) Manager in AIX Network Management/6000.

Related Information

Alphabetic List of Intrinsic Functions, Functional List of Intrinsic Functions, How to Create **xgmon** Intrinsic Functions, How to Create **xgmon** Library Commands in *Communications Programming Concepts.*

Using Intrinsic Functions in *Communications Programming Concepts.*

ascii Function

Purpose

Returns the integer ASCII value of the first character in the specified string.

Syntax

(int) ascii (*String*)
string *String*;

Description

The **ascii** function returns the integer ASCII value of the first character in the parameter string. It is useful, in conjunction with the **mid** function, for handling binary data embedded within a string.

Parameter

String	Specifies the string in which the ASCII value of the first character is requested.

Return Value

The **ascii** function returns the integer ASCII value of the first character in the parameter string.

Implementation Specifics

This intrinsic function is part of Simple Network Management Protocol (SNMP) Manager in AIX Network Management/6000.

Related Information

The **mid** function.

Alphabetic List of Intrinsic Functions, Functional List of Intrinsic Functions, How to Create **xgmon** Intrinsic Functions in *Communications Programming Concepts*.

Using Intrinsic Functions in *Communications Programming Concepts*.

base_type Function

Purpose

Takes a Management Information Database (MIB) numeric–format variable name or numeric–format instance ID and returns a number that indicates its base type.

Syntax

(int) base_type (*ObjectID*)
string *ObjectID*;

Description

The **base_type** function takes an MIB numeric–format variable name or numeric–format instance ID and returns a number that indicates its base type.

Note: See RFC 1066 for further information.

Parameter

ObjectID	Specifies the MIB numeric–format variable name or numeric–format instance ID whose base type is queried. This parameter must be a string data type.

Return Values

If the parameter identifies an integer, the **base_type** intrinsic function returns a 1. If the parameter identifies a string, the **base_type** intrinsic function returns a 2.

Implementation Specifics

This intrinsic function is part of Simple Network Management Protocol (SNMP) Manager in AIX Network Management/6000.

Related Information

Alphabetic List of Intrinsic Functions, Functional List of Intrinsic Functions, How to Create **xgmon** Intrinsic Functions in *Communications Programming Concepts*.

Understanding the Simple Network Management Protocol (SNMP), Understanding the Management Information Base (MIB), Understanding Terminology Related to Management Information Base (MIB) Variables, Working with Management Information Base (MIB) Variables in *Communications Programming Concepts*.

Using Intrinsic Functions in *Communications Programming Concepts*.

close Function

Purpose

Closes the open file indicated by the specified file descriptor.

Syntax

(int) close (*FileDescriptor*)
int *FileDescriptor*;

Description

The **close** function closes the open file indicated by the *FileDescriptor* parameter.

Parameter

FileDescriptor	File descriptor. This parameter must be an integer.

Return Values

The **close** function returns 0 (zero) if the file closes successfully; otherwise, it returns –1.

Implementation Specifics

This intrinsic function is part of Simple Network Management Protocol (SNMP) Manager in AIX Network Management/6000.

Related Information

The **fopen** function.

Alphabetic List of Intrinsic Functions, Functional List of Intrinsic Functions, How to Create **xgmon** Intrinsic Functions, How to Create **xgmon** Library Commands in *Communications Programming Concepts*.

Using Intrinsic Functions in *Communications Programming Concepts*.

create_SNMP_port Subroutine

Purpose

Creates a UDP socket to communicate with an SNMP agent.

Syntax

extern struct sockaddr_in *snmp_dest*;
extern int *SNMP_port*;
int create_SNMP_port (*agent_address*)
unsigned long *agent_address*;

Description

The **create_SNMP_port** subroutine creates a UDP socket and prepares the structure specified by the *snmp_dest* parameter for communication with an SNMP agent specified by the *agent_address* parameter.

Note: This subroutine should only be called *once*. It does *not* support opening multiple sockets for concurrent communication with several agents.

Parameter

agent_address Specifies the agent with which to communicate.

External Variables

snmp_dest The structure that contains the socket address prepared for communication by the **create_SNMP_port** subroutine.

SNMP_port The file descriptor that denotes the UDP socket created for communication by the **create_SNMP_port** subroutine.

Return Values

If an error occurs, the **create_SNMP_port** subroutine returns –1. Otherwise, it returns 0 (zero).

Note: The file descriptor for the socket is stored in the *SNMP_port* external variable. A socket address is stored in the *snmp_dest* external variable.

Implementation Specifics

This subroutine is part of SNMP Application Programming Interface in AIX Network Management/6000.

Related Information

The **SNMP_errormsg** array.

Using the SNMP API Subroutine Library, Understanding the Simple Network Management Protocol (SNMP) in *Communications Programming Concepts*.

ctime Function

Purpose

Generates a text string that corresponds to an integer expression of time.

Syntax

(string) ctime (*TimeExpr***)**
int *TimeExpr*;

Description

The **ctime** function generates a text string that corresponds to the time specified by the *TimeExpr* parameter. Note that the **ctime** function does not add a new-line character to the end of the string, while the C library function does.

Parameter

TimeExpr Specifies the time to be expressed. This parameter must be an integer.

Return Value

The **ctime** function returns a string of text characters that expresses the time as an integer.

Implementation Specifics

This intrinsic function is part of Simple Network Management Protocol (SNMP) Manager in AIX Network Management/6000.

Related Information

The **time** function.

Alphabetic List of Intrinsic Functions, Functional List of Intrinsic Functions, How to Create **xgmon** Intrinsic Functions, How to Create **xgmon** Library Commands in *Communications Programming Concepts*.

Using Intrinsic Functions in *Communications Programming Concepts*.

dep_info Function

Purpose

Returns information about a display element.

Syntax

(string) dep_info (*ElementName***)**
string *ElementName*;

Description

The **dep_info** function returns information about a display element. If the name passed does not refer to a display element, the null string is returned. The type of string returned depends on the type of the display element specified.

Parameter

ElementName Specifies the name of the element about which the **dep_info** function is to get information. This parameter must be a string data type. To name a link, this parameter takes the form:

```
hostname1<–>hostname2
```

Return Values

Returns information about a specific display element, for instance:

1. When information is requested on a node, the **xgmon** program returns the following:

   ```
   x,y|n:
   ```

   In this example, the "n" indicates the display element is a node. The x and y coordinates indicate the position of the display element on the screen.

2. When information is requested on a host, the **xgmon** program returns the following:

   ```
   x,y|h:addr1,addr2,addr3,...
   ```

   In this example, the "h" indicates the display element is a host. The x and y coordinates indicate the position of the display element on the screen. Also, the IP addresses associated with the host are listed.

3. When information is requested on a link, the **xgmon** program returns the following:

   ```
   x,y|l:from_addr,to_addr
   ```

   In this example, the | (bar) indicates the display element is a link. The x and y coordinates are meaningless in this case, but the format is identical to the previous examples to permit easier parsing. Also, the IP addresses indicate where each end of the link connects.

4. If the name passed does not refer to a display element, the null string is returned.

Implementation Specifics

This intrinsic function is part of Simple Network Management Protocol (SNMP) Manager in AIX Network Management/6000.

Related Information

Alphabetic List of Intrinsic Functions, Functional List of Intrinsic Functions, How to Create **xgmon** Intrinsic Functions, How to Create **xgmon** Library Commands in *Communications Programming Concepts*.

Using Intrinsic Functions in *Communications Programming Concepts*.

dotaddr Function

Purpose

Returns a string representing the IP address in dot notation.

Syntax

(string) dotaddr (*IPAddress*)
int *IPAddress*;

Description

The **dotaddr** function returns a string representing the specified IP address in dot notation.

Parameter

IPAddress Specifies the IP address. This parameter must be an integer.

Return Value

The **dotaddr** function returns a string representing the IP address in dot notation (for example, 129.35.1.1).

Implementation Specifics

This intrinsic function is part of Simple Network Management Protocol (SNMP) Manager in AIX Network Management/6000.

Related Information

The **hostname** function, **ipaddr** function.

Alphabetic List of Intrinsic Functions, Functional List of Intrinsic Functions, How to Create **xgmon** Intrinsic Functions in *Communications Programming Concepts*.

Using Intrinsic Functions in *Communications Programming Concepts*.

draw_line Function

Purpose

Draws a line.

Syntax

(int) draw_line (*x1*, *y1*, *x2*, *y2*, *Width*, *Color*)
int *x1*, *y1*, *x2*, *y2*;
int *Width*;
int *Color*;

Description

The **draw_line** function draws a line from pixel point (*x1,y1*) to pixel point (*x2,y2*) on the display associated with the virtual G machine. The width of the line is specified by the *Width* parameter,and is drawn in the color or style indicated by the *Color* parameter. On a monochrome display, this can be either 1 or 2, indicating a solid or dotted line, respectively. On color displays, the color specification can be 1, 2, 3, 4, 5, 6, 7, or 8 corresponding to the color for up, unknown, down, background, white, acknowledged, ignored, and inactive, respectively.

Parameters

x1, y1, x2, y2
: Indicate, in pixel points, the exact location of a line on the display. These parameters are integers.

Width
: Indicates the width, in pixel points, of the line. This parameter is an integer.

Color
: Indicates the color of the line or, if the display is monochrome, indicates whether the line is solid or dotted. This parameter is an integer.

Return Values

If the line displays successfully, the **draw_line** function returns 0 (zero). Otherwise, the **draw_line** function returns –1.

Implementation Specifics

This intrinsic function is part of Simple Network Management Protocol (SNMP) Manager in AIX Network Management/6000.

Related Information

Alphabetic List of Intrinsic Functions, Functional List of Intrinsic Functions, How to Create **xgmon** Intrinsic Functions, How to Create **xgmon** Library Commands in *Communications Programming Concepts*.

Using Intrinsic Functions in *Communications Programming Concepts*.

draw_string Function

Purpose

Enables the display of formatted output in color.

Syntax

(int) draw_string (*Format*, *Argument* ..., *Color*)
string *Format*;
DataType Argument;
int *Color*;

Description

The **draw_string** function is used to display formatted output on the graphics window associated with the virtual G machine. The string is drawn in the color indicated by the *Color* parameter. On a color display, this can be 1, 2, 3, 4, 5, 6, 7, or 8 corresponding to the color for up, unknown, down, background, white, acknowledged, ignored, and inactive, respectively. On a monochrome display, the color specification can only be white.

Parameters

Format	This parameter must be a string.
Argument	This parameter can be an integer, string, or pointer.
Color	Indicates the color of the string. If the display is monochrome, the color is white. This parameter is an integer.

Note: The *Format* string can be specified as permitted by the **printf** subroutine.

Return Values

If the string displays successfully, the **draw_string** function returns 0 (zero). If the string fails to display, the **draw_string** function returns –1.

Examples

1. `draw_string ("%c%c%c%s", 27, 1, 1, "hello world", 3);`

   In this example, the **draw_string** function writes `hello world` to the upper left corner in the color for down.

2. `draw_string ("%s", "hello world", 2);`

   In this example, the **draw_string** function writes `hello world` at the current cursor position in the color for unknown.

Implementation Specifics

This intrinsic function is part of Simple Network Management Protocol (SNMP) Manager in AIX Network Management/6000.

Related Information

The **printf** command.

Alphabetic List of Intrinsic Functions, Functional List of Intrinsic Functions, How to Create **xgmon** Intrinsic Functions, How to Create **xgmon** Library Commands in *Communications Programming Concepts*.

Using Intrinsic Functions in *Communications Programming Concepts*.

exec Function

Purpose

Allows a virtual G machine to start execution of a library command in another virtual G machine or allows a virtual G machine to issue a system command.

Syntax

(int) exec (*Command*)
string *Command*;

Description

The **exec** function allows a virtual G machine to start execution of a library command in another virtual G machine or to issue a system command, such as the **compile** command. If the **exec** function successfully executes a library command, it returns the machine ID of the virtual G machine on which the command is to be executed. If there are no free virtual G machines, the **exec** function returns –1. If a system command is invoked, a 0 (zero) is returned. If the command is not recognized as a system command or a library command, the **exec** function returns –2.

Parameter

Command	Specifies the library command or system command to be executed. This parameter must be a string data type.

Return Values

Machine ID	If a library command is successfully executed, the **exec** function returns the ID of the virtual G machine on which the command is to be executed.
0	This value is returned if a system command is invoked.
–1	This value is returned if no machines are available to execute the library command.
–2	This value is returned if the command is not recognized as a system command or a library command.

Implementation Specifics

This intrinsic function is part of Simple Network Management Protocol (SNMP) Manager in AIX Network Management/6000.

Related Information

Alphabetic List of Intrinsic Functions, Functional List of Intrinsic Functions, How to Create **xgmon** Intrinsic Functions, How to Create **xgmon** Library Commands in *Communications Programming Concepts*.

Using Intrinsic Functions in *Communications Programming Concepts*.

extract_SNMP_name Subroutine

Purpose

Extracts the variable name portion of a numeric–format instance ID.

Syntax

char *extract_SNMP_name (*instance_id*)
char **instance_id*;

Description

An instance ID consists of a variable name followed by an instance value. The **extract_SNMP_name** subroutine accepts instance IDs in numeric format, and returns a pointer to the numeric–format variable name. The returned name is terminated by a . (dot) so that an instance value can be directly concatenated to it.

Parameter

instance_id A pointer to an instance ID in numeric format.

Return Values

If the *instance_id* parameter contains a variable name registered in the **/etc/mib_desc** file, a pointer to that name (in numeric format) is returned. Otherwise, a pointer to the empty string is returned.

Example

1. The following line returns a pointer to `"1.3.6.1.2.1.4.21.1.10."`:

   ```
   extract_SNMP_name ("1.3.6.1.2.1.4.21.1.10.127.0.0.1");
   ```

   Note: An instance ID value of `"ipRouteAge.127.0.0.1"` is invalid since the *instance_id* parameter must be numeric.

Implementation Specifics

This subroutine is part of SNMP Application Programming Interface in AIX Network Management/6000.

File

/etc/mib_desc Defines the Management Information Base (MIB) variables.

Related Information

The **SNMP_errormsg** array.

The **lookup_SNMP_group** subroutine, **lookup_SNMP_name** subroutine.

Using the SNMP API Subroutine Library, Understanding the Simple Network Management Protocol (SNMP), Understanding the Management Information Base (MIB), Understanding Terminology Related to Management Information Base (MIB) Variables, Working with Management Information Base (MIB) Variables in *Communications Programming Concepts*.

flush_trap Function

Purpose

Flushes the current trap that is being processed.

Syntax

(int) flush_trap (*Flag*)
int *Flag*;

Description

The **flush_trap** function is used to flush the current trap that is being processed. It returns the number of traps pending (this number is also available in the **traps_pending** global variable.) Normally, 0 (zero) is passed, and only the current trap is flushed. If a value other than 0 (zero) is passed, all of the pending traps are flushed.

Parameter

Flag Specifies either a zero or nonzero integer.

Return Value

Returns the number of traps pending.

Implementation Specifics

This intrinsic function is part of Simple Network Management Protocol (SNMP) Manager in AIX Network Management/6000.

Related Information

Alphabetic List of Intrinsic Functions, Functional List of Intrinsic Functions, How to Create **xgmon** Intrinsic Functions, How to Create **xgmon** Library Commands in *Communications Programming Concepts.*

Using Intrinsic Functions in *Communications Programming Concepts.*

Working with Virtual G Machine (VGM) Variables in *Communications Programming Concepts.*

font_height Function

Purpose

Returns the height, in pixels, of the font being used in the graphics window associated with a virtual G machine.

Syntax

(int) font_height(0)

Description

The **font_height** function returns the height, in pixels, of the font being used in the graphics window associated with the virtual G machine in which the program is running.

Parameter

Dummy parameter 0 (zero) is required.

Return Values

The **font_height** function returns the height of the font being used in the graphics window. If there is no window associated with the virtual G machine, –1 is returned.

Implementation Specifics

This intrinsic function is part of Simple Network Management Protocol (SNMP) Manager in AIX Network Management/6000.

Related Information

The **font_width** function.

Alphabetic List of Intrinsic Functions, Functional List of Intrinsic Functions, How to Create **xgmon** Intrinsic Functions, How to Create **xgmon** Library Commands in *Communications Programming Concepts*.

Using Intrinsic Functions in *Communications Programming Concepts*.

font_width Function

Purpose

Returns the width, in pixels, of the font being used in the graphics window associated a virtual G machine.

Syntax

(int) font_width(0)

Description

The **font_width** function returns the width, in pixels, of the font being used in the graphics window associated with the virtual G machine in which the program is running.

Parameter

Dummy parameter 0 (zero) is required.

Return Values

The **font_width** function returns the width of the font being used in the graphics window. If there is no window associated with the virtual G machine, –1 is returned.

Implementation Specifics

This intrinsic function is part of Simple Network Management Protocol (SNMP) Manager in AIX Network Management/6000.

Related Information

The **font_height** function.

Alphabetic List of Intrinsic Functions, Functional List of Intrinsic Functions, How to Create **xgmon** Intrinsic Functions, How to Create **xgmon** Library Commands in *Communications Programming Concepts.*

Using Intrinsic Functions in *Communications Programming Concepts.*

fopen Function

Purpose

Opens the file indicated by the specified file name.

Syntax

(int) fopen (*File*, *AccessMode*)
string *File*;
string *AccessMode*;

Description

The **fopen** function is used to open the file specified by the *File* parameter. Three values for *AccessMode* are recognized. The first two open the file for writing using the **print ... to** statement; the third opens the file for read using the **read** function.

The integer value returned is a file descriptor to be used in the **to** clause of a **print** statement or passed as an argument to the **read** function. Files are automatically closed when a virtual G machine is halted but can be closed by the **close** function.

Note: Although **xgmon** normally runs setuid to the root user, file opens are validated with the permissions associated with the user running the **xgmon** client instead of the unlimited permissions associated with root privileges.

Parameters

File — The name of the file including the path name. This parameter must be a string data type.

AccessMode — Indicates how the file is to be opened as follows:

- *w* — Creates or truncates a file.
- *a* — Appends a file. If the file does not exist, append mode creates the file.
- *r* — Reads a file.

This parameter must be a string data type.

Return Values

Returns the file descriptor (integer value) if the file opens successfully, or returns 0 (zero) if the file fails to open.

Example

1.
```
int fd;
string filename;
filename="my_file";
fd = (int) fopen (filename, "r");
```

Implementation Specifics

This intrinsic function is part of Simple Network Management Protocol (SNMP) Manager in AIX Network Management/6000.

Related Information

The **close** function, **read** function.

Alphabetic List of Intrinsic Functions, Functional List of Intrinsic Functions, How to Create **xgmon** Intrinsic Functions, How to Create **xgmon** Library Commands in *Communications Programming Concepts*.

Using Intrinsic Functions in *Communications Programming Concepts*.

get_deps Function

Purpose

Returns a list of display elements that are grouped under a particular node.

Syntax

(string) get_deps (*ElementName***)**
string *ElementName*;

Description

The **get_deps** function makes available the hierarchy of display elements that make up the current active topology description. This function is passed the name of a node and returns a list of display elements that are grouped underneath it. The pseudo–root element is specified by the null string.

If the specified display element does not exist or is not a node with display elements grouped under it, the null string is returned.

Parameter

ElementName	Specifies the name of the node about which information is desired. This parameter must be a string data type.

Return Values

Returns the null string if the specified display element does not exist or if the specified display element is not a node with display elements grouped under it. Otherwise, the **get_deps** function returns a list of the display elements that are grouped under the display element.

Implementation Specifics

This intrinsic function is part of Simple Network Management Protocol (SNMP) Manager in AIX Network Management/6000.

Related Information

Alphabetic List of Intrinsic Functions, Functional List of Intrinsic Functions, How to Create **xgmon** Intrinsic Functions, How to Create **xgmon** Library Commands in *Communications Programming Concepts*.

Using Intrinsic Functions in *Communications Programming Concepts*.

get_MIB_base_type Subroutine

Purpose

Returns a value indicating the base type of a Management Information Base (MIB) variable.

Syntax

int get_MIB_base_type (*object_id*)
char **object_id*;

Description

The **get_MIB_base_type** subroutine returns a value indicating the base type of the specified variable. These types are defined by RFC 1066 for the standard MIB.

Parameter

object_id	Specifies the MIB variable name in numeric format.

Return Values

If the numeric-format MIB variable name is unrecognized, –1 is returned. Otherwise, one of the following values is returned:

1	unsigned long
2	string.

Implementation Specifics

This subroutine is part of SNMP Application Programming Interface in AIX Network Management/6000.

File

/etc/mib_desc Defines the Management Information Base (MIB) variables.

Related Information

The **SNMP_errormsg** array.

The **get_MIB_name** subroutine, **get_MIB_variable_type** subroutine, **lookup_SNMP_name** subroutine.

Using the SNMP API Subroutine Library, Understanding the Simple Network Management Protocol (SNMP), Understanding the Management Information Base (MIB), Understanding Terminology Related to Management Information Base (MIB) Variables, Working with Management Information Base (MIB) Variables in *Communications Programming Concepts*.

get_MIB_group Function

Purpose

Finds the set of all Management Information Base (MIB) variable names that contain a given text string as a prefix.

Syntax

(pointer) get_MIB_group (*Prefix*)
string *Prefix*;

Description

The **get_MIB_group** intrinsic function searches all text–format names in the **/etc/mib_desc** file and extracts those that contain the string specified by the *Prefix* parameter. The search is not case–sensitive. A pointer to an array of the numeric–format names is returned. Each numeric–format name is terminated with a . (dot) so that an instance can be directly concatenated with it.

Note: See RFC 1066 for further information.

Parameter

Prefix	Specifies a prefix of a group of MIB variable names in text format. This parameter must be of the string data type.

Return Values

If matching names are found, a pointer to an array of strings containing the matching names is returned. Otherwise, a pointer to the empty string ("") is returned.

Example

1. The following example obtains a list of MIB variables that contain the `if` prefix, and prints out all the numeric–format variables in the list:

```
pointer  list;
string   variable;
int      i;

list = (pointer) get_MIB_group("if");
variable = list[0];
i = 0;
while (variable != "") {
  print "\nvariable = %s", variable;
  i = i + 1;
  variable = list[i];
}
```

Implementation Specifics

This intrinsic function is part of Simple Network Management Protocol Manager in AIX Network Management/6000.

Files

/etc/mib_desc Defines the Management Information Base (MIB) variables. The user specifies a time-to-live (TTL) value (in seconds) for each variable.

Related Information

Alphabetic List of Intrinsic Functions, Functional List of Intrinsic Functions, How to Create **xgmon** Intrinsic Functions in *Communications Programming Concepts.*

Understanding the Simple Network Management Protocol (SNMP), Understanding the Management Information Base (MIB), Understanding Terminology Related to Management Information Base (MIB) Variables, Working with Management Information Base (MIB) Variables in *Communications Programming Concepts.*

Using Intrinsic Functions in *Communications Programming Concepts.*

get_MIB_name Subroutine

Purpose

Returns the text name of a Management Information Base (MIB) variable.

Syntax

char *get_MIB_name (*var_name*)
char **var_name*;

Description

The **get_MIB_name** subroutine maps the numeric-format variable name specified by the *var_name* parameter to the corresponding text name. These names are defined by RFC 1066 for the standard MIB.

Parameter

var_name Specifies the MIB variable name in numeric format.

Return Values

The text name corresponding to the numeric-format variable name specified by the *var_name* parameter is returned. If the variable name is unrecognized, the null string is returned.

Example

1. If the *var_name* parameter is `"1.3.6.1.2.1.1.1"`, a pointer to the string `"sysDescr"` is returned.

   Note: A variable name value of `"sysDescr"` is invalid since the *var_name* parameter must be in numeric format.

Implementation Specifics

This subroutine is part of SNMP Application Programming Interface in AIX Network Management/6000.

File

/etc/mib_desc Defines the Management Information Base (MIB) variables.

Related Information

The **SNMP_errormsg** array.

The **get_MIB_base_type** subroutine, **get_MIB_variable_type** subroutine.

Using the SNMP API Subroutine Library, Understanding the Simple Network Management Protocol (SNMP), Understanding the Management Information Base (MIB), Understanding Terminology Related to Management Information Base (MIB) Variables, Working with Management Information Base (MIB) Variables in *Communications Programming Concepts.*

get_MIB_variable_type Subroutine

Purpose

Returns a value indicating the variable type of a Management Information Base (MIB) variable.

Syntax

int get_MIB_variable_type (*var_name*)
char **var_name*;

Description

The **get_MIB_variable_type** subroutine returns a value indicating the type of the specified variable. These types are defined by RFC 1066 for the standard MIB.

Parameter

var_name Specifies the MIB variable name in numeric format.

Return Values

If the variable name is unrecognized, –1 is returned. Otherwise, one of the following values is returned:

1 number

2 string

3 object identifier

4 empty

5 internet address

6 counter

7 gauge

8 time ticks.

Implementation Specifics

This subroutine is part of SNMP Application Programming Interface in AIX Network Management/6000.

File

/etc/mib_desc Defines the Management Information Base (MIB) variables.

Related Information

The **SNMP_errormsg** array.

The **get_MIB_base_type** subroutine, **get_MIB_name** subroutine.

Using the SNMP API Subroutine Library, Understanding the Simple Network Management Protocol (SNMP), Understanding the Management Information Base (MIB), Understanding Terminology Related to Management Information Base (MIB) Variables, Working with Management Information Base (MIB) Variables in *Communications Programming Concepts.*

get_primary Function

Purpose

Returns the current primary address associated with the specified host.

Syntax

(int) get_primary (*HostName*)
string *HostName*;

Description

The **get_primary** function returns the current primary address associated with the specified host. The primary address can be changed by the **next_alternate** function. These two functions are used to implement adaptive, alternate addresses, permitting **xgmon** applications to adapt to the failure of an interface on a network element the **xgmon** program is monitoring. The designated host should be fully described by the current topology description. Most applications would want to use this function instead of the **ipaddr** function.

Parameter

HostName Specifies the name of the host to be queried. This parameter must be a string data type.

Return Value

Returns the primary address of the host queried.

Implementation Specifics

This intrinsic function is part of Simple Network Management Protocol (SNMP) Manager in AIX Network Management/6000.

Related Information

The **next_alternate** function, **ipaddr** function.

Alphabetic List of Intrinsic Functions, Functional List of Intrinsic Functions, How to Create **xgmon** Intrinsic Functions in *Communications Programming Concepts.*

Using Intrinsic Functions in *Communications Programming Concepts.*

getenv Function

Purpose

Obtains the value of a user–defined environment variable for a host.

Syntax

(string) getenv (*DisplayElementName*, *VariableName*)
string *DisplayElementName*;
string *VariableName*;

Description

The **getenv** function retrieves the value of a user–defined environment variable associated with the specified display element. The **xgmon** program recognizes the **RIGHTCLICK** environment variable name as being associated with the name of the library command that should be run when the display element is double–clicked using the right mouse button.

Parameters

DisplayElementName
: Specifies the name or IP address (in dot notation) of the display element for which an environment variable is to be retrieved. This parameter must be a string data type.

VariableName
: Specifies the name of the user–defined environment variable. This parameter must be a string data type.

Return Values

Returns the value of the user–defined environment variable in string format.

Implementation Specifics

This intrinsic function is part of Simple Network Management Protocol (SNMP) Manager in AIX Network Management/6000.

Related Information

The **getenv** library command, **load_env** library command, **setenv** library command.

The **setenv** intrinsic function.

Alphabetic List of Intrinsic Functions, Functional List of Intrinsic Functions, How to Create **xgmon** Intrinsic Functions in *Communications Programming Concepts*.

Using Intrinsic Functions in *Communications Programming Concepts*.

group_dep Function

Purpose

Maps a dynamically created node or host to the topology display window.

Syntax

(int) group_dep (*Node*, *ElementName*)
string *Node*;
string *ElementName*;

Description

The **group_dep** function maps a dynamically created node or host to the topology display. The normal sequence of events is for a display element to be created by the **make_dep** function or by the **xgmon** program in the *learn* mode. Then, the **move_dep** function positions the element and finally, the **group_dep** function maps the element to the topology display.

Parameters

Node Specifies the name of the node under which the display element specified by the *ElementName* parameter is to be mapped to the display. This parameter must be a string data type. The pseudo–root node is indicated by passing the null string as the value of the *Node* parameter.

ElementName Specifies the name of the display element to be mapped to the topology display. This parameter must be a string data type.

Return Values

Returns an integer value as follows:

0 (zero) The **group_dep** function successfully groups the display element specified by the *ElementName* parameter under the node specified by the *Node* parameter.

1 The *ElementName* parameter does not specify a display element when grouping under the pseudo–root.

2 The node defined by the *Node* parameter is not a display element.

3 The node defined by the *Node* parameter is invalid because it has an IP address; that is, the parameter specified is a host, not a node.

4 The *ElementName* parameter does not specify a display element when grouping under a node other than the pseudo–root.

5 The display element specified by the *ElementName* parameter is already grouped under a node.

Implementation Specifics

This intrinsic function is part of Simple Network Management Protocol (SNMP) Manager in AIX Network Management/6000.

Related Information

The **learn** subcommand, the **make_dep**subcommand, **move_dep** subcommand.

Alphabetic List of Intrinsic Functions, Functional List of Intrinsic Functions, How to Create **xgmon** Intrinsic Functions, How to Create **xgmon** Library Commands in *Communications Programming Concepts*.

Using Intrinsic Functions in *Communications Programming Concepts*.

gw_var Function

Purpose

Extracts the value of the specified Management Information Base (MIB) numeric–format instance ID for the specified host from the internal database.

Syntax

(DataType) **gw_var** (*HostName*, *ObjectID*)
string *HostName*;
string *ObjectID*;

Description

The **gw_var** intrinsic function extracts the value of the specified *HostName, ObjectID* pair from the internal database.

Notes:

1. If the MIB variable's time-to-live (TTL) has expired, there will be no value for the specified *HostName, ObjectID* pair in the internal database. The TTL value is specified in the /**etc**/**mib_desc** file.
2. See RFC 1066 for further information.

Data Type

The data type can be an integer, a string, or a pointer.

Parameters

HostName	Specifies the name or IP address (in dot notation) of a host. The value of this parameter must be a string data type.
ObjectID	Specifies the MIB numeric–format instance ID. The value of this parameter must be a string data type.

Return Values

The return value is the value of the MIB variable. Since the **gw_var** function can return variables of any type (such as integers, strings, or pointers), it is up to the programmer to know which of these formats is used for the stored data. To find out which type to expect, use the **base_type** function.

Implementation Specifics

This intrinsic function is part of Simple Network Management Protocol (SNMP) Manager in AIX Network Management/6000.

File

/**etc**/**mib**/**_desc**	Defines the Management Information Base (MIB) variables that the **xgmon** program should recognize and handle. The user also specifies a time-to-live (TTL) value (in seconds) for each variable.

Related Information

The **base_type** function.

Alphabetic List of Intrinsic Functions, Functional List of Intrinsic Functions, How to Create **xgmon** Intrinsic Functions in *Communications Programming Concepts*.

Understanding the Simple Network Management Protocol (SNMP), Understanding the Management Information Base (MIB), Understanding Terminology Related to Management Information Base (MIB) Variables, Working with Management Information Base (MIB) Variables in *Communications Programming Concepts*.

Using Intrinsic Functions in *Communications Programming Concepts*.

hexval Function

Purpose

Returns the integer value represented by the text characters in the specified string.

Syntax

(int) hexval (*HexString*)
string *HexString*;

Description

The **hexval** function returns the integer value represented by the text characters in the *HexString* parameter. It assumes the number is to be interpreted as a hexadecimal number and accepts both uppercase and lowercase representations of hex digits.

Note: If a character is specified in the string that is not a valid hex digit, it is ignored.

Parameter

HexString	Specifies the hex string to be queried. This parameter must be a string data type.

Return Value

This **hexval** function returns the integer value represented by the text characters in the specified string.

Implementation Specifics

This intrinsic function is part of Simple Network Management Protocol (SNMP) Manager in AIX Network Management/6000.

Related Information

The **num** function, **val** function.

Alphabetic List of Intrinsic Functions, Functional List of Intrinsic Functions, How to Create **xgmon** Intrinsic Functions in *Communications Programming Concepts.*

Using Intrinsic Functions in *Communications Programming Concepts.*

highlight_dep Function

Purpose

Permits a virtual G machine to temporarily highlight a display element.

Syntax

(int) highlight_dep (*ElementName*)
(int) highlight_dep (*ElementName*, *TimeOut*)
(int) highlight_dep (*ElementName*, *TimeOut*, *State*)
string *ElementName*;
int *TimeOut*;
string *State*;

Description

The **highlight_dep** function permits a virtual G machine to change the color of a display element temporarily. If the timeout value is not specified, the display element is changed to white for 30 seconds. If the state is not specified, the display element is changed to white for the specified timeout period; otherwise, the display element is changed to the color specified by the *State* parameter.

The virtual G machine uses different colors to indicate a display element's state for the timeout period. When the time elapses and the window is redrawn, the element returns to its normal color.

Note: To redraw the screen, call the **set_element_mask** function.

Parameters

ElementName Name of the display element to be highlighted. This parameter must be a string.

TimeOut Specifies the period of time for the display element to be highlighted. This parameter must be an integer.

State Specifies, through the use of color, the state of a display element. This parameter must be a string. Choose from the following states:

State	Color	Black-and-White	Black-and-White
	hosts, node, links	hosts and nodes	links only
up	green	white background	solid line
down	red	black background	dotted line, large spaces
unknown	yellow	shaded, white letters	dotted line, finely spaced
highlight	white	white background	solid line
acknowledge	cyan (blue green)	shaded, black letters	thin, dashed line
ignore	violet	shaded, black letters	thin, dashed line
inactive	blue	shaded, black letters	thin, dashed line

Return Values

The function returns 0 if the element was defined, otherwise it returns –1.

Implementation Specifics

This intrinsic function is part of Simple Network Management Protocol (SNMP) Manager in AIX Network Management/6000.

Related Information

The **set_element_mask** function.

Alphabetic List of Intrinsic Functions, Functional List of Intrinsic Functions, How to Create **xgmon** Intrinsic Functions, How to Create **xgmon** Library Commands in *Communications Programming Concepts.*

Using Intrinsic Functions in *Communications Programming Concepts.*

hostname Function

Purpose

Returns the text name of the host.

Syntax

(string) hostname (*IPAddress*)
int *IPAddress*;

Description

The **hostname** function returns the text name of the host.

Parameter

IPAddress	Specifies the IP address of the specified host. This parameter must be an integer.

Return Values

The **hostname** function returns the name of the host. If the text name of the host cannot be determined, the IP address in dot notation is returned.

Implementation Specifics

This intrinsic function is part of Simple Network Management Protocol (SNMP) Manager in AIX Network Management/6000.

Related Information

The **dotaddr** function, **ipaddr** function.

Alphabetic List of Intrinsic Functions, Functional List of Intrinsic Functions, How to Create **xgmon** Intrinsic Functions in *Communications Programming Concepts*.

Using Intrinsic Functions in *Communications Programming Concepts*.

ipaddr Function

Purpose

Returns the normal, primary IP address of the specified host.

Syntax

(int) ipaddr (*HostName*)
string *HostName*;

Description

The **ipaddr** function returns the normal, primary IP address of the host queried.

Parameter

HostName Specifies the name of the host to be queried. The *HostName* parameter may be specified as either the text name or the IP address in dotted decimal or dot notation (for example, `129.35.1.1`). This parameter must be a string data type.

Return Value

Returns the binary 4–byte value of the IP address of the *HostName* parameter.

Example

1. The following is an example of the **ipaddr** function with a dotted decimal parameter:

```
int rc;
rc = (int) ipaddr ("128.83.1.35");
print "rc is: %d\n",rc;
```

Implementation Specifics

This intrinsic function is part of Simple Network Management Protocol (SNMP) Manager in AIX Network Management/6000.

Related Information

The **dotaddr** function, **hostname** function.

Alphabetic List of Intrinsic Functions, Functional List of Intrinsic Functions, How to Create **xgmon** Intrinsic Functions in *Communications Programming Concepts.*

Using Intrinsic Functions in *Communications Programming Concepts.*

left Function

Purpose

Extracts a substring beginning at the leftmost portion of the source string.

Syntax

(string) left (*Source*, *Length*)
string *Source*;
int *Length*;

Description

The **left** function returns a substring of a specified length that is extracted beginning at the leftmost portion of the source string.

Parameters

Source — Specifies which string to use as source. This parameter must be a string data type.

Length — Specifies a number of characters to extract. This parameter must be an integer data type.

Note: The index of the first character in the source string is always 1 (one). All strings in the **xgmon** programming utility are indexed this way.

Return Value

The **left** function returns a substring extracted from the left side of the source string.

Implementation Specifics

This intrinsic function is part of Simple Network Management Protocol (SNMP) Manager in AIX Network Management/6000.

Related Information

The **mid** function, **right** function, **strlen** function, **substr** function.

Alphabetic List of Intrinsic Functions, Functional List of Intrinsic Functions, How to Create **xgmon** Intrinsic Functions in *Communications Programming Concepts.*

Using Intrinsic Functions in *Communications Programming Concepts.*

lookup_addr Subroutine

Purpose

Returns the text name of a host.

Syntax

char *lookup_addr (*address***)**
**unsigned long ***address*;

Description

The **lookup_addr** subroutine returns the text name of the host specified by the *address* parameter.

Parameter

address	A pointer to the Internet address of the host.

Return Value

Returns the text name of the host specified by the *address* parameter.

Implementation Specifics

This subroutine is part of SNMP Application Programming Interface in AIX Network Management/6000.

Related Information

The **SNMP_errormsg** array.

The **lookup_host** subroutine.

Using the SNMP API Subroutine Library, Understanding the Simple Network Management Protocol (SNMP) in *Communications Programming Concepts.*

lookup_host Subroutine

Purpose

Returns the Internet address of a host.

Syntax

unsigned long lookup_host (*hostname*)
char **hostname*;

Description

The **lookup_host** subroutine returns the Internet address associated with the host denoted by the *hostname* parameter.

Parameter

hostname Specifies the host for which the address is requested. The *hostname* parameter can be specified by using either the text name of the host (for example, `localhost`) or the name in dot notation (for example, `127.0.0.1`). If the *hostname* parameter is not specified in dot notation, the **gethostbyname** library routine is used to look up the host's address.

Return Values

If the host is unknown, the **lookup_host** subroutine returns a 0 (zero). Otherwise, the return value is the Internet address of the named host.

Implementation Specifics

This subroutine is part of SNMP Application Programming Interface in AIX Network Management/6000.

Related Information

The **SNMP_errormsg** array.

The **lookup_addr** subroutine.

Using the SNMP API Subroutine Library, Understanding the Simple Network Management Protocol (SNMP) in *Communications Programming Concepts.*

lookup_SNMP_group Subroutine

Purpose

Finds the set of all numeric-format variable names that contain a given text string as prefix.

Syntax

char **lookup_SNMP_group (*prefix*)
char **prefix*;

Description

The **lookup_SNMP_group** subroutine searches all text–format names in the **/etc/mib_desc** file and extracts those that are prefixed by the string given in the *text* parameter. The search is not case–sensitive. An array of pointers to the numeric–format names is returned. Each numeric–format name is terminated by a . (dot) so that an instance can be directly concatenated to it.

Parameter

prefix	A pointer to a text string assumed to be the prefix of a group of MIB variable names in text format.

Return Values

If matching names are found, a pointer to an array of pointers to the matching names is returned. The array is terminated by a pointer to an empty string. If no matching names are found, the array contains only the empty string pointer.

Example

1. The following entry returns a pointer to an array of four pointers:

   ```
   lookup_SNMP_group ("sys");
   ```

   The first three pointers refer to the following character strings:

   ```
   "1.3.6.1.2.1.1.1."
   "1.3.6.1.2.1.1.2."
   "1.3.6.1.2.1.1.3."
   ```

   which are, respectively, `"sysDescr"`, `"sysObjectId"`, and `"sysUpTime"`.

   The fourth pointer ("") refers to an empty string.

   Note: A prefix value of `"1.3.6"` is invalid since the *prefix* parameter must *not* be numeric.

Implementation Specifics

This subroutine is part of SNMP Application Programming Interface in AIX Network Management/6000.

File

/etc/mib_desc Defines the Management Information Base (MIB) variables.

Related Information

The **SNMP_errormsg** array.

The **extract_SNMP_name** subroutine, **lookup_SNMP_name** subroutine.

Using the SNMP API Subroutine Library, Understanding the Simple Network Management Protocol (SNMP), Understanding the Management Information Base (MIB), Understanding Terminology Related to Management Information Base (MIB) Variables, Working with Management Information Base (MIB) Variables in *Communications Programming Concepts*.

lookup_SNMP_name Subroutine

Purpose

Returns the numeric–format name of a Management Information Base (MIB) variable.

Syntax

char *lookup_SNMP_name (*text_name***)**
char ***text_name*;

Description

The **lookup_SNMP_name** subroutine maps the text name of the MIB variable specified by the *text_name* parameter to the corresponding numeric-format name. This search is not case–sensitive.

Parameter

text_name Specifies the text name of the MIB variable.

Return Values

A pointer to the numeric–format name of the MIB variable specified by the *text_name* parameter is returned. If the text name is not recognized, a pointer to the null string is returned. Note that the returned name is terminated with a . (dot) so that an instance value can be directly concatenated to it.

Example

1. If the *text_name* parameter is `"sysDescr"`, a pointer to the string `"1.3.6.1.2.1.1.1."` is returned.

   Note: A text name value of `"1.3.6.1.2.1.1.1"` is invalid since the *text_name* parameter must *not* be numeric.

Implementation Specifics

This subroutine is part of SNMP Application Programming Interface in AIX Network Management/6000.

File

/etc/mib_desc Defines the Management Information Base (MIB) variables.

Related Information

The **SNMP_errormsg** array.

The **extract_SNMP_name** subroutine, **get_MIB_base_type** subroutine, **get_MIB_name** subroutine, **get_MIB_variable_type** subroutine, **lookup_SNMP_group** subroutine.

Using the SNMP API Subroutine Library, Understanding the Simple Network Management Protocol (SNMP), Understanding the Management Information Base (MIB), Understanding Terminology Related to Management Information Base (MIB) Variables, Working with Management Information Base (MIB) Variables in *Communications Programming Concepts*.

make_dep Function

Purpose

Dynamically creates a new node or host.

Syntax

(int) make_dep (*ElementName*)
string *ElementName*;

Description

The **make_dep** function dynamically creates a new node or host in the topology display window. Before a node or host created by the **make_dep** function can appear in the topology display window, it must be mapped to the display by calling the **group_dep** function.

If the new display element cannot be mapped to an IP address, the display element is treated as a node. If the display element does map to an IP address, the display element is treated as a host.

Note: The display element names are treated case–insensitive; that is, a node or host named `austin` is the same node or host as the one named `Austin`.

Parameter

ElementName	Indicates the name to be assigned to the new display element. This parameter must be a string data type.

Return Values

Returns a 0 (zero) if successful. Otherwise, –1 is returned.

Implementation Specifics

This intrinsic function is part of Simple Network Management Protocol (SNMP) Manager in AIX Network Management/6000.

Related Information

The **group_dep** function, **make_link** function, **move_dep** function.

Alphabetic List of Intrinsic Functions, Functional List of Intrinsic Functions, How to Create **xgmon** Intrinsic Functions, How to Create **xgmon** Library Commands in *Communications Programming Concepts*.

Using Intrinsic Functions in *Communications Programming Concepts*.

make_link Function

Purpose

Dynamically creates a link between two hosts.

Syntax

(int) make_link (*FromAddr*, *ToAddr*)
string *FromAddr*;
string *ToAddr*;

Description

The **make_link** function dynamically creates a link display element.

Note: A node does not have an IP address and can therefore not be linked.

Parameters

FromAddr — Specifies the host name or IP address from which the link extends. This parameter must be a string data type.

ToAddr — Specifies the host name or the IP address to which the link extends. This parameter must be a string data type.

Return Values

Returns an integer value as follows:

0 (zero) — The **make_link** function successfully created a link between two hosts.

1 — The host defined by the *FromAddr* parameter has an invalid IP address.

2 — The *Fromaddr* parameter defines a node in the topology description file and has no interface.

3 — The host defined by the *ToAddr* parameter has an invalid IP address.

4 — The *ToAddr* parameter defines a node in the topology description file and has no interface.

Implementation Specifics

This intrinsic function is part of Simple Network Management Protocol (SNMP) Manager in AIX Network Management/6000.

Related Information

The **make_dep** function.

Alphabetic List of Intrinsic Functions, Functional List of Intrinsic Functions, How to Create **xgmon** Intrinsic Functions, How to Create **xgmon** Library Commands in *Communications Programming Concepts*.

make_SNMP_request Subroutine

Purpose

Encodes an SNMP request.

Syntax

int make_SNMP_request (*req_type*, *community*, *num_vars*, *req_name*, *set_value*, *out_packet*, *max_outlen*)
int *req_type*;
char **community*;
int *num_vars*;
char **req_name*[ ];
unsigned long *set_value*[ ];
char **out_packet*;
int *max_outlen*;

Description

The **make_SNMP_packet** subroutine encodes a get, get-next, or set request.

Parameters

community Specifies a string that is the community name to be encoded in the packet.

max_outlen Specifies the maximum length of the output buffer into which the encoded packet is placed.

num_vars Specifies the number of variables to be requested or set.

out_packet Points to a buffer in which the encoded packet is placed.

req_name Specifies an array of pointers to the instance IDs on which an operation is performed. Each entry in the *req_name* array points to a string that represents a MIB instance ID in numeric format.

req_type Specifies the request type, which can be one of the following:

1 Indicates a get request.

2 Indicates a get-next request.

3 Indicates a set request.

set_value Specifies an array of pointers or unsigned integers that correspond one-to-one with the instance IDs in the *req_name* array. Each entry is either the value of the corresponding instance ID if its base type is integer, or a pointer to the value if the base type is string. The *set_value* parameter is used only with set requests.

Return Values

If a fatal error occurs, –1 is returned. If the return value is non–negative, it represents the length of the generated packet.

Implementation Specifics

This subroutine is part of SNMP Application Programming Interface in AIX Network Management/6000.

Related Information

The **SNMP_errormsg** array.

The **parse_SNMP_packet** subroutine, **send_recv_SNMP_packet** subroutine.

Using the SNMP API Subroutine Library, Understanding the Simple Network Management Protocol (SNMP), Understanding the Management Information Base (MIB), Understanding Terminology Related to Management Information Base (MIB) Variables, Working with Management Information Base (MIB) Variables in *Communications Programming Concepts.*

mid Function

Purpose

Extracts a substring from within the source string.

Syntax

(string) mid (*Source*, *Start*, *Length*)
string *Source*;
int *Start*;
int *Length*;

Description

The **mid** function returns a substring of a specified length that is extracted from the source string beginning at the start position.

Parameters

Source Specifies which string to use as source. This parameter must be a string data type.

Start Specifies the position of the first character extracted from the specified source string. This parameter must be an integer data type.

Length Specifies a number of characters to extract. This parameter must be an integer data type.

Note: The index of the first character in the source string is always 1 (one). All strings in the **xgmon** programming utility are indexed this way. For example, to specify the first character in the source string, set the *Start* parameter to 1.

Return Value

The **mid** function returns characters from the middle of the source string.

Implementation Specifics

This intrinsic function is part of Simple Network Management Protocol (SNMP) Manager in AIX Network Management/6000.

Related Information

The **ascii** function, **left** function, **right** function, **strlen** function, **substr** function.

Alphabetic List of Intrinsic Functions, Functional List of Intrinsic Functions, How to Create **xgmon** Intrinsic Functions in *Communications Programming Concepts*.

Using Intrinsic Functions in *Communications Programming Concepts*.

move_dep Function

Purpose

Changes the relative location of a display element within a topology display window.

Syntax

(int) move_dep (*ElementName*, *IntX*, *IntY*)
string *ElementName*;
int *IntX*;
int *IntY*;

Description

The **move_dep** function changes the relative location of a display element within a topology display window. The x and y coordinates are relative to the 100 x 100 reference grid used by the topology description utility.

Note: Links cannot be moved. They are rooted to the hosts they connect.

Parameters

ElementName Specifies the name of the element to be moved. This parameter must be a string data type.

IntX Specifies the position of the x coordinate. This parameter must be an integer data type.

IntY Specifies the position of the y coordinate. This parameter must be an integer data type.

Return Values

If successful, the **move_dep** function returns a 0 (zero). Otherwise, –1 is returned.

Implementation Specifics

This intrinsic function is part of Simple Network Management Protocol (SNMP) Manager in AIX Network Management/6000.

Related Information

Alphabetic List of Intrinsic Functions, Functional List of Intrinsic Functions, How to Create **xgmon** Intrinsic Functions, How to Create **xgmon** Library Commands in *Communications Programming Concepts*.

Using Intrinsic Functions in *Communications Programming Concepts*.

new_deps Function

Purpose

Returns a pointer to an array of strings representing the names of dynamically created display elements.

Syntax

(pointer) new_deps(0)

Description

The **new_deps** function returns a pointer to an array of strings representing the names of dynamically created display elements. The end of the list is marked by the null string.

The list represents all of the display elements created by the **xgmon** program since the last call to the **new_deps** function. Note that the display elements created by the **make_dep** and **make_link** functions also appear in this list if they were created after the first display element was created by the **xgmon** program.

Parameter

Dummy parameter 0 (zero) is required.

Return Value

Returns a pointer to an array of strings representing the names of dynamically created display elements.

Implementation Specifics

This intrinsic function is part of Simple Network Management Protocol (SNMP) Manager in AIX Network Management/6000.

Related Information

The **make_dep** function, **make_link** function.

Alphabetic List of Intrinsic Functions, Functional List of Intrinsic Functions, How to Create **xgmon** Intrinsic Functions, How to Create **xgmon** Library Commands in *Communications Programming Concepts.*

Using Intrinsic Functions in *Communications Programming Concepts.*

next_alternate Function

Purpose

Changes the current primary address of the specified host to be the next available alternate address.

Syntax

(int) next_alternate (*Hostname*)
string *HostName*;

Description

The **next_alternate** function changes the current primary address of the designated host to be the next available alternate address. Alternate addresses are selected in round–robin order.

Note: Alternate addresses are specified in the topology description file. The designated host must have alternate addresses specified in the current topology description file; otherwise this command has no effect.

Parameter

HostName	Specifies the name of the host to be queried. This parameter must be a string data type.

Return Values

The **next_alternate** function returns 0 if the attempt fails, or 1 if it is successful.

Implementation Specifics

This intrinsic function is part of Simple Network Management Protocol (SNMP) Manager in AIX Network Management/6000.

Related Information

The **get_primary** function.

Alphabetic List of Intrinsic Functions, Functional List of Intrinsic Functions, How to Create **xgmon** Intrinsic Functions in *Communications Programming Concepts.*

Using Intrinsic Functions in *Communications Programming Concepts.*

num Function

Purpose

Returns a string of text characters representing the decimal value of the specified integer.

Syntax

(string) num (*Number***)**
int *Number*;

Description

The **num** function returns the string of text characters that represent the decimal value of the *Number* parameter. The **num** function is the inverse of the **val** function. For various ways to format the string, refer to the **sprintf** function.

Parameter

Number The decimal value to be converted into a string of text characters. This parameter must be an integer.

Return Value

The **num** function returns the string of text characters that represent the decimal value of the *Number* parameter.

Implementation Specifics

This intrinsic function is part of Simple Network Management Protocol (SNMP) Manager in AIX Network Management/6000.

Related Information

The **val** function, **sprintf** function.

Alphabetic List of Intrinsic Functions, Functional List of Intrinsic Functions, How to Create **xgmon** Intrinsic Functions in *Communications Programming Concepts.*

Using Intrinsic Functions in *Communications Programming Concepts.*

parse_SNMP_packet Subroutine

Purpose

Decodes an SNMP packet.

Syntax

int parse_SNMP_packet *(packet, packet_len, from_host)*
char **packet*;
int *packet_len*;
unsigned long *from_host*;

Description

The **parse_SNMP_packet** subroutine is called by the **send_recv_SNMP_packet** subroutine when an SNMP get–response packet is received. It may be called directly if an application receives packets directly. It extracts variable bindings from the packet and calls the **save_SNMP_var** or **save_SNMP_trap** subroutines as appropriate to process each binding in the packet.

Parameters

from_host Specifies the Internet address of the host sending the trap.

packet Points to the contents of the packet.

packet_len Specifies the packet length.

Return Value

If a fatal error occurs, a –1 is returned. If the return value is not non–negative, it is the error status from the SNMP packet. A return value of 0 (zero) indicates no error.

Implementation Specifics

This subroutine is part of SNMP Application Programming Interface in AIX Network Management/6000.

Related Information

The **SNMP_errormsg** array.

The **save_SNMP_var** subroutine, **save_SNMP_trap** subroutine, **send_recv_SNMP_packet** subroutine.

Using the SNMP API Subroutine Library, Understanding the Simple Network Management Protocol (SNMP) in *Communications Programming Concepts.*

password Function

Purpose

Returns the SNMP community name associated with the specified host.

Syntax

(string) password (*HostName*)
string *HostName*;

Description

The **password** function returns the SNMP community name associated with the specified host. This information is obtained from the current topology description. If there is no entry for the host, then the null string is returned.

Parameter

HostName	Specifies the name of the host to be queried. This parameter must be a string data type.

Return Values

The **password** function returns the SNMP community name associated with the specified host in string format. If there is no entry for the host, the null string is returned.

Implementation Specifics

This intrinsic function is part of Simple Network Management Protocol (SNMP) Manager in AIX Network Management/6000.

Related Information

Alphabetic List of Intrinsic Functions, Functional List of Intrinsic Functions, How to Create **xgmon** Intrinsic Functions in *Communications Programming Concepts.*

Using Intrinsic Functions in *Communications Programming Concepts.*

ping Function

Purpose

Sends an Internet Control Message Protocol (ICMP) ECHO request to the specified host.

Syntax

(int) ping (*HostName*)
string *HostName*;

Description

The **ping** function sends an ICMP ECHO request to the named host. The *HostName* parameter can be either the host name or an IP address in dot notation.

Parameter

HostName Specifies the text name or IP address (in dot notation) of the host to be queried. This parameter must be a string data type.

Return Values

If a reply is not received, the return value is –1; otherwise, the return value is the number of milliseconds elapsed between the sending of the request and the arrival of the response.

Implementation Specifics

This intrinsic function is part of Simple Network Management Protocol (SNMP) Manager in AIX Network Management/6000.

Related Information

The **ping** subcommand, **ping_all** subcommand.

Alphabetic List of Intrinsic Functions, Functional List of Intrinsic Functions, How to Create **xgmon** Intrinsic Functions in *Communications Programming Concepts*.

Using Intrinsic Functions in *Communications Programming Concepts*.

raise_window Function

Purpose

Raises the graphics window associated with the virtual G machine in which the program is running.

Syntax

(int) raise_window(0)

Description

The **raise_window** function attempts to raise the graphics window associated with the virtual G machine in which the program is running.

Parameter

Dummy parameter 0 (zero) is required.

Return Values

If no window is associated with the virtual G machine, –1 is returned. If successful, the **raise_window** function returns 0 (zero).

Implementation Specifics

This intrinsic function is part of Simple Network Management Protocol (SNMP) Manager in AIX Network Management/6000.

Related Information

Alphabetic List of Intrinsic Functions, Functional List of Intrinsic Functions, How to Create **xgmon** Intrinsic Functions, How to Create **xgmon** Library Commands in *Communications Programming Concepts.*

Using Intrinsic Functions in *Communications Programming Concepts.*

read Function

Purpose

Reads the next line in an open file specified by the file descriptor.

Syntax

(string) read (*FileDescriptor*)
int *FileDescriptor*;

Description

The **read** function returns the next line in the open file indicated by the *FileDescriptor* parameter. When it reaches end–of–file, this routine returns the null string. The **read** function always adds a trailing space to the actual data.

Parameter

FileDescriptor	File descriptor. This parameter must be an integer.

Return Value

The line of text string read from the file.

Example

1.
```
int fd;
string s;
fd = (int) fopen (filename, "r");
if (fd !=0)
      s=(string)read(fd);
```

Implementation Specifics

This intrinsic function is part of Simple Network Management Protocol (SNMP) Manager in AIX Network Management/6000.

Related Information

The **fopen** function.

Alphabetic List of Intrinsic Functions, Functional List of Intrinsic Functions, How to Create **xgmon** Intrinsic Functions, How to Create **xgmon** Library Commands in *Communications Programming Concepts*.

Using Intrinsic Functions in *Communications Programming Concepts.*

real_type Function

Purpose

Takes a Management Information Base (MIB) numeric–format variable name or numeric–format instance ID and returns a number indicating its actual MIB type.

Syntax

(int) real_type (*ObjectID*)
string *ObjectID*;

Description

The **real_type** function takes an MIB numeric–format variable name or numeric–format instance ID and returns a number indicating its actual MIB type.

Parameter

ObjectID Specifies the numeric–format variable name or numeric–format instance ID of the MIB object whose MIB type is queried. This parameter must be a string data type.

Return Values

Returns an integer designating the MIB type as follows:

1 = number

2 = string

3 = object ID

4 = empty

5 = IP address

6 = counter

7 = gauge

8 = time ticks.

If the MIB type cannot be determined, –1 is returned.

Implementation Specifics

This intrinsic function is part of Simple Network Management Protocol (SNMP) Manager in AIX Network Management/6000.

Related Information

Alphabetic List of Intrinsic Functions, Functional List of Intrinsic Functions, How to Create **xgmon** Intrinsic Functions in *Communications Programming Concepts*.

Understanding the Simple Network Management Protocol (SNMP), Understanding the Management Information Base (MIB), Understanding Terminology Related to Management Information Base (MIB) Variables, Working with Management Information Base (MIB) Variables in *Communications Programming Concepts*.

Using Intrinsic Functions in *Communications Programming Concepts*.

rename_dep Function

Purpose

Renames a display element.

Syntax

(int) rename_dep (*ElementName*, *NewName***)**
string *ElementName*;
string *NewName*;

Description

The **rename_dep** function renames a display element. The new name must map to the same IP address as the original. This means that the new name and IP address must be in the /**etc**/**hosts** file or the auxiliary host file. Remember to execute the **clearcache** system command when these files are altered.

Parameters

ElementName	Specifies the name of the display element to be renamed. This parameter must be a string data type.
NewName	Specifies the new name of the display element. This parameter must be a string data type.

Return Values

If successful, the **rename_dep** function returns 0 (zero). Otherwise, it returns –1.

Implementation Specifics

This intrinsic function is part of Simple Network Management Protocol (SNMP) Manager in AIX Network Management/6000.

Related Information

The **clearcache** system command, **hostdata** system command.

Alphabetic List of Intrinsic Functions, Functional List of Intrinsic Functions, How to Create **xgmon** Intrinsic Functions, How to Create **xgmon** Library Commands in *Communications Programming Concepts.*

Using Intrinsic Functions in *Communications Programming Concepts.*

reuse_mem Function

Purpose

Controls garbage collection by a virtual G machine (VGM).

Syntax

(int) reuse_mem(*EnableFlag***)**
int *EnableFlag***;**

Description

The **reuse_mem** function is used by a VGM to control garbage collection. By default, garbage collection is not enabled. If the **reuse_mem** function is called with a nonzero argument, an attempt to enable garbage collection is made. This may not be successful because the operator has the ability to disable garbage collection by using the **reuse** system command.

Parameter

EnableFlag This parameter is set to a nonzero value to enable garbage collection.

Return Values

Returns a 1 if garbage collection is enabled. Otherwise, a 0 (zero) is returned.

Implementation Specifics

This intrinsic function is part of Simple Network Management Protocol (SNMP) Manager in AIX Network Management/6000.

Related Information

The **reuse** system command.

Alphabetic List of Intrinsic Functions, Functional List of Intrinsic Functions, How to Create **xgmon** Intrinsic Functions, How to Create **xgmon** Library Commands in *Communications Programming Concepts.*

Using Intrinsic Functions in *Communications Programming Concepts.*

right Function

Purpose

Extracts a substring from the rightmost portion of the source string.

Syntax

(string) right (*Source*, *Length*)
string *Source*;
int *Length*;

Description

The **right** function returns a substring of a specified length that is extracted from the rightmost portion of the source string.

Parameters

Source Specifies which string to use as source. This parameter must be a string data type.

Length Specifies a number of characters to extract. This parameter must be an integer data type.

Note: The index of the first character in the source string is always 1 (one). All strings in the **xgmon** programming utility are indexed this way.

Return Value

The **right** function returns a substring extracted from the right side of the source string.

Implementation Specifics

This intrinsic function is part of Simple Network Management Protocol (SNMP) Manager in AIX Network Management/6000.

Related Information

The **left** function, **mid** function, **strlen** function, **substr** function.

Alphabetic List of Intrinsic Functions, Functional List of Intrinsic Functions, How to Create **xgmon** Intrinsic Functions in *Communications Programming Concepts*.

Using Intrinsic Functions in *Communications Programming Concepts*.

save_SNMP_trap Subroutine

Purpose

Stores SNMP trap data.

Syntax

void save_SNMP_trap (*enterprise*, *address*, *generic*, *specific*, *time_stamp***)**
char **enterprise*;
unsigned long *address*;
char **generic*;
char **specific*;
unsigned long *time_stamp*;

Description

The **save_SNMP_trap** subroutine is called by the **parse_SNMP_packet** subroutine when a trap packet is parsed. This routine prints the obtained values on the standard output in the following format:

```
"trap received from [enterprise] address: (generic, specific) time
 stamp = time_stamp"
```

Parameters

enterprise — Specifies the value of the sysObjectID MIB variable of the agent generating the trap. This MIB variable is explained in RFC 1066.

address — Specifies the Internet address of the host generating the trap.

generic — Specifies the generic trap type. The string is the text representation of the number associated with the trap type. For example, the number 0 corresponds to a cold-start trap, and the number 2 corresponds to a link-down trap.

specific — Specifies a particular instance of a trap identified by a user. For the link-up and link-down traps, the value specified by the *specific* parameter indicates the interface number associated with the trap. For EGP neighbor-loss trap, *specific* indicates the address of the neighbor in dot notation.

time_stamp — Specifies the time stamp associated with the trap. The time stamp represents the number of 100ths of seconds passed since the agent was initialized at the time the trap was regenerated.

Note: The *generic* and *specific* parameters point to space on the stack; this space is reclaimed when the **save_SNMP_trap** subroutine returns. The *enterprise* parameter points to a static data area which will be overwritten after the **save_SNMP_trap** returns.

Implementation Specifics

This subroutine is part of SNMP Application Programming Interface in AIX Network Management/6000.

Related Information

The **SNMP_errormsg** array.

The **parse_SNMP_packet** subroutine.

Using the SNMP API Subroutine Library, Understanding the Simple Network Management Protocol (SNMP) in *Communications Programming Concepts.*

save_SNMP_var Subroutine

Purpose

Stores retrieved SNMP variable data.

Syntax

void save_SNMP_var (*from_host*, *var_name*, *real_type*, *base_type*, *result*, *len***)**
unsigned long *from_host*;
char **var_name*;
int *real_type*;
int *base_type*;
union_var_val *result*;
int *len*;

Description

The **save_SNMP_var** subroutine is called by the **parse_SNMP_packet** subroutine when a get–response packet is parsed. The default routine prints the obtained values on the standard output in the format of either *var_name = string value* or *var_name = integer value*. The **save_SNMP_var** subroutine does not manipulate the retrieved data.

Parameters

base_type
Specifies the base type of the object. A value of 1 indicates that the object is a string. A value of 2 indicates that the object is an unsigned long integer.

from_host
Specifies the Internet address of the host generating the trap.

len
The size of the integer specified by the base type, or the length of the string specified by the base type.

If the value specified by the *base_type* parameter is a string, the value of the *len* parameter does not include the trailing null byte.

If the value specified by the *base_type* parameter is an integer, the *len* parameter has the value of 0 (zero) in special cases of empty objects.

real_type
Specifies the variable type as defined in RFC 1066.

The values for the *real_type* parameter are:

1 number

2 octet string

3 object identifier

4 empty

5 Internet address

6 counter

7 gauge

8 time ticks.

result Specifies the value of the variable. It has the following format:

```
union_var_val     {
    unsigned long  ul;
    char          *cp;
};
```

var_name Specifies the variable name in numeric format.

Note: If the base type of the object is a string (that is, *base_type* = 1), then the storage pointed to by the *var_name* and *result* parameters is reclaimed by the operating system when the **save_SNMP_var** subroutine returns.

Implementation Specifics

This subroutine is part of SNMP Application Programming Interface in AIX Network Management/6000.

Related Information

The **SNMP_errormsg** array.

The **parse_SNMP_packet** subroutine.

Using the SNMP API Subroutine Library, Understanding the Simple Network Management Protocol (SNMP) in *Communications Programming Concepts*.

send_recv_SNMP_packet Subroutine

Purpose

Sends a query to and awaits a response from an SNMP agent.

Syntax

int send_recv_SNMP_packet (*fd*, *dest*, *out_packet*, *packet_len***)**
int *fd*;
struct sockaddr_in **dest*;
char **out_packet*;
int *packet_len*;

Description

The **send_recv_SNMP_packet** subroutine can be used to send an SNMP request to an SNMP agent, await a response, and process the response packet.

The routine sends the packet to the destination specified by the *dest* parameter.

If a response is obtained from the SNMP agent, then the **parse_SNMP_packet** subroutine will be called with the contents of the received response packet.

Parameters

dest — Specifies the destination address to which the SNMP request is sent. The *dest* parameter can be the *dest_host* external variable set by the **create_SNMP_port** subroutine.

fd — Describes a socket used for the **sendto** and **recvfrom** I/O subroutines. The *fd* parameter can be the address of the *SNMP_port* external variable set by the **create_SNMP_port** subroutine.

out_packet — Contains the SNMP request to be sent.

packet_len — Length of the data specified by the *out_packet* parameter.

External Variables

max_SNMP_retries — Determines the maximum number of times to retry a request. The default value is 3.

SNMP_timeout — Determines the time to wait for a response to be received. The default value is 5 seconds.

The values of these external variables can be reset in the user's main( ) initialization code if necessary.

Return Values

If the agent does not respond, or if an I/O error occurs, –1 is returned; otherwise, the SNMP error status from the response packet is returned. An SNMP error status of 0 indicates no error.

Implementation Specifics

This subroutine is part of SNMP Application Programming Interface in AIX Network Management/6000.

Related Information

The **SNMP_errormsg** array.

The **create_SNMP_port** subroutine, **parse_SNMP_packet** subroutine.

Using the SNMP API Subroutine Library, Understanding the Simple Network Management Protocol (SNMP) in *Communications Programming Concepts.*

set_element_mask Function

Purpose

Allows a virtual G machine to change the current display element mask.

Syntax

(int) set_element_mask (*New Mask*)
int *NewMask*;

Description

The **element_mask** global variable controls how the display elements are drawn. The **set_element_mask** function allows a virtual G machine to change the current mask and returns the value of the *old* mask. The function also controls whether or not the bell sounds and whether the bell sound is double or two–tone.

The mask element starts at the low–order bit 0 (zero) and controls several types of objects. When one of the following bits is specified, the **set_element_mask** function causes the bit's corresponding object to be drawn on the screen:

Bit	Object
0	reserved
1	hosts
2	nodes
3	logical links
5	physical links
9	bell; double alert if bit 10 is not set
10	two-tone alert.

Note: The **set_element_mask** function always causes the visible topology window to be redrawn, even if the mask has not been changed.

Parameter

NewMask Specifies the display element mask to be changed. This parameter must be an integer.

Return Value

The **set_element_mask** function returns the value of the new mask and expresses it as an integer.

Implementation Specifics

This intrinsic function is part of Simple Network Management Protocol (SNMP) Manager in AIX Network Management/6000.

Related Information

Alphabetic List of Intrinsic Functions, Functional List of Intrinsic Functions, How to Create **xgmon** Intrinsic Functions, How to Create **xgmon** Library Commands in *Communications Programming Concepts*.

Using Intrinsic Functions in *Communications Programming Concepts*.

Working with Virtual G Machine (VGM) Variables in *Communications Programming Concepts*.

setenv Function

Purpose

Sets the user–defined environment variable for a host to the specified value.

Syntax

(int) setenv (*DisplayElementName*, *VariableName*, *Value*)
string *DisplayElementName*;
string *VariableName*;
string *Value*;

Description

The **setenv** function sets the user–defined environment variables associated with a display element to the specified value. The **xgmon** program recognizes the **RIGHTCLICK** environment variable name as being associated with the name of the library command that should be run when the display element is double–clicked with the right mouse button.

Parameters

DisplayElementName
Specifies the name or IP address (in dot notation) of the display element for which an environment variable is to be set. This parameter must be a string data type.

Variable Specifies the name of the user–defined environment variable to be set. This parameter must be a string data type.

Value Specifies the value to which the user–defined environment variable will be set. This parameter must be a string data type.

Examples of environment variables and values defined by the user are as follows:

OS	`IBM AIX Version 3.1`
owner	`Gideon Kim`
name	`Token–Ring LAN`

Return Values

The return code is 0 (zero) if the **setenv** function is successful; otherwise, –1 is returned.

Implementation Specifics

This intrinsic function is part of Simple Network Management Protocol (SNMP) Manager in AIX Network Management/6000.

Related Information

The **getenv** subcommand, **setenv** subcommand, **load_env** subcommand.

The **getenv** intrinsic function.

Alphabetic List of Intrinsic Functions, Functional List of Intrinsic Functions, How to Create **xgmon** Intrinsic Functions*Communications Programming Concepts*.

Using Intrinsic Functions*Communications Programming Concepts*.

SNMP_errormsg Array

Purpose

Stores SNMP error messages.

Syntax

char *SNMP_errormsg[ ];

Description

The **SNMP_errormsg** array is an array of pointers to strings containing the appropriate English text corresponding to each SNMP error status value returned by the **send_recv_SNMP_packet** subroutine as follows:

Index	**Contents**
0	No error
1	Too big
2	No such name
3	Bad value
4	Read only.
5	Unsupported or unauthorized operation.

Implementation Specifics

This array is part of SNMP Application Programming Interface in AIX Network Management/6000.

Related Information

The **send_recv_SNMP_packet** subroutine.

Using the SNMP API Subroutine Library, Understanding the Simple Network Management Protocol (SNMP) in *Communications Programming Concepts.*

snmp_var Function

Purpose

Returns the Management Information Base (MIB) numeric–format variable name associated with a specified MIB text–format variable name.

Syntax

(string) snmp_var (*VariableName***)**
string *VariableName*;

Description

The **snmp_var** function returns the MIB numeric–format variable name associated with the *VariableName* parameter. The returned string always has a trailing . (dot). If no such MIB text-format variable name is known, the null string is returned. The MIB text-format variable name and MIB numeric–format variable name mappings are obtained from the **mib_desc** file.

Parameter

VariableName	Specifies the MIB text–format variable name for which the MIB numeric–format variable name is queried. This parameter must be a string data type.

Return Value

Returns the MIB numeric–format variable name associated with the Simple Network Management Protocol (SNMP) variable in string format.

Implementation Specifics

This intrinsic function is part of Simple Network Management Protocol (SNMP) Manager in AIX Network Management/6000.

File

/etc/mib/_desc	Defines the Management Information Base (MIB) variables that the **xgmon** program should recognize and handle. The user also specifies a time-to-live (TTL) value (in seconds) for each variable.

Related Information

Alphabetic List of Intrinsic Functions, Functional List of Intrinsic Functions, How to Create **xgmon** Intrinsic Functions in *Communications Programming Concepts.*

Understanding the Simple Network Management Protocol (SNMP), Understanding the Management Information Base (MIB), Understanding Terminology Related to Management Information Base (MIB) Variables, Working with Management Information Base (MIB) Variables in *Communications Programming Concepts.*

Using Intrinsic Functions in *Communications Programming Concepts.*

sprintf Function

Purpose

Enables formatted arguments.

Syntax

(string) sprintf (*Format*,. *Argument1, Argument2...*)
string *Format*;
DataType Argument;

Description

The **sprintf** function provides various ways to format arguments.

Parameters

Format	A string specifying the format requirements.
Argument1, Argument2...	These parameters can be integers, strings, or pointers.

Return Value

The **sprintf** intrinsic function returns a formatted string.

Implementation Specifics

This intrinsic function is part of Simple Network Management Protocol (SNMP) Manager in AIX Network Management/6000.

Related Information

The **sprintf** function.

Alphabetic List of Intrinsic Functions, Functional List of Intrinsic Functions, How to Create **xgmon** Intrinsic Functions in *Communications Programming Concepts*.

Using Intrinsic Functions in *Communications Programming Concepts*.

strlen Function

Purpose

Returns the length of a string.

Syntax

(int) strlen (*String*)
string *String*;

Description

The **strlen** function returns the length of the string specified by the *String* parameter.

Parameter

String Specifies the string to be queried. This parameter must be a string data type.

Return Value

The **strlen** function returns the length of the string.

Implementation Specifics

This intrinsic function is part of Simple Network Management Protocol (SNMP) Manager in AIX Network Management/6000.

Related Information

The **left** function, **mid** function, **right** function, **substr** function.

Alphabetic List of Intrinsic Functions, Functional List of Intrinsic Functions, How to Create **xgmon** Intrinsic Functions in *Communications Programming Concepts*.

Using Intrinsic Functions in *Communications Programming Concepts*.

substr Function

Purpose

Searches a source string for a particular substring and returns the position of the leftmost occurrence of that substring.

Syntax

(int) substr (*Source*, *Target*)
string *Source*;
string *Target*;

Description

The **substr** function searches the string specified by the *Source* parameter for a string specified by the *Target* parameter. Once the target string is located, the **substr** function returns the position of the leftmost occurrence of the *Target* string.

Note: The index of the first character in the source string is always 1 (one). All strings in the **xgmon** programming utility are indexed this way.

Parameters

Source Specifies the name of the source string to be queried. This parameter must be a string data type.

Target Specifies the name of the target string to be queried. This parameter must be a string data type.

Return Values

If the *Target* string does not appear in the *Source* string, 0 is returned. Otherwise, the position of the leftmost occurrence of the *Target* string is returned.

Implementation Specifics

This intrinsic function is part of Simple Network Management Protocol (SNMP) Manager in AIX Network Management/6000.

Related Information

The **left** function, **mid** function, **right** function, **strlen** function.

Alphabetic List of Intrinsic Functions, Functional List of Intrinsic Functions, How to Create **xgmon** Intrinsic Functions in *Communications Programming Concepts.*

Using Intrinsic Functions in *Communications Programming Concepts.*

time Function

Purpose

Returns the current system time.

Syntax

(int) time(0)

Description

The **time** function returns the current system time and expresses it in seconds.

Parameter

Dummy parameter 0 (zero) is required.

Return Value

The **time** function returns the current system time and expresses it in seconds.

Implementation Specifics

This intrinsic function is part of Simple Network Management Protocol (SNMP) Manager in AIX Network Management/6000.

Related Information

The **ctime** function.

Alphabetic List of Intrinsic Functions, Functional List of Intrinsic Functions, How to Create **xgmon** Intrinsic Functions, How to Create **xgmon** Library Commands in *Communications Programming Concepts.*

Using Intrinsic Functions in *Communications Programming Concepts.*

val Function

Purpose

Returns the integer value represented by the text characters in the specified string.

Syntax

(int) val (*NumberString*)
string *NumberString*;

Description

The **val** function returns the integer value represented by the text characters in the specified string. It assumes the number is to be interpreted as a decimal number.

Note: See the **atoi** subroutine for details.

Parameter

NumberString	Specifies the number string to be queried. This parameter must be a string data type.

Return Value

The **val** function returns the integer value represented by the text characters in the specified string.

Implementation Specifics

This intrinsic function is part of Simple Network Management Protocol (SNMP) Manager in AIX Network Management/6000.

Related Information

The **hexval** function, **num** function.

The **atoi** subroutine.

Alphabetic List of Intrinsic Functions, Functional List of Intrinsic Functions, How to Create **xgmon** Intrinsic Functions in *Communications Programming Concepts.*

Using Intrinsic Functions in *Communications Programming Concepts.*

window_height Function

Purpose

Returns the height, in pixels, of the graphics window associated with a virtual G machine.

Syntax

int window_height(0)

Description

The **window_height** function returns the height, in pixels, of the graphics window associated with the virtual G machine in which the program is running.

Parameter

Dummy parameter 0 (zero) is required.

Return Values

Returns the height, in pixels, of the graphics window associated with the virtual G machine in which the program is running. If there is no window associated with the virtual G machine, –1 is returned.

Implementation Specifics

This intrinsic function is part of Simple Network Management Protocol (SNMP) Manager in AIX Network Management/6000.

Related Information

The **window_width** function.

Alphabetic List of Intrinsic Functions, Functional List of Intrinsic Functions, How to Create **xgmon** Intrinsic Functions, How to Create **xgmon** Library Commands in *Communications Programming Concepts.*

Using Intrinsic Functions in *Communications Programming Concepts.*

window_width Function

Purpose

Returns the width, in pixels, of the graphics window associated with a virtual G machine.

Syntax

(int) window_width(0)

Description

The **window_width** function returns the width, in pixels, of the graphics window associated with the virtual G machine in which the program is running.

Parameter

Dummy parameter 0 (zero) is required.

Return Values

Returns the width, in pixels, of the graphics window associated with the virtual G machine in which the program is running. If there is no window associated with the virtual G machine, –1 is returned.

Implementation Specifics

This intrinsic function is part of Simple Network Management Protocol (SNMP) Manager in AIX Network Management/6000.

Related Information

The **window_height** function.

Alphabetic List of Intrinsic Functions, Functional List of Intrinsic Functions, How to Create **xgmon** Intrinsic Functions, How to Create **xgmon** Library Commands in *Communications Programming Concepts.*

Using Intrinsic Functions in *Communications Programming Concepts.*

words_free Function

Purpose

Returns the number of free words remaining in the data segment of the virtual G machine.

Syntax

(int) words_free(0)

Description

The **words_free** function returns the amount of free space available in the data segment of the virtual G machine. This storage space is measured in units of 32 bits. These units are also referred to as *words*.

Note: If a virtual G machine attempts to use more storage than allocated in its data segment, it is stopped.

Parameter

Dummy parameter 0 (zero). This parameter is required.

Return Value

Returns the number of free storage units (32–bit words) remaining in the data segment of the virtual G machine.

Implementation Specifics

This intrinsic function is part of Simple Network Management Protocol (SNMP) Manager in AIX Network Management/6000.

Related Information

Alphabetic List of Intrinsic Functions, Functional List of Intrinsic Functions, How to Create **xgmon** Intrinsic Functions, How to Create **xgmon** Library Commands in *Communications Programming Concepts.*

Using Intrinsic Functions in *Communications Programming Concepts.*

SNA Services

close Subroutine for SNA Services/6000

Purpose

Closes a file descriptor.

Syntax

#include <luxsna.h>

int close(*fildes***)**
int *fildes***;**

Description

The **close** subroutine closes a connection specified by its file descriptor.

Limited Interface

If the file descriptor was opened using the limited interface, this routine also deallocates the conversation associated with the file descriptor, using the following parameters (see the **ioctl**(DEALLOCATE) subroutine):

- The `type` parameter has a value of FLUSH (DEAL_FLUSH)
- The `deal_flag` parameter is DISCARD.

Extended Interface

If the file descriptor was opened using the extended interface, any active conversations on the connection end abnormally.

Parameter

fildes	Specifies a variable containing the file descriptor of the connection to be closed. This file descriptor is the value returned by the **open** subroutine that opened the connection.

Return Values

When the subroutine completes successfully, it returns a value of 0. If an error occurs, the routine returns a value of –1 and sets the **errno** global variable to indicate the error.

Error Code

The **close** subroutine sets the **errno** global variable to a value to indicate the cause of any errors that occur. The value that this variable can receive is shown below. Error Code Constants in *Communications Programming Concepts* contains a brief description of the error values for AIX SNA Services/6000.

EBADF

File

/usr/include/luxsna.h

Defines constants and structures used by AIX SNA Services/6000 subroutines.

Related Information

The **close** subroutine.

The **open** subroutine for SNA Services/6000, **ioctl** subroutine for SNA Services/6000, **snaclse** subroutine for SNA Services/6000.

close Subroutine for Generic SNA

Purpose

Closes a file descriptor.

Syntax

#include <luxgsna.h>

int close (*fildes, ext*)
int *fildes*;
int *ext*;

Description

The **close** subroutine releases resources that are tied to an AIX SNA Services/6000 attachment specified by its file descriptor.

Parameters

fildes — Specifies a variable containing the file descriptor to be closed. This file descriptor is the value returned by the **open** subroutine.

ext — Ignored by generic SNA.

Return Values

Upon successful completion, the **close** subroutine returns a value of 0. If an error occurs, it returns a value of –1 and sets **errno** to indicate the error.

Error Code

The call sets **errno** to a value which indicates the cause of any errors that occur, as in the following case:

EBADF — An invalid file descriptor was specified.

Related Information

The **open** Subroutine for Generic SNA, **read** Subroutine for Generic SNA, **write** Subroutine for Generic SNA, **ioctl** Subroutine for Generic SNA, **select** Subroutine for Generic SNA.

Developing Special AIX SNA Services/6000 Functions in *Communications Programming Concepts.*

ioctl Subroutine for SNA Services/6000

Purpose

Controls data transfer between local and remote transaction programs.

Syntax

#include <luxsna.h>

int ioctl(*fd*, *request*, *arg*)
int *fd*;
int *request*;
int *arg*;

Description

Note: Do not use this subroutine for programs that use the limited interface.

This subroutine provides control functions for transfer operations between a local and a remote transaction program. The specific control function is specified by the *request* parameter and must be one of the integers (defined in the **luxsna.h** include file) as explained in the following sections:

- ALLOCATE
- ALLOCATE_LISTEN (LU 6.2 only)
- CONFIRM
- CONFIRMED
- CP_STATUS (LU 6.2 only)
- DEALLOCATE
- FLUSH
- GET_ATTRIBUTE (LU 6.2 only)
- GET_PARAMETERS (LU 6.2 only)
- GET_STATUS (LUs 1, 2, and 3 only)
- PREPARE_TO_RECEIVE
- REQUEST_TO_SEND
- SEND_ERROR
- SEND_FMH (LU 1 only)
- SEND_STATUS (LUs 1, 2, and 3 only).

ALLOCATE

The ALLOCATE request allocates a session between the local logical unit or control point (LU/CP) and a remote LU/CP. It then allocates a conversation between the local transaction program and a remote transaction program using the allocated session. The request returns a resource ID to identify the conversation. Use this request before using any other subroutine that refers to the conversation.

If two LUs, connected by a session, try to allocate a conversation on that session at the same time, one of the LUs will be successful and the other will not. Which LU is successful is determined by the BIND negotiation that occurred when the session was established.

The *arg* parameter is a pointer to a structure of type **allo_str** that contains additional information for the request. This structure contains a pointer to an additional structure, **pip_str**. These structures are defined in the **luxsna.h** include file. The resource ID (RID) is returned in the extended allocate structure. Refer to the **allo_str** and **pip_str** structures for information about the fields in these structures.

You must ensure that the remote transaction program name (`tpn`) is in EBCDIC coding before it is stored in the **allo_str** structure. Refer to EBCDIC to ASCII Translation for US English (TEXT)*Communication Concepts and Procedures* for assistance in converting ASCII to EBCDIC.

LU 6.2

When this request completes successfully, the local transaction program (the one that used this request) is in the send state and the remote transaction program is in the receive state.

For two programs to reconnect to each other after they have been disconnected, the following events must occur:

1. One program uses a DEALLOCATE request with the `deal_flag` parameter set to `retain` to deallocate the conversation.
2. The program initiating the reconnection uses an ALLOCATE request with the `type` parameter set to `reconnect`. This action sends a reconnection request to the remote LU.
3. The remote program completes the reconnection when it uses the **read** or **readx** subroutines to receive information.
4. If the application program is performing a remotely attached ALLOCATE request or reconnection, the program must specify the resource ID, `rid`, in the **allo_str** structure.

LUs 1, 2, and 3

When this request completes successfully, the appropriate session (SSCP–LU or LU–LU) is established and both LUs are in HDX contention state. The SSCP–LU session must be active and allocated before allocating the LU–LU session. Trying to allocate the LU–LU session before the SSCP–LU session results in a SNA_STATE error return from the ALLOCATE request.

If an application program tries to allocate an SSCP–LU session and the ACTLU request has not yet been received:

- This subroutine returns a SNA_NSES error.
- When the ACTLU request arrives, the logical unit (LU) starts.

  Note: If a NOTIFY signal is not supported by the host system, the ACTLU request is always accepted with a +RSP (SLU–enabled) ACTLU. If a NOTIFY signal is supported by the host system, the ACTLU request is accepted with a +RSP (SLU–enabled) if an ALLOCATE SSCP request was previously attempted or with a +RSP (SLU–disabled) if an ALLOCATE SSCP request has not yet been attempted.
- A +RSP ACTLU (SLU–enabled) request is sent.

- The **select** subroutine notifies the application program of a change in status.
- The GET_STATUS request indicates that the SSCP is active.
- The application program allocates the SSCP–LU session.

If an application program tries to allocate an SSCP–LU session after the ACTLU request is received:

- If a NOTIFY signal is supported by the Host system and an ACTLU (SLU–disabled) signal was accepted, a NOTIFY (SLU–enabled) signal is sent to the Host system.
- A session is allocated.

If an application program tries to allocate an LU–LU session and the BIND request has not yet been received:

- The **ioctl** subroutine puts the application to sleep waiting for the BIND request to be received and processed.
- An INIT_SELF request is sent to the Host system to request the BIND request.
- When the BIND request arrives, the logical unit (LU) starts.
- A +RSP BIND is sent.
- The LU–LU session is allocated to the application program.

If an application program tries to allocate an LU–LU session after the BIND request is received:

- A session is allocated.

If the host bids for a session, a subsequent ALLOCATE causes sense code 0x0813 (Host bid reject). The bid can be either sent with the Begin Bracket (BB) bit on or a BID command.

ALLOCATE_LISTEN

The ALLOCATE_LISTEN request registers a list or Transaction Program Names (TPNs) for which an application wishes to accept allocate requests (for example, FMH5 Attaches). When an allocate request is received by SNA for one of the registered TPNs, the application will be informed via the **select**(EXCEPTION) subroutine. This routine may be issued multiple time on any connection. Each successive call registers another set of TPNs on that connection. The ALLOCATE_LISTEN request can be used by LU 6.2 only.

CONFIRM

The CONFIRM request asks the remote transaction program to tell whether the last transmission was successfully received. The remote transaction program must respond with one of two **ioctl** subroutine requests: CONFIRMED or SEND_ERROR.

The filedes for this request is the connection ID (`cid`) returned from a previous **open** routine.

For this request, the *arg* parameter is a pointer to a structure of type **confirm_st**. The resource ID (`rid`) that was returned on a previous ALLOCATE must be passed in the **confirm_str** structure. Information about the conversation is returned in the `sense_code` field of the **confirm_str** structure. This structure is defined in the **luxsna.h** include file. Refer to the **confirm_str** structure for information about the fields in this structure.

LU 6.2

The program may use the CONFIRM request for the following special cases:

- Immediately following an **ioctl**(ALLOCATE) request to determine if the allocation of the conversation was successful before sending data
- Following transmission of data to the remote program to get an acknowledgment from the remote program.

LUs 1, 2, and 3

LU 1 uses the CONFIRM request to get an acknowledgment for data that the LU sent to the remote program. However, LUs 2 and 3 do not use this request. The remote program must handle error recovery for the local LU 2 or 3 program.

CONFIRMED

The CONFIRMED request is a response to the CONFIRM request indicating that the transmission was received without detecting any errors. This request can only be used in response to a CONFIRM request.

The filedes for this request is the connection ID (`cid`) returned from a previous **open** routine.

For this request, the *arg* parameter is the resource ID (`rid`) that was returned on a previous ALLOCATE request.

CP_STATUS

CP_STATUS requests information about the capabilities of the control point at the remote node.

The filedes for this request is the connection ID (`cid`) returned from a previous **open** routine.

For this request, the *arg* parameter is a pointer to a structure of type **cp_str**. The resource ID (`rid`) that was returned on a previous ALLOCATE request must be passed in the **cp_str** structure. The remote control point name, the conversation group0 ID, and the session type (contention winner or contention loser) are returned by the **ioctl** subroutine in the **cp_str** structure pointed to by the *arg* parameter. This subroutine also returns a list of remote CP capabilities in the **cp_str** structure as defined in the **luxsna.h** include file. Refer to the **cp_str** structure for a description of the fields in this structure.

DEALLOCATE

The DEALLOCATE request deallocates the specified conversation from the local transaction program and makes the conversation available to be allocated by another transaction program. Information about the specific type of DEALLOCATE request is supplied in the **deal_str** structure pointed to by the *arg* parameter.

The filedes for this request is the connection ID (`cid`) returned from a previous **open** routine.

For this request, the *arg* parameter is a pointer to a structure of type **deal_str**. The resource ID (`rid`) that was returned on a previous ALLOCATE must be passed in the **deal_str** structure. The transaction program should specify the type of deallocate to be preformed (DEFAULT, CONFIRM, ABEND, or FLUSH) in the `type` field of the **deal_str** structure. The transaction program should also specify whether the conversation should be discarded or retained for possible reconnection in the `deal_flag` field of the **deal_str** structure.

This structure is defined in the **luxsna.h** include file. Refer to the **deal_str** structure for information about the fields in this structure.

LU 6.2

The DEALLOCATE request ends the conversation but not the session. The LU resource manager determines whether to keep or end the session. The DEALLOCATE request is only issued by a local transaction program. A remote transaction program receives an indication that a DEALLOCATE request was received from the SNA device driver. The device driver sets the `what_control_rcvd` field in the **ext_io_str** structure to indicate the type of deallocation the device driver received from the remote program.

The remote transaction program must take appropriate action based on the type of DEALLOCATE request received. For example, for deallocate type CONFIRM, the remote transaction program issues the **ioctl**(CONFIRMED) subroutine to complete the deallocation sequence. The remote transaction program is not required to issue a DEALLOCATE request in response to a received DEALLOCATE.

See the **readx** subroutine for an explanation of the **ext_io_str** structure. The SNA device driver performs the local deallocation function when it receives a DEALLOCATE request from the remote transaction program.

LUs 1, 2, and 3

Do not use the DEALLOCATE request with an LU–LU session for either LUs 2 or 3. If used with these sessions, the routine returns with an SNA_STATE error.

To deallocate an LU–LU session that has a corresponding SSCP–LU session, use the DEALLOCATE request to deallocate the SSCP–LU session.

When the local transaction program issues a DEALLOCATE request with `type` set to (DEAL_FLUSH) for the SSCP–LU session:

1. An RSHUTD is sent to the host requesting that the host issue an UNBIND request to terminate the LU–LU session.
2. The local LU rejects all data from the host on the LU–LU session until it receives the UNBIND request.
3. If the host supports a NOTIFY signal, a NOTIFY (SLU_DISABLED) signal is sent to the SSCP.
4. The next allocation of SSCP causes a NOTIFY (SLU_enabled) signal.
5. The local LU rejects all data from the host on the SSCP–LU session until that session is allocated to another application program.

The host can issue an UNBIND request at other times to end the LU–LU session. When the UNBIND occurs, the local program using the LU–LU session receives a return code of SNA_NSES to notify it of the session end.

Unless an RSHUTD is sent, all UNBIND requests are unsolicited. The SNA_NSES signal is returned by SNA_DD (for the **read** subroutine, the **write** subroutine, and so on). The SELECT is completed, if there is one pending. When an unsolicited UNBIND request occurs, the local LU ends the session and uses the **select** subroutine to notify the application program using the session. The **select** subroutine will complete with an exceptional condition. If the application uses a GET_STATUS request of the **ioctl** subroutine, the returned status indicates that the session is not active.

The local LU cannot issue a DACTLU request to end the SSCP–LU session. Therefore, the session remains active until the host ends it with a DACTLU, DACTPU or ACTPU request. In this case, the DEALLOCATE request used on the SSCP–LU session removes the connection to the local program, but does not remove the SSCP–LU session itself.

When used on an LU 1 LU–LU session, the DEALLOCATE request ends a bracket.

FLUSH

The FLUSH request sends any information in the local LU send buffer to the remote LU. The LU normally buffers the data from **write** subroutines until it has enough data to transmit. Using this request the local program forces the local LU to transmit the data in the buffer. The local program can use this request to decrease the delay required to get the data to the remote system. If you use this request when the local transmit buffer is empty, the local LU transmits a null chain element to the remote LU if end chain is specified.

The *arg* parameter for this request is a pointer to a structure of type **flush_str**, which contains the input parameters for the request. For LU 6.2, the `end_chain` field in the **flush_str** structure should always be set to a value of 0. Refer to the **flush_str** structure for a description of the fields in this structure.

GET_ATTRIBUTE

The GET_ATTRIBUTE request gets information about the specified LU 6.2 conversation.

The *arg* parameter for this request is a pointer to a structure of type **attr_str** which contains the input parameter `rid` and receives the output information from the request. Refer to the **attr_str** structure for a description of the fields in this structure.

GET_PARAMETERS

The GET_PARAMETERS request retrieves the data associated with the receipt of an allocate request (for example, FMH5 Attach) for a registered TPN on a particular connection. The data that is returned is for the first allocate request received since the last GET_PARAMETERS call was issued. The GET_PARAMETERS request can be used by LU 6.2 only.

GET_STATUS

The GET_STATUS request gets information about the current link and session, as well as the unprocessed image from the BIND request for the LU–LU session. It can be used by LUs 1, 2, and 3 only.

When an event occurs that changes the status of a link or session, the system informs the application program, using the **select** subroutine. If the application program then uses GET_STATUS, the status returned is for the session that was affected by the change in status. For example, when a BIND request and an SDT request are received, the system uses the **select** subroutine to notify the application program of the change. When the application uses GET_STATUS, the returned status indicates that the LU–LU session is active.

The *arg* parameter for this request is a pointer to a structure of type **gstat_str**, which contains the input parameter `rid` and receives the output status information from the request. Refer to the **gstat_str** structure for a description of the fields in this structure.

Because GET_STATUS reports status changes, **gstat_str** can have a status value of 0.

PREPARE_TO_RECEIVE

The PREPARE_TO_RECEIVE request notifies the remote LU that the local LU needs to change the conversation direction so that the local LU can begin receiving from the remote LU.

The *arg* parameter for this request is a pointer to a structure of type **prep_str** which contains the input parameters for the request. Refer to the **prep_str** structure for a description of the fields in this structure.

Special Cases

Transaction programs can use this request with a type field of FLUSH to complete a **write** subroutine and send a change-of-direction (CD) indication to the remote transaction program. If the send buffer is empty, the request sends a Null FMD with a CD to the remote transaction program.

Transaction programs also use this request with a type field of CONFIRM to complete a **write** subroutine. In this case, the request sends the CD indication with a `Request Definite Response` message.

REQUEST_TO_SEND

The REQUEST_TO_SEND request notifies the remote LU that the local LU needs to change the conversation direction so that the local LU can begin sending to the remote LU. The local program uses a **readx** subroutine to get the send indication from the remote program (in the `what_control_rcvd` field). When the local program receives this indication from the remote program, it enters the send state and the remote program is in the receive state.

The *arg* parameter for this request specifies the resource ID for the conversation that was returned by the ALLOCATE request for the conversation.

SEND_ERROR

The SEND_ERROR request informs the remote transaction program that the local transaction program has detected an error in the information that it received from the remote program.

The *arg* parameter for this request is a pointer to a structure of type **erro_str** which contains the input parameters for the request. Refer to the **erro_str** structure for a description of the fields in this structure.

LU 6.2

When this request is issued in send state, the LU:

1. Flushes the local send buffer.
2. Creates and sends an FMH7.

When this request is issued in receive state, the LU:

1. Generates a negative response.
2. Purges all incoming data.
3. Creates an FMH7.
4. Waits for a send indication to arrive from the remote program.
5. Enters send state to send the error message.

LUs 1, 2, and 3

When this request is issued in send state, the LU:

1. Flushes the local send buffer.
2. Sends a CANCEL request to the remote session.

When this request is issued in receive state, the LU:

1. Generates a negative response, using the `sense_code` specified in the subroutine.
2. Purges all incoming data to the end of chain.

SEND_FMH

The SEND_FMH request sends the FM header to the remote LU. Used only by LU 1 support, this request must be used on a basic conversation.

The *arg* parameter for this request is a pointer to a structure of type **fmh_str** which contains the input parameters for the request. Refer to the **fmh_str** structure for a description of the fields in this structure.

The application program must build the complete FM header to be sent. If more than one FM header is to be sent, the application must build all FM headers with the concatenation bit set within a contiguous area. The application program must also enforce concatenation and chaining rules.

SEND_STATUS

The SEND_STATUS request sends status information about the devices on the local session (LUs 1, 2 and 3, only) to the host program. This request can be used on a basic conversation only. When issued in send state, an LUSTAT is sent to the remote session using the ID to indicate which device the LUSTAT is for. When issued in receive state, the SNA_STATE return code is returned. This request is used for LU–LU sessions only.

The *arg* parameter for this request is a pointer to a structure of type **stat_str**, which contains the input parameters for the status request. Refer to the **stat_str** structure for a description of the fields in this structure.

Parameters

fd — Specifies the variable that contains the file descriptor returned by the **open** subroutine.

request — Specifies the function to be performed as defined in the **luxsna.h** include file.

arg — An integer that can be used to specify either the variable that contains the resource ID (`rid`) returned by the **ioctl**(ALLOCATE) subroutine or a pointer to a structure that contains additional input parameters for the requested function.

Return Values

When the subroutine completes successfully, it returns a value of 0. If an error occurs, the subroutine returns a value of –1 and sets the **errno** global variable to indicate the error.

Error Codes

The subroutine sets the **errno** global variable to a value to indicate the cause of any errors that occur. The values that this variable can receive vary with the requested function. The table in the Error Code Constants in *Communications Programming Concepts* section contains a brief description of the error values for AIX SNA Services/6000.

File

/usr/include/luxsna.h
Defines constants and structures used by AIX SNA Services/6000 subroutines.

Related Information

Node Verification in Defining LU Type 6.2 Connection Characteristics in *Communication Concepts and Procedures*.

The **ioctl** subroutine.

The **open** subroutine for SNA Services/6000, **read** subroutine for SNA Services/6000, **readx** subroutine for SNA Services/6000, **write** subroutine for SNA Services/6000.

ioctl Subroutine for Generic SNA

Purpose

Controls data transfer between local and remote transaction programs.

Syntax

#include <sys/devinfo.h>
#include <luxgsna.h>

int ioctl (*fildes, request, arg*)
int *fildes*;
int *request*;
struct devinfo **arg*;

Description

This subroutine provides control functions for generic SNA applications. The specific control function is specified by the *request* parameter and must be one of the integers (defined in the **luxgsna.h** file) as explained in the following:

HIER_RESET_RSP
: The HIER_RESET_RSP function informs AIX SNA Services/6000 that cleanup has been done after receiving a hierarchical reset from the PU Services of AIX SNA Services/6000. This command is allowed only if the file descriptor was opened for an AIX SNA Services/6000 attachment.

INOP_RSP
: The INOP_RSP function is used to inform AIX SNA Services/6000 that cleanup has been done after receiving an **INOP** command from the PU Services of AIX SNA Services/6000. This command is allowed only if the file descriptor was opened for an AIX SNA Services/6000 attachment.

IOCINFO
: The IOCINFO function returns a devinfo structure that describes the device. After the IOCINFO operation is executed, the device type and flags associated with the file descriptor are returned in the **devinfo** structure pointed to by the *arg* parameter.

Parameters

fildes
: Specifies the file descriptor return by the **open** subroutine.

request
: Specifies the function to be performed as defined in the **luxgsna.h** file.

arg
: A pointer to a structure of type **devinfo** if the request is IOCINFO, (`struct devinfo*`). Otherwise the *arg* parameter is NULL.

Return Values

Upon successful completion, the **ioctl** subroutine returns a value of 0. If an error occurs, it returns a value of –1 and sets **errno** to indicate the error.

Error Codes

The **ioctl** subroutine sets **errno** to a value which indicates the cause of any errors that occur, as shown in the following list:

EBADF An invalid file descriptor was specified.

EFAULT An invalid address was specified.

EINVAL An invalid parameter was passed.

Related Information

The **close** Subroutine for Generic SNA, **open** Subroutine for Generic SNA, **read** Subroutine for Generic SNA, **write** Subroutine for Generic SNA, **select** Subroutine for Generic SNA.

Developing Special AIX SNA Services/6000 Functions in *Communications Programming Concepts*.

lu0api Subroutine

Purpose

Creates 4680 commands for the ADCS Emulator.

Syntax

#include <lu0.h>
#include <adscapi.h>
extern int lu0api();

rc = lu0api (*cmd*, *option*, *data*, *fname*, *parm1*, *parm2*);
short int *cmd*;
short int *option*;
char **data*;
char **fname*;
short int *parm1*;
short int *parm2*;
int *rc*;

Description

The **lu0api** subroutine creates 4680 commands for the ADCS Emulator and places them in the **/usr/lpp/lu0/cmd4680** file, which is read by the **adcs** command when it starts the ADCS emulator. You must compile the **lu0api.c** module with your source code.

Parameters

cmd Specifies the code for the command to be created. Valid values for the *cmd* parameter are:

add

```
cmd  = ADDK;
add/replace option:
  add
   option = 0;
  replace
   option = 1;
fname = (6 byte ascii string, blank padded,
    left justified)
parm1 = (the number of records you wish to add)
parm2 = (the maximum length of a logical record)
rc = lu0api (cmd, option, data, fname, parm1, parm2);
```

create

```
cmd  = CREATEF;
fname = (6 byte ascii string, blank padded,
    left justified)
parm1 = (how many 256 byte blocks in this file)
parm2 = (maximum length of a logical record)
rc = lu0api (cmd, option, data, fname, parm1, parm2);
(fill in the next 3 fields for keyed files only)
(short int) &data[102] = (key length)
(short int) &data[104] = (key offset)
(short int) &data[106] = (randomizing divisor)
```

delete
```
cmd  = DELETEK;
ignore error option:
  don't ignore errors
   option = 0;
  ignore errors
   option = 1;
fname = (6 byte ascii string, blank padded,
    left justified)
parm1 = (the number of records you wish to delete);
parm2 = (maximum length of a logical record);
rc = lu0api (cmd, option, data, fname, parm1, parm2);
```

dump
```
cmd  = DUMPF;
fname = (6 byte ascii string, blank padded,
    left justified)
parm1 = (relative starting sector or token);
parm2 = (number of sectors to dump);
rc = lu0api (cmd, option, data, fname, parm1, parm2);
```

load
```
cmd  = LOADF;
replace option:
  don't replace
    option = 0;
  replace
    option = 1;
fname = (6 byte ascii string, blank padded,
    left justified)
parm1 = (how many records to load)
parm2 = (starting sector number)
rc = lu0api (cmd, option, data, fname, parm1, parm2);
(short int) &data[60] = (how many bytes in last sector);
```

purge
```
cmd  = PURGEF;
fname = (6 byte ascii string, blank padded,
    left justified)
rc = lu0api (cmd, option, data, fname, parm1, parm2);
```

replace
```
cmd  = REPLACEK;
add/replace option:
  add
   option = 0;
  replace
   option = 1;
fname = (6 byte ascii string, blank padded,
    left justified)
parm1 = (the number of records you wish to add);
parm2 = (maximum record length);
rc = lu0api (cmd, option, data, fname, parm1, parm2);
```

option Specifies an option dependent on the value of the *cmd* parameter.

data Specifies the address where the command is stored.

fname Specifies the address of the file that the command affects.

parm1 Specifies a value dependent on the value of the *cmd* parameter.

parm2 Specifies a value dependent on the value of the *cmd* parameter.

rc Specifies the return value indicating the success or failure of the **lu0api** subroutine.

Return Values

If the function is completed successfully a 0 will be returned. If the command code is not valid, then the LU0_CMD value is returned.

Files

/usr/lpp/lu0/lu0.h
Specifies the LU0 header file containing common definitions.

/usr/lpp/lu0/adcsapi.h
Specifies the LU0 header file containing ADCS user application API definitions.

/usr/lpp/lu0/lu0api.h
Specifies the LU0 header file containing LU0 API definitions.

/usr/lpp/lu0/lu0apis.h
Specifies the LU0 header file containing **cmd4680** command file record formats.

/usr/lpp/lu0/lu0conf.h
Specifies the LU0 header file containing configuration file definitions.

/usr/lpp/lu0/lu0api.c
Specifies the C source file that defines the **lu0api** subroutine.

/usr/lpp/lu0/cmd4680
Specifies the file containing 4680 commands, which the ADCS emulator program processes.

Related Information

The **adcs** command.

Applications in *Communications Programming Concepts*.

lu0closep Subroutine

Purpose

Allows the application to end a session with a secondary LU.

Syntax

#include <lu0.h>
extern int lu0closep();

rc = lu0closep (*luid***);**
int *luid*;
int *rc*;

Description

The **lu0closep** subroutine closes a SNA Primary LU0 session. The **lu0closep** subroutine ends a session with the secondary LU.

Parameters

luid Specifies the LU identifier which was returned by a previous **lu0openp** subroutine.

rc Specifies the return value indicating the success or failure of the **lu0closep** subroutine.

Return Values

If the close is successful, a 0 is returned. If there are close errors, –1 is returned and the **errno** global variable is set to one of the error codes specified in the **lu0.h** file.

File

/usr/lpp/lu0/lu0.h Specifies the LU0 header file containing common definitions.

Related Information

The **lu0ctlp** subroutine, **lu0openp** subroutine, **lu0readp** subroutine, **lu0writep** subroutine.

Application Program Interface in *Communications Programming Concepts.*

lu0closes Subroutine

Purpose

Allows the application to end a session with a host application.

Syntax

#include <lu0.h>
extern int lu0closes();

rc = lu0closes (*luid***);**
int *luid*;
int *rc*;

Description

The **lu0closes** subroutine closes the SNA Secondary LU0 session. The **lu0closes** subroutine ends a session with the host application.

Parameters

luid Specifies the LU identifier that was returned by the **lu0opens** subroutine.

rc Specifies the return value indicating the success or failure of the **lu0closes** subroutine.

Return Values

If the close is successful, a 0 is returned. If there are close errors, –1 is returned and the **errno** global variable is to one of the error codes specified in the **lu0.h** file.

File

/usr/lpp/lu0/lu0.h Specifies the LU0 header file containing common definitions.

Related Information

The **lu0ctls** subroutine, **lu0opens** subroutine, **lu0reads** subroutine, **lu0writes** subroutine.

Application Program Interface in *Communications Programming Concepts.*

lu0ctlp Subroutine

Purpose

Allows the application to send SNA commands to the secondary LU.

Syntax

#include <lu0.h>
extern int lu0ctlp();

rc = lu0ctlp(*luid*, *senseptr*, *optcode***)**
int *luid*;
char **senseptr*;
int *optcode*;
int *rc*;

Description

The **lu0ctlp** subroutine sends SNA commands or responses to the secondary LU.

Parameters

luid Specifies the LU identifier which was returned by the **lu0openp** subroutine.

senseptr Points to the address of the sense data buffer (6 bytes).

optcode Specifies the option code. The following are valid values for the *optcode* parameter:

LU0ACPT Sends `accept` to previous request.

LU0REJ Sends `reject` to previous request, parm2 has 4 bytes of sense data.

LU0RSTAT Receives sense bytes(6) from **LUSTAT** or response information into parm2 buffer.

LU0FSM Returns 4 FSM bytes into parm2 buffer.

LU0SDT Sends `sdt`.

LU0CLEAR Sends `clear`.

LU0STSN Sends `stsn`, parm2 has ru data, 5 bytes.

LU0STAT Sends `lustat`, parm2 has 4 sense bytes.

LU0QEC Sends `quiesce at end of chain`.

LU0QC Sends `quiesce complete`.

LU0RELQ Sends `release quiesce`.

LU0CAN Sends `cancel partially sent chain`.

LU0CHASE Sends `chase`.

LU0SHUTD	Sends `shutdown`.
LU0BID	Sends `bid`.
LU0SIG	Sends `signal`, parm2 has 4 sense bytes.

rc Specifies the return value indicating the success or failure of the **lu0ctlp** subroutine.

Return Values

If the function completes successfully, a 0 will be returned. If errors are encountered, –1 is returned and the **errno** global variable is set to one of the error codes specified in the **lu0.h** file.

File

/usr/lpp/lu0/lu0.h Specifies the LU0 header file containing common definitions.

Related Information

The **lu0closep** subroutine, **lu0openp** subroutine, **lu0readp** subroutine, **lu0writep** subroutine.

Application Program Interface in *Communications Programming Concepts*.

luOctls Subroutine

Purpose

Allow the application to send SNA commands to the host.

Syntax

#include <lu0.h>
extern int luOctls();

rc = luOctls(_luid_, _senseptr_, _optcode_**);**
int _luid_;
char *_senseptr_;
int _optcode_;
int _rc_;

Description

The **luOctls** subroutine controls function for SNA Secondary LU0 sessions. The **luOctls** subroutine sends SNA commands or responses to the host system. The _luid_ parameter is the LU identifier returned on a previous **luOopens** subroutine.

Parameters

luid Specifies the LU identifier that was returned by the **luOopens** subroutine.

senseptr Points to the address of the sense data buffer (6 bytes).

optcode Specifies the option code. The following are valid values for the _optcode_ parameter:

LUOACPT Sends `accept` to previous request.

LUOREJ Sends `reject` to previous request, parm2 has 4 bytes of sense data.

LUORSTAT Receives sense bytes (6) from **LUSTAT** or response information into parm2 buffer.

LUOFSM Returns 4 FSM bytes into parm2 buffer.

LUORQR Sends `request recovery`.

LUOSTAT Sends `lustat`, parm2 has 4 sense bytes.

LUORTR Sends `ready to receive`.

LUOQEC Sends `quiesce at end of chain`.

LUOQC Sends `quiesce complete`.

LUORELQ Sends `release quiesce`.

LUOCAN Sends `cancel partially sent chain`.

LUOCHASE Sends `chase`.

LU0SHUTC Sends `shutdown complete`.

LU0SIG Sends `signal`, parm2 has 4 sense bytes.

rc Specifies the return value indicating the success or failure of the **lu0ctls** subroutine.

Return Values

If the function completes successfully a 0 is returned. If errors are encountered, –1 is returned and the **errno** global variable is set to one of the error codes specified in the **lu0.h** file.

File

/usr/lpp/lu0/lu0.h Specifies the LU0 header file containing common definitions.

Related Information

The **lu0closes** subroutine, **lu0opens** subroutine, **lu0reads** subroutine, **lu0writes** subroutine.

Application Program Interface in *Communications Programming Concepts*.

lu0openp Subroutine

Purpose

Allows the application to begin a session with a secondary application.

Syntax

#include <lu0.h>
extern int lu0openp();

rc = lu0openp (*luname*, *bindptr*, *bindlenp*);
char **luname*;
char **bindptr*;
int **bindlenp*;
int *rc*;

Description

The **lu0openp** subroutine opens a SNA Primary LU0 session. The **lu0openp** subroutine opens a session with the secondary application. The value returned for the *luid* parameter is used on subsequent operations to define the LU identifier.

Parameters

luname	Specifies the address of an 8 byte name in the configuration file. The name should be left justified and padded with spaces.
bindptr	Points to the address of a 256 byte bind record buffer. On input to this routine, if the buffer contains a bind record (first byte = 0x31), then it will be the bind record sent to the secondary LU. Otherwise, the bind record is formatted from the information in the configuration file LU record and will be copied into this buffer.
bindlenp	Specifies the address of the integer that contains the length of the bind record. If the bind record is formatted by the caller in parm 2, then parm 3 should point to an integer that contains the length of the bind record. Otherwise parm 3 should point to an integer into which the length of the bind record is returned.
rc	Specifies the return value indicating the success or failure of the **lu0openp** subroutine.

Return Value

If the open is successful, the LU identifier is returned. The LU identifier is actually the address of the LU0 table entry in shared memory. If there are open errors, –1 is returned and the **errno** global variable is set to one of the error codes specified in the **lu0.h** file.

File

/usr/lpp/lu0/lu0.h	Specifies the LU0 header file containing common definitions.

Related Information

The **lu0closep** subroutine, **lu0ctlp** subroutine, **lu0readp** subroutine, **lu0writep** subroutine.

Application Program Interface in *Communications Programming Concepts.*

lu0opens Subroutine

Purpose

Allows the application to begin a session with a host application.

Syntax

#include <lu0.h>
extern int lu0opens();

luid = lu0opens (*luname*, *bindptr*, *bindlenp*);
int *luid*;
char **luname*;
char **bindptr*;
int **bindlenp*;

Description

The **lu0opens** subroutine opens a SNA Secondary LU0 session. The **lu0opens** subroutine opens a session with the host application. The value returned for the *luid* parameter is used on subsequent operations to define the LU identifier.

Parameters

luname Specifies the address of an 8-byte name in the configuration file. The name should be left-justified and padded with spaces.

bindptr Points to the address of a 256 byte buffer into which the bind record is placed.

bindlenp Specifies the address of the integer into which the length of the bind record is placed.

luid Specifies the return value indicating the success or failure of the **lu0opens** subroutine.

Return Values

If the open is successful, the LU identifier is returned. The LU identifier is actually the address of the LU0 table entry in shared memory. After a successful open, the application should check the bind record that was returned. It should then call the **lu0ctls** subroutine with the LU0ACPT function code to accept the BIND request or the LU0REJ function code to reject the BIND request. No other processing can be done until the bind has been responded to. If there are open errors, –1 is returned and the **errno** global variable is set to one of the error codes specified in the **lu0.h** file.

File

/usr/lpp/lu0/lu0.h Specifies the LU0 header file containing common definitions.

Related Information

The **lu0closes** subroutine, **lu0ctls** subroutine, **lu0reads** subroutine, **lu0writes** subroutine.

Application Program Interface in *Communications Programming Concepts.*

lu0readp Subroutine

Purpose

Allows the application to receive data from the secondary LU.

Syntax

#include <lu0.h>
extern int lu0readp ();

rc = lu0readp(*luid*, *buffptr*, *bufflen***)**
int *luid*;
char **buffptr*;
int *bufflen*;
int *rc*;

Description

The **lu0readp** subroutine reads a request from a SNA Primary LU0 session. The **lu0readp** subroutine receives data from the secondary LU.

Parameters

luid Specifies the LU identifier returned by the **lu0openp** subroutine.

buffptr Points to the address of the PIU buffer.

bufflen Specifies the length of the PIU buffer. The buffer length should be long enough to hold the largest request plus the SNA TH and RH.

rc Specifies the return value indicating the success or failure of the **lu0readp** subroutine.

Return Value

If the read is successful, the length of the PIU is returned If there are read errors, –1 is returned and the **errno** global variable is set to one of the error codes specified in the **lu0.h** file.

File

/usr/lpp/lu0/lu0.h Specifies the LU0 header file containing common definitions.

Related Information

The **lu0closep** subroutine, **lu0ctlp** subroutine, **lu0openp** subroutine, **lu0writep** subroutine.

Application Program Interface in *Communications Programming Concepts.*

lu0reads Subroutine

Purpose

Allows the application to receive data from the host application.

Syntax

#include <lu0.h>
extern int lu0reads();

rc = lu0reads(*luid*, *buffptr*, *bufflen***);**
int *luid*;
char **buffptr*;
int *bufflen*;
int *rc*;

Description

The **lu0reads** subroutine reads a request from a SNA Secondary LU0 session. The **lu0reads** subroutine receives data from the host application.

Parameters

luid	Specifies the LU identifier returned by the **lu0opens** subroutine.
buffptr	Points to the address of the PIU buffer.
bufflen	Specifies the length of the PIU buffer. The buffer length should be long enough to hold the largest request plus the SNA TH and RH.
rc	Specifies the return value indicating the success or failure of the **lu0reads** subroutine.

Return Value

If the read is successful, the length of the PIU is returned. If there are read errors, –1 is returned, and the **errno** global variable is set to one of the error codes specified in the **lu0.h** file.

File

/usr/lpp/lu0/lu0.h Specifies the LU0 header file containing common definitions.

Related Information

The **lu0closes** subroutine, **lu0ctls** subroutine, **lu0opens** subroutine, **lu0writes** subroutine.

Application Program Interface in *Communications Programming Concepts*.

lu0writep Subroutine

Purpose

Allows the application to send data to the secondary LU.

Syntax

#include <lu0.h>
extern int lu0writep();

rc = lu0writep(*luid*, *buffptr*, *bufflen***);**
int *luid*;
char **buffptr*;
int *bufflen*;
int *rc*;

Parameters

luid	Specifies the LU identifier returned by a previous **lu0openp** subroutine.
buffptr	Points to the address of the PIU.
bufflen	Specifies the length of the PIU.
rc	Specifies the return value indicating the success or failure of the **lu0writep** subroutine.

Description

The **lu0writep** subroutine writes a request in a SNA Primary LU0 session. The **lu0writep** subroutine sends data to the secondary LU. The following fields will be set in the TH and RH:

TH FID type = 2

Destination address field
Origination address field
Sequence number field

RH Type = request

RU category = FMD
No sense data
No enciphered data
No padded data
No conditional end bracket

Return Value

If the write is successful, the length of the PIU is returned If there are write errors, –1 is returned and the **errno** global variable is set to one of the error codes specified in the **lu0.h** file.

File

/usr/lpp/lu0/lu0.h
Specifies the LU0 header file containing common definitions.

Related Information

The **lu0closep** subroutine, **lu0ctlp** subroutine, **lu0openp** subroutine, **lu0readp** subroutine.

Application Program Interface in *Communications Programming Concepts*.

lu0writes Subroutine

Purpose

Allows the application to send data to the host application.

Syntax

#include <lu0.h>
extern int lu0writes();

rc = lu0writes (*luid*, *buffptr*, *bufflen***);**
int *luid*;
char **buffptr*;
int *bufflen*;
int *rc*;

Description

The **lu0writes** subroutine writes a request in a SNA Secondary LU0 session. The **lu0writes** subroutine sends data to the host application. The following fields are set in the TH and RH:

TH FID type = 2

Destination address field
Origination address field
Sequence number field

RH Type = request

RU category = FMD
No sense data
No enciphered data
No padded data
No conditional end bracket

Parameters

luid Specifies the LU identifier returned by the **lu0opens** subroutine.

buffptr Points to the address of the PIU.

bufflen Specifies the length of the PIU.

rc Specifies the return value indicating the success or failure of the **lu0writes** subroutine.

Return Value

If the write is successful, the length of the PIU is returned If there are write errors, –1 is returned and the **errno** global variable is set to one of the error codes specified in the **lu0.h** file.

File

/usr/lpp/lu0/lu0.h
Specifies the LU0 header file containing common definitions.

Related Information

The **lu0closes** subroutine, **lu0ctls** subroutine, **lu0opens** subroutine, **lu0reads** subroutine.

Application Program Interface in *Communications Programming Concepts*.

nm_close Subroutine

Purpose

Releases the SSCP_PU session.

Syntax

int nm_close (*sscp_id***)**
char ****sscp_id*;**

Description

An application uses the **nm_close** subroutine to release the specified session so another application can use it.

Parameter

sscp_id An ID predefined by the host for a specific SSCP–PU session.

Return Values

When the subroutine completes successfully, it returns a 0 (zero) to indicate that the specified session is closed. Otherwise, the subroutine returns a value of –1 and sets the **errno** global variable to indicate the error.

Error Code

The subroutine sets the **errno** global variable to a value to indicate the cause of any errors that occur. The values that this variable can receive are shown below. Error Code Constants in *Communications Programming Concepts* contains a brief description of the error values for AIX SNA Services/6000.

SNA_INVALID

File

/usr/include/luxsna.h
Defines constants and structures used by AIX SNA Services/6000 subroutines.

Related Information

The **nmopen** subroutine, **nmsend** subroutine, **nmrecv** subroutine, **nmstat** subroutine.

nm_open Subroutine

Purpose

Establishes the SSCP_PU session.

Syntax

int nm_open (*sscp_id*, *application_server_name*, *sna_server_name*)

char **sscp_id*;
char **application_server_name*;
char **sna_server_name*;

Description

An application uses the **nm_open** subroutine to associate itself with the specified session. The **nm_open** subroutine should be the first network management API subroutine called.

Parameters

sscp_id — An ID predefined by the host for a specific SSCP–PU session.

application_server_name
A pointer to the application server name. The *argv*[0] parameter contains the server name.

sna_server_name
A pointer to the `sna` server name. If the name pointer is NULL, the default SNA server name is `sna`.

Return Values

When the subroutine completes successfully, it returns a zero to indicate that the specified session is active and not being used by another application. Otherwise, the subroutine returns a value of –1 and sets the **errno** global variable to indicate the error.

Error Codes

The subroutine sets the **errno** global variable to a value to indicate the cause of any errors that occur. The values that this variable can receive are shown in the following list. Error Code Constants in *Communications Programming Concepts*contains a brief description of the error values for AIX SNA Services/6000.

SNA_ERP **SNA_INACT**
SNA_INUSE **SNA_NOTAVAIL**
SNA_UNDEF_SVR

File

/usr/include/luxsna.h
Defines constants and structures used by AIX SNA Services/6000 subroutines.

Related Information

The **nm_send** subroutine, **nm_recv** subroutine, **nm_close** subroutine, **nm_stat** subroutine.

nm_receive Subroutine

Purpose

Receives NMVT data from the specified SSCP_PU session.

Syntax

int nm_receive (*sscp_id*, *buffer*, *length*, *type*)

char **sscp_id*;
char **buffer*;
int *length*
int **type*

Description

An application uses the **nm_receive** subroutine to receive NMVT data for the specified session. If no data is available for that session, the subroutine waits until data is available or until the application ends the subroutine.

Parameters

sscp_id — An ID predefined by the host for a specific SSCP–PU session.

buffer — A pointer to the buffer area where the data is received.

length — The number of bytes of data. If this value is less that the actual number of bytes received, the message is truncated and the remainder is discarded.

type — The type of data. The following types are returned:

- **0** Request
- **1** Positive response
- **2** Negative response.

Return Values

When the subroutine completes successfully, it returns a positive integer that indicates the number of bytes received and placed in the user buffer. Otherwise, the subroutine returns a value of –1 and sets the **errno** global variable to indicate the error.

Error Codes

The subroutine sets the **errno** global variable to a value to indicate the cause of any errors that occur. The values that this variable can receive are shown in the following list. Error Code Constants in *Communications Programming Concepts* contains a brief description of the error values for AIX SNA Services/6000.

SNA_ERP

SNA_INVALID

SNA_INACT

SNA_LENGTH

File

/usr/include/luxsna.h
Defines constants and structures used by AIX SNA Services/6000 subroutines.

Related Information

The **nm_open** subroutine, **nm_send** subroutine, **nm_close** subroutine, **nm_stat** subroutine.

nm_send Subroutine

Purpose

Sends NMVT data to the specified SSCP_PU session.

Syntax

int nm_send (*sscp_id*, *data*, *length*, *type*)

char **sscp_id*;
char **data*;
int *length*
int *type*

Description

An application uses the **nm_send** subroutine to send NMVT data, including the NMVT header, to the specified session. The application must specify the data, the data length and the data type. SNA sets RH according to the data type:

request	`0x0B8000`
positive response	`0x838000`
negative response	`0x879000`

Parameters

sscp_id An ID predefined by the host for a specific SSCP–PU session.

data A pointer to the buffer area from which the data is sent. The data should contain a sense code if the *type* parameter is negative response.

length The number of bytes of data sent

type The type of data. The following types are allowed:

0 Request

1 Positive response

2 Negative response.

Return Values

When the subroutine completes successfully, it returns a positive integer that indicates the number of bytes sent. Otherwise, the subroutine returns a value of –1 and sets the **errno** global variable to indicate the error.

Error Codes

The subroutine sets the **errno** global variable to a value to indicate the cause of any errors that occur. The values that this variable can receive are shown in the following list. Error Code Constants in *Communications Programming Concepts* contains a brief description of the error values for AIX SNA Services/6000.

SNA_ERPSNA_INACT

SNA_NMVT_HDR

SNA_INVALID

SNA_LENGTH

SNA_STATE

File

/usr/include/luxsna.h
Defines constants and structures used by AIX SNA Services/6000 subroutines.

Related Information

The **nm_open** subroutine, **nm_recv** subroutine, **nm_close** subroutine, **nm_stat** subroutine.

nm_status Subroutine

Purpose

Provides the status of the specified SSCP_PU session.

Syntax

int nm_status (*sscp_id*)

char **sscp_id*;

Description

An application uses the **nm_status** subroutine to obtain the status of a session. There are three types of status: active, inactive, and reset.

Parameter

sscp_ID An ID predefined by the host for a specific SSCP–PU session.

Return Values

When the subroutine completes successfully, it returns a 0 (zero) to indicate that the specified session is active. Otherwise, the subroutine returns a value of –1 and sets the **errno** global variable to indicate the error.

Error Codes

The subroutine sets the **errno** global variable to a value to indicate the cause of any errors that occur. The values that this variable can receive are shown in the following list. Error Code Constants in *Communications Programming Concepts* contains a brief description of the error values for AIX SNA Services/6000.

SNA_ERP (reset)

SNA_INVALID

SNA_INACT (inactive)

File

/usr/include/luxsna.h
Defines constants and structures used by AIX SNA Services/6000 subroutines.

Related Information

The **nm_open** subroutine, **nm_send** subroutine, **nm_recv** subroutine, **nm_stat** subroutine.

open Subroutine for SNA Services/6000

Purpose

Opens a resource.

Syntax

#include <luxsna.h>

int open(*path*, *oflag***)**
char **path*;
int *oflag*;

Description

The **open** subroutine for AIX SNA Services/6000 initializes a connection to a resource described in a specified connection profile. You must use the **open** subroutine before using any other SNA subroutine for a particular connection. Each **open** subroutine ties a local LU to a remote LU.

Note: Opening a connection causes the associated attachment to be started if the attachment is not already active.

Limited Interface

When the connection profile defines the connection to be `limited`, the **open** subroutine also allocates the conversation between the local LU and the remote LU using the connection created by the **open** subroutine. The allocation uses the default allocation parameters as defined in the connection profile. Only one conversation can be allocated to this connection.

Extended Interface

The calling program must also allocate conversations to the connection after it is opened. Use the **ioctl** subroutine to perform the allocation.

For LUs 1, 2 and 3, only two sessions (SSCP–LU, LU–LU) can be allocated to a particular open file descriptor (connection). However, for LU 6.2 several conversations can be allocated to an open file descriptor.

Parameters

path Specifies the resource to be opened. It must be in the form:

ddn/*cpn*[/*tpn*]

The parameters in this string have the following meanings:

ddn Specifies the SNA device driver name to be used to open the resource. This will always be in the **/dev/sna** directory.

cpn Specifies the connection profile name of the resource to be opened.

tpn — Specifies the remote transaction profile name to be used in place of the transaction profile name found in the connection profile. This parameter is optional. If you do not specify this parameter, the **open** subroutine uses the remote transaction profile name found in the connection profile. If you specify this parameter, separate it from the connection profile name with a / (slash).

oflag — Specifies a value to set the file status flag. The *oflag* parameter values for the SNA device driver are constructed by logically ORing flags from the following list:

O_RDWR — Open for reading and writing

O_NDELAY — Subsequent reads will return immediately if no data is present.

Return Values

When the routine completes successfully, it returns a non–negative integer that specifies the file descriptor (*fildes*) or connection ID (`cid`) for the connection. If an error occurs, the routine returns a value of –1 and sets the **errno** global variable to indicate the error.

Error Codes

The routine sets the **errno** global variable to a value to indicate the cause of any errors that occur. The values that this variable can receive are shown in following list. Error Code Constants in *Communications Programming Concepts* contains a brief description of the following error values for AIX SNA Services/6000:

ENOENT	**ENOMEM**
EMFILE	**ENXIO**
ENOTDIR	**ETXTBSY**
EACCES	**EROFS**
EFAULT	**EISDIR**

File

/usr/include/luxsna.h
Defines constants and structures used by AIX SNA Services/6000 subroutines.

Related Information

The **open** subroutine, **close** subroutine.

The **close** subroutine for SNA Services/6000, **ioctl** subroutine for SNA Services/6000, **snaopen** subroutine for SNA Services/6000.

open Subroutine for Generic SNA

Purpose

Opens a file descriptor.

Syntax

#include <luxgsna.h>

int openx(*path, oflag, mode, ext***)**
char **path*;
int *oflag*;
int *mode*;
int *ext*;

Description

The **open** subroutine initializes resources to tie an AIX SNA Services/6000 attachment to a file descriptor such that each file descriptor corresponds to an AIX SNA Services/6000 attachment. This command must be issued before using any other generic SNA device driver subroutine.

Parameters

path Specifies the resource to be opened. It must be in the following form:

`/dev/gsna/attachment_profile_name`

The `attachment_profile_name` in the path is required. It is used to start an AIX SNA Services/6000 attachment only if AIX SNA Services/6000 is running and an appropriate AIX SNA Services/6000 attachment profile is defined. AIX SNA Services/6000 supports only PU type 2.1 nodes.

oflag Specifies the value of the file status flag. The generic SNA device driver uses only the following values of this flag (all other values are ignored):

O_RDWR	Open for reading and writing
O_NDELAY	Subsequent reads will return immediately if no data is present.

mode Ignored by generic SNA.

ext Ignored by generic SNA.

Return Values

Upon successful completion, the **open** subroutine returns a 0 (zero). If an error occurs, it returns a value of –1 and sets **errno** to indicate the error.

Error Codes

The **open** subroutine sets **errno** to a value which indicates the cause of any errors that occur. The values that **errno** can receive are shown in the following list:

ENXIO	No such device or address exists.
ENOENT	The named file does not exist.
EACCES	A component of the path prefix denies search permission, or permission is denied for the named file.
ENOMEM	Either this node or the server does not have enough memory available to service the request.
EISDIR	The named file is a directory, and the *oflag* parameter is write or read/write.
ENOTDIR	A component of the path prefix is not a directory.
ETXTBSY	The file is a pure procedure (shared text) file that is being executed and the *oflag* parameter is write or read/write.
EROFS	The named file resides on a read–only file system, and the *oflag* parameter is write or read/write.
EMFILE	The maximum number of file descriptors are currently open.
EFAULT	The *path* parameter points to a location outside the process's allocated address space.
SNA_NO_LU	No LUs are registered for the Generic SNA device driver.
SNA_FAIL	SNA system failure. SNA is not currently running.

Related Information

The **close** Subroutine for Generic SNA, **read** Subroutine for Generic SNA, **write** Subroutine for Generic SNA, **ioctl** Subroutine for Generic SNA, **select** Subroutine for Generic SNA.

Developing Special AIX SNA Services/6000 Functions in *Communications Programming Concepts.*

read Subroutine for SNA Services/6000

Purpose

Receives data from the remote transaction program.

Syntax

#include <luxsna.h>

int read(*fildes*, *data*, *length*)

int *fildes*;
char **data*;
int *length*;

Description

Note: Use this subroutine for LU 6.2 limited connections only. Applications using LU 6.2 extended connections or LU 1, LU 2, or LU 3 connections should use the **readx** subroutine.

The **read** subroutine waits for information to arrive on the specified conversation and then receives the information. If the information is already available, it receives the information without waiting.

When trying to read from a conversation that has no data available, the state of the O_NDELAY flag (see the **open** or **fcntl** subroutines) determines what happens to the read operation:

set The read returns a value of 0 to indicate that no data has been received.

clear The read is blocked until data becomes available.

An application program should not use the **read** subroutine unless the program is performing a very simple function. Use the **readx** subroutine instead. The **readx** subroutine returns additional information to inform the application program what state it is in. The **read** subroutine does not provide that information.

If the application program uses this command when the conversation is in send state, the following actions occur:

1. The LU flushes its send buffer, sending all buffered information and the send indication to the remote program,
2. The local program enters the receive state and waits for data from the remote program.

Extended Interface

When using the extended interface for LU 6.2 connections with multiple conversations, only the first conversation is accessible with this command.

Parameters

fildes Specifies the variable that contains the file descriptor returned by the **open** subroutine.

data Specifies a pointer to the buffer area into which the data will be read.

length Specifies the variable that contains a value indicating the maximum number of bytes of data to be received. This value cannot be larger than 32,764 (32K – 4) bytes.

Return Values

When the subroutine completes successfully, it returns a positive integer that indicates the number of bytes received. If an error occurs, the routine returns a value of –1 and sets the **errno** global variable to indicate the error.

If an interrupt occurs while the subroutine is processing, it returns the number of bytes that have already been transferred to the user buffer. If no data has been moved when the interrupt occurs, it returns a value of –1 and sets the **errno** global variable to EINTR.

Error Codes

The subroutine sets the **errno** global variable to a value to indicate the cause of any errors that occur. The values that this variable can receive are shown in the following list. Error Code Constants in *Communications Programming Concepts* contains a brief description of the following error values for AIX SNA Services/6000:

EBADF	**SNA_NPIP**	**SNA_PROTOCOL**
EINTR	**SNA_NREC**	**SNA_PTR**
EINVAL	**SNA_NRMDEAL**	**SNA_RFN**
ENOMEM	**SNA_NRREC**	**SNA_RFR**
SNA_ALFN	**SNA_NSES**	**SNA_RREC**
SNA_ALFR	**SNA_NSYC**	**SNA_SNTR**
SNA_BOUNDARY	**SNA_PGMDEAL**	**SNA_SPURG**
SNA_CTYPE	**SNA_PNREC**	**SNA_STATE**
SNA_INVACC	**SNA_PNSYC**	**SNA_SVCDEAL**
SNA_NOCONN	**SNA_PNTR**	**SNA_TIMDEAL**
SNA_NOTPN	**SNA_PPURG**	**SNA_WRGPIP**

File

/usr/include/luxsna.h
Defines constants and structures used by AIX SNA Services/6000 subroutines.

Related Information

The **read** subroutine.

The **write** subroutine for SNA Services/6000, **ioctl** subroutine for SNA Services/6000, **open** subroutine for SNA Services/6000.

read Subroutine for Generic SNA

Purpose

Receives data from a file descriptor.

Syntax

#include <luxgsna.h>

int read (*fildes, data, length)*
int readx (*fildes, data, length, ext)*
int *fildes*;
char **data*;
int *length*;
int *ext*;

Description

The **read** subroutine receives normal sequenced data, exchange ID (XID) data, network data, or datagram data.

A **read** subroutine can be issued for normal data when the **open** subroutine has completed successfully.

The **read** subroutine waits for information to arrive on the specified AIX SNA Services/6000 attachment, then receives the information. If information is already available, it receives the information without waiting.

Parameters

fildes	Specifies the file descriptor returned by the **open** subroutine.
data	Specifies a pointer to the buffer area into which the data will be read.
length	Specifies the maximum number of bytes of data to be received. This value cannot be larger than 32,764 (32K – 4) bytes.
ext	Ignored by generic SNA.

Return Values

Upon successful completion, this subroutine returns a non–negative integer that indicates the number of bytes received. If an error occurs, it returns a value of –1 and sets **errno** to indicate the error.

Error Codes

The call sets **errno** to a value that indicates the cause of any errors that occur, as shown in the following list:

EBADF	An invalid file descriptor was specified.
EFAULT	An invalid address was specified.
EINTR	The **read** subroutine was interrupted.

EINVAL	An invalid parameter was passed.
SNA_HIER_RESET	Hierarchical Reset was received from AIX SNA Services/6000.
SNA_INOP	INOP was received from AIX SNA Services/6000.
SNA_FAIL	SNA system failure.

Related Information

The **close** Subroutine for Generic SNA, **open** Subroutine for Generic SNA, **write** Subroutine for Generic SNA, **ioctl** Subroutine for Generic SNA, **select** Subroutine for Generic SNA.

Developing Special AIX SNA Services/6000 Functions in *Communications Programming Concepts*.

readx Subroutine for SNA Services/6000

Purpose

Receives data from the remote transaction program.

Syntax

#include <luxsna.h>

int readx (*fildes*, *data*, *length*, *ext*)
int *fildes*;
char **data*;
int *length*;
struct ext_io_str **ext*;

Description

Note: Do not use this subroutine for LU 6.2 programs that use the limited interface.

The **readx** subroutine waits for information to arrive on the specified conversation and then receives the information. If the information is already available, it receives the information without waiting. The information can be data, conversation status, or a request for confirmation. The connection profile for this connection must designate the use of the extended interface as described in Defining AIX SNA Services/6000 Characteristics in *Communication Concepts and Procedures*.

Note: SNA Services defines all LU 1, LU 2, and LU 3 connections as extended interface connections.

If the application program uses this command when the conversation is in send state, the following actions occur:

1. The LU flushes its send buffer, sending all buffered information and the send indication to the remote program,
2. The local program enters the receive state and waits for information from the remote program.

If you specify the `rid` parameter in the **ext_io_str** structure to be 0 (zero) or **NULL**, the subroutine performs a `read any` operation. The data that the routine returns is from the first resource allocated to the connection that has data available. The subroutine then sets the `rid` parameter to indicate the resource ID of the resource that supplied the data.

When trying to read from a conversation that has no data available, the state of the **O_NDELAY** flag (see the **open** or **fcntl** subroutine) determines what happens to the read operation:

set The read returns a value of 0 to indicate that no data has been received.

clear The read is blocked until data becomes available.

When header information is received, it is moved into the `usrhdr` field of the extended I/O structure, **ext_io_str**. If the header information is longer than the space allowed (specified in the `usrhdr_len` parameter), the `usr_trunc` field is set to indicate the error. No information, data or header is returned when the header is truncated.

The *ext* parameter points to a structure that contains additional input and output parameters for the **readx** subroutine. This same structure is used for the **writex** subroutine. Refer to the **ext_io_str** structure for a description of the fields. The **readx** subroutine uses only the following fields in that structure:

- `deallocate`
- `deallo_type` (`type=B'010'` only - deallocate with abnormal end of conversation)
- `deallo_flag`
- `allocate`
- `fill`
- `sess_type`
- `rq_to_snd_rcvd`
- `what_data_rcvd`
- `what_control_rcvd`
- `sense_code`
- `rid`
- `usrhdr_len`
- `usr_trunc`
- `usrhdr`

User Header Field

In addition to the data provided in the extended I/O structure, the sending program can supply header information about the data being sent. The receiving program must allow for receiving this information by doing the following:

1. Define the length of the header information in the `usrhdr_len` field.
2. Reserve consecutive space following the extended I/O structure (`ext_io_str`) for the header information.
3. Pass the extended I/O structure pointer (the *ext* parameter) in a **readx** subroutine.

LUs 1, 2 and 3

The amount of data returned depends upon the type of information read (FM header or data), the length specified in the read, and the size of the data. For FM headers data, the amount of data depends upon the size of the FM header. For a chain element, the amount of data depends upon the size of the chain element request unit.

Use only the buffer value for the `fill` option. Use `confirm_deallocate` and `normal_deallocate` parameters only on an LU–LU session to indicate the end of a bracket.

Do not use the following:

- `confirm_deallocate_retain`
- `normal_deallocate_retain`
- `confirm_deallocate` on an SSCP–LU session
- `normal_deallocate` on an SSCP–LU session.

When FM header data is received, the FM header data is moved to the user buffer. The `what_data_rcvd` field is set to indicate the receipt of FM header data. If data was received in addition to the FM header data, a separate **readx** subroutine must be issued to obtain the data.

If the host bids for the session, a **readx** subroutine with the allocate bit on accepts the bid. For additional information, refer to the ALLOCATE section of the **ioctl** subroutine and to the **writex** subroutine.

Parameters

fildes — Specifies the variable that contains the file descriptor returned by the **open** subroutine.

data — Specifies a pointer to the buffer area into which the data will be read.

length — Specifies the variable that contains a value indicating the maximum number of bytes of data to be received. This value cannot be larger than 32,764 (32K – 4) bytes.

ext — Specifies a pointer to an extended I/O structure of type **ext_io_str**. The **ext_io_str** structure allows the user to combine functions into one routine. You can use **readx**(ALLOCATE and DEALLOCATE) on one routine. This structure type is defined in the **luxsna.h** include file. See the **ext_io–str** structure.

Return Values

When the subroutine completes successfully, it returns a positive integer that indicates the number of bytes received. If an error occurs, the routine returns a value of –1 and sets the **errno** global variable to indicate the error.

If an interrupt occurs while the subroutine is processing, it returns the number of bytes that have already been transferred to the user buffer. If no data has been moved when the interrupt occurs, it returns a value of –1 and sets the **errno** global variable to `EINTR`.

Error Codes

The subroutine sets the **errno** global variable to a value to indicate the cause of any errors that occur. The values that this variable can receive are shown in the following list. Error Code Constants in *Communications Programming Concepts* contains a brief description of the following error values for AIX SNA Services/6000:

EBADF	**SNA_NPIP**	**SNA_PROTOCOL**
EINTR	**SNA_NREC**	**SNA_PTR**
EINVAL	**SNA_NRMDEAL**	**SNA_RFN**
ENOMEM	**SNA_NRREC**	**SNA_RFR**
SNA_ALFN	**SNA_NSES**	**SNA_RREC**
SNA_ALFR	**SNA_NSYC**	**SNA_SNTR**
SNA_BOUNDARY	**SNA_PGMDEAL**	**SNA_SPURG**
SNA_CTYPE	**SNA_PNREC**	**SNA_STATE**

SNA_INVACC	**SNA_PNSYC**	**SNA_SVCDEAL**
SNA_NOCONN	**SNA_PNTR**	**SNA_TIMDEAL**
SNA_NOTPN	**SNA_PPURG**	**SNA_WRGPIP**

File

/usr/include/luxsna.h
Defines constants and structures used by AIX SNA Services/6000 subroutines.

Related Information

The **readx** subroutine.

The **writex** subroutine for SNA Services/6000, **ioctl** subroutine for SNA Services/6000, **open** subroutine for SNA Services/6000, **snaread** subroutine for SNA Services/6000.

select Subroutine for SNA Services/6000

Purpose

Examines a file descriptor or message queue.

Syntax

#include <sys/time.h>
#include <sys/select.h>

int select(*nfdsmsgs*, *readlist*, *writelist*, *exceptlist*, *timeout***)**
ulong *nfdsmsgs*;
void **readlist*;
void **writelist*;
void **exceptlist*;
struct timeval **timeout*;

Description

The **select** subroutine examines a set of resource IDs (file descriptors) to determine how many of the indicated resources:

- Are available for reading or receiving data
- Are available for writing or sending data (not supported by AIX SNA Services/6000)
- Have an outstanding exceptional condition.

Exceptional conditions include the status conditions listed in the **gstat_str** structure.

AIX SNA Services/6000 supports the **select**(data received) and **select**(exceptional conditions) subroutines.

A time-out value is also provided to prevent the subroutine from waiting for a response for too long a period of time.

Each of the operations (**read**, **write** or **exception**) is described with a structure of type **sellist** that is defined in the **/usr/include/sys/select.h** file. This structure is defined as follows:

```
struct sellist
{
        long  fdsmask[ ];
        long  msgids[ ];
}
```

The additional parameters have the following meanings:

fdsmask[ ] Specifies an array of `int` values that are used as a continuous stream of bits. Each long value contains 32 bits, so that the first array member contains bits 0 through 31, the second array member contains bits 32 through 63, and so forth. The bit number plus one corresponds to the file descriptor that the number represents (that is, bit 35 represents file descriptor 36). The SELECT operation examines all file descriptors up to the limit specified in the *nfdsmsgs* parameter for the condition corresponding to this structure (**read**, **write**, or **exception**). When the subroutine returns, it sets the bits in this structure that represent the file descriptors that satisfied the examination to a value of 1. To disable file-descriptor checking for an operation, set all members of this array to a value of 0.

msgids[ ] Specifies an array of `int` values. Each long value is a message queue identifier that specifies a message queue to be examined. The **select** operation examines the message queues for each ID up to the number of IDs indicated by the limit specified in the *nfds* parameter for the condition corresponding to this structure (**read**, **write**, or **exception**). When the subroutine returns, it sets all members of this array whose queues do *not* satisfy the examination to a value of 0xFFFFFFFF.

Message queue checking is not supported by AIX SNA Services/6000 and should be disabled by setting all members of this array to a value of 0xFFFFFFFF. The value provided in the *nfdsmsgs* parameter determines the size that must be given to the *fdsmask[ ]* and *msgids[ ]* arrays.

The /**usr/include/sys/select.h** header file also defines two macros to help split the *nfdsmsgs* parameter and the return value into their component halves:

NFDS(*nfdsmsgs*) Extracts the number of file descriptors.

NMSGS(*nfdsmsgs*) Extracts the number of message queues.

Parameters

nfdsmsgs A long integer that is evaluated in two halves, described as follows:

Low 16 bits Contain the number of file descriptor bits to use from the mask value provided in the *fdsmask[ ]* array of the structure of type **sellist** for each of the operations. If this value is 0x0000, no file-descriptor checking is performed.

High 16 bits Not supported by AIX SNA Services/6000 and should be set to 0x0000.

readlist Points to a structure defined by the SELLIST( ) macro that specifies file descriptors for examination to see if they are ready for reading or receiving data.

writelist Points to a structure defined by the SELLIST( ) macro that specifies file descriptors for examination to see if they are ready for writing or sending data. Do not use this structure with SNA Services.

exceptlist Points to a structure defined by the SELLIST( ) macro that specifies file descriptors for examination to see if they have an exception condition pending.

timeout Points to a structure of type **timeval** that indicates the maximum number of seconds or microseconds to wait for the selection to complete. If this value is 0, the operation waits indefinitely. For polling, this value should be nonzero, pointing to a zero-valued structure.

Return Values

When the subroutine completes successfully, it returns an integer value. The integer is evaluated in two halves:

Low 16 bits	This half contains the total number of file descriptors that satisfied the selection criteria (for all requested operations: **read**, **write**, and **exception**).
High 16 bits	Not supported by SNA Services.

In addition, the SELECT operation modifies the **sellist** structures to indicate which file descriptors were selected:

- Selected file descriptor bits are set to 1.
- Not selected file descriptor bits are set to 0.

If the time limit runs out, the subroutine returns a value of 0. If an error occurs, the **ioctl** subroutine returns a value of –1. In either case, the subroutine sets the **errno** global variable to indicate the error.

Error Codes

The subroutine sets the **errno** global variable to a value to indicate the cause of any errors that occur. The values that this variable can receive are shown in the following list.

EBADF	One of the bit masks specified an invalid file descriptor.
EINTR	A signal interrupted the subroutine before it found any of the selected events, or the time limit ran out.

Files

/usr/include/sys/times.h
Defines constants and structures used by the AIX operating system.

/usr/include/sys/select.h
Defines constants and structures used by the AIX operating system.

Related Information

The **select** subroutine.

select Subroutine for Generic SNA

Purpose

Examines a set of file descriptors.

Syntax

#include <sys/time.h>
#include <sys/select.h>

int select(*nfdsmsgs*, *readlist*, *writelist*, *exceptlist*, *timeout***)**

ulong *nfdsmsgs*;

void **readlist*;

void **writelist*;

void **exceptlist*;

struct timeval **timeout*;

Description

The **select** subroutine examines a set of resource IDs (file descriptors) to determine how many of the indicated resources:

- Are available for reading or receiving data
- Are available for writing or sending data (not supported by AIX SNA Services/6000)
- Have an outstanding exception condition.

The Generic SNA device driver supports the **select** (data received) and the **select** (exception condition). The **select** (write available) condition will always be satisfied because the Generic SNA device driver does not support the *writelist* parameter.

The **select** (exception condition) results from one of the following:

- INOP received from the PU Services of AIX SNA Services/6000.
- Hierarchical_Reset received from the PU Services of AIX SNA Services/6000.

After the **select** subroutine completes successfully, the application may issue a **read** subroutine to get the exception condition (returned by the **errno** of the **read** subroutine).

A timeout value is also provided to prevent the operation from waiting for a long response time. If the *nfdsmsgs* parameter is a value of 0, the **select** subroutine acts as a timer and returns after the time period specified in the **timeval** structure.

The **read**, **write**, and **exception** events are described with an unnamed structure, which is defined by a macro in the **/usr/include/sys/select.h** header file. This macro is defined as follows:

```
#define SELLIST (F,M)
  struct
  {
      int fdsmask[F];
      int msgids[M];
  };
```

The additional parameters have the following meanings:

fdsmask[F] Specifies an array of int values that is used as a continuous stream of bits. Each long value contains 32 bits, so that the first array member contains bits 0 through 31, the second array member contains bits 32 through 63, etc. The bit number plus one corresponds to the file descriptor that it represents (that is, bit 35 represents file descriptor 36). The SELECT operation examines all file descriptors up to the limit specified in the *nfdsmsgs* parameter for the condition corresponding to this structure (**read, write** or **exception**). When the subroutine returns, it sets the bits in this structure that represent the file descriptors that satisfied the examination to a value of 1. To disable a file descriptor checking for an operation, set all members of this array to a value of 0.

msgids[M] Specifies an array of int values. Each long value is a message queue identifier that specifies a message queue to be examined. The SELECT operation examines the message queues for each ID up to the number of IDs indicated by the limit specified in the *nfdsmsgs* parameter for the condition corresponding to this structure (**read**, **write** or **exception**). When the subroutine returns, it sets all members of this array whose queues do not satisfy the examination to a value of −1. To disable message queue checking for an operation, set all members of this array to a value of −1.

Message queue checking is not supported by AIX SNA Services/6000 and should be disabled by setting all members of this array to a value of 0xFFFFFFFF. The value provided in the *nfdsmsgs* parameter determines the size that must be given to the *fdsmask[F]* and *msgids[M]* arrays.

The /**usr/include/sys/select.h** header file also defines two macros to help split *nfdsmsgs* and the return value into their component halves:

NFDS(*nfdsmsgs*)
Extracts the number of file descriptors.

NMSGS(*nfdsmsgs*)
Extracts the number of message queues.

Parameters

nfdsmsgs A long integer that is evaluated in two halves, described as follows:

Low 16 bits: Contains the number of file descriptor bits to use from the mask value provided in the *fdsmask[F]* array of the structure of type sellist for each of the operations. If this value is 0x0000, no file descriptor checking is performed.

High 16 bits: Not supported by AIX SNA Services/6000 and should be set to 0x0000.

readlist Points to a structure defined by the SELLIST( ) macro that specifies file descriptors for examination to see if they are ready for reading or receiving data.

writelist Points to a structure defined by the SELLIST( ) macro that specifies file descriptors for examination to see if they are ready for writing or sending data. Do not use this structure with generic SNA device driver subroutines.

exceptlist — Points to a structure defined by the SELLIST( ) macro that specifies file descriptors for examination to see if they have an exception condition pending.

timeout — Points to a structure of type **timeval** that indicates the maximum number of seconds or microseconds to wait for the selection to complete. If this value is 0, the operation waits indefinitely. For polling, this value should be non–zero, pointing to a zero–valued structure.

Return Values

When the subroutine completes successfully, it returns an integer value that is evaluated in two halves, described as follows:

Low 16 bits — Contains the total number of file descriptors that satisfy the selection criteria (for all requested subroutines: **read**, **write,** and **exception**).

High 16 bits — Contains the number of message queues that satisfy the selection criteria (for all requested subroutines: **read**, **write,** and **exception**).

In addition, the SELECT operation modifies the **sellist** structures to indicate which file descriptors and message queues are selected:

- Selected file descriptor bits are set to 1.
- Unselected file descriptor bits are set to 0.
- Selected message queue IDs remain unchanged.
- Unselected message queue IDs are set to –1.

If the time limit runs out, the subroutine returns a value of 0. If an error occurs, the **ioctl** subroutine returns a value of –1. In either case, it sets **errno** to indicate the error.

Error Codes

The call sets **errno** to a value that indicates the cause of any errors that occur, as shown in the following list:

EBADF — One of the bit masks specified an invalid file descriptor or message queue index.

EINTR — A signal interrupted the subroutine before it found any of the selected events, or the time limit ran out.

EINVAL — A bad timeout value was given in the *timeout* parameter.

EAGAIN — An internal storage allocation problem was detected.

EFAULT — A bad address value was passed in one of the parameters.

Related Information

The **close** Subroutine for Generic SNA, **open** Subroutine for Generic SNA, **read** Subroutine for Generic SNA, **write** Subroutine for Generic SNA, **ioctl** Subroutine for Generic SNA.

Developing Special AIX SNA Services/6000 Functions in *Communications Programming Concepts*.

snaclse Subroutine

Purpose

Closes a connection.

Syntax

#include <luxsna.h>

int snaclse(*cid***)**
int *cid*;

Description

The **snaclse** subroutine closes a connection specified by its connection ID. Deallocate all conversations on the connection before closing the connection (see the **snadeal** subroutine). If the conversations are not deallocated, closing the connection causes an abnormal end of the conversation.

Parameter

cid Specifies the variable that contains the connection ID returned by the **snaopen** subroutine.

Return Values

When the subroutine completes successfully, it returns a value of 0. If an error occurs, the subroutine returns a value of –1 and sets the **errno** global variable to indicate the error.

Error Code

The subroutine sets the **errno** global variable to a value that indicates the cause of any errors that occur. The values that this variable can receive are shown below. Error Code Constants in *Communications Programming Concepts* contains a brief description of the error values for AIX SNA Services/6000.

EBADF

File

/usr/include/luxsna.h
Defines constants and structures used by AIX SNA Services/6000 subroutines.

Related Information

The **snaopen** subroutine, **snadeal** subroutine.

snactl Subroutine

Purpose

Controls data transfer between local and remote transaction programs.

Syntax

#include <luxsna.h>

int snactl(*cid*, *request*, *arg*, *c_type*)
int *cid*;
int *request*;
int *arg*;
char *c_type*;

Description

This subroutine provides control functions for transfer operations between a local and a remote transaction program. The control function is specified by the *request* parameter, and must be one of the integers (defined in the **luxsna.h** include file) explained in the following paragraphs:

- ALLOCATE_LISTEN (LU 6.2 only)
- CONFIRM
- CONFIRMED
- CP_STATUS (LU 6.2 only)
- FLUSH
- GET_ATTRIBUTE (LU 6.2 only)
- GET_PARAMETERS (LU 6.2 only)
- GET_STATUS (LUs 1, 2, and 3 only)
- PREPARE_TO_RECEIVE
- REQUEST_TO_SEND
- SEND_ERROR
- SEND_FMH (LU 1 only)
- SEND_STATUS (LUs 1, 2, and 3 only).

CONFIRM

The CONFIRM request asks the remote transaction program to tell whether the last transmission was successfully received. The remote transaction program must respond with one of two **snactl** requests: CONFIRMED or SEND_ERROR.

LU 6.2

The program may use the CONFIRM request for the following special cases:

- Directly following a **snalloc** function to determine if the allocation of the conversation was successful before sending data
- Following transmission of data to the remote program to get an acknowledgment from the remote program.

For LU 6.2, the *arg* parameter specifies a pointer to a structure of type **confirm_str**, which contains additional input parameters for the request. Refer to the **confirm_str** structure for field descriptions.

LUs 1, 2, and 3

LU 1 uses the CONFIRM request to get an acknowledgement for data that it sent to the remote program. However, LUs 2 and 3 do not use this request for that purpose. The remote program must handle error recovery for the local LU 2 or 3 program.

For LUs 1, 2, and 3, the *arg* parameter specifies a pointer to a structure of type **confirm_str** which contains additional parameters for the request. Refer to the **confirm_str** structure for field descriptions.

CONFIRMED

The CONFIRMED request is a response to the CONFIRM request indicating that the remote site received the transmission without detecting any errors. This request cannot be used except in response to a CONFIRM request.

This request can be used to create and send a positive response to the remote session to indicate the successful receipt of a command.

For this request, the *arg* parameter specifies the resource ID.

CP_STATUS requests information about the capabilities of the control point at the remote node. The request includes the resource ID, the `rid` parameter, returned from the ALLOCATE request.

The remote node responds with its control point name and the session type, contention winner CONWINNER or contention loser CONLOSER. The remote node also returns a list of capabilities, each followed by a `YES` or `NO`, indicating whether the feature is supported.

For this request, the *arg* parameter is a pointer to a structure of type **cp_status**. This structure contains the parameters that are sent and the parameters that are returned. Refer to the **cp_str** structure for field descriptions.

FLUSH

The FLUSH request sends any information in the local LU send buffer to the remote LU. This function can be used on a basic conversation only. The LU normally buffers the data from **snawrit** functions until it has enough data to transmit. Using this request the local program forces the local LU to transmit the data in the buffer. The local program can use this request to decrease the delay required to get the data to the remote system.

The *arg* parameter for this request is a pointer to a structure of type **flush_str**, which contains the input and output parameters for the request. Refer to the **flush_str** structure for field descriptions.

GET_ATTRIBUTE

The GET_ATTRIBUTE request gets information about the specified LU 6.2 conversation.

The *arg* parameter for this request is a pointer to a structure of type **attr_str**, which contains the input parameter rid and receives the output information from the request. Refer to the **attr_str** structure for field descriptions.

GET_STATUS

The GET_STATUS request gets information about the current link and session, as well as information from the BIND request for the LU–LU session. This information is used for LUs 1, 2, and 3 only.

The *arg* parameter for this request is a pointer to a structure of type **gstat_str**, which contains the the output status information from the request. Refer to the **gstat_str** structure for field descriptions.

PREPARE_TO_RECEIVE

The PREPARE_TO_RECEIVE request notifies the remote LU that the local LU needs to change the conversation direction so that the local LU can begin receiving from the remote LU.

The *arg* parameter for this request is a pointer to a structure of type **prep_str**, which contains the input and output parameters for the request. Refer to the **prep_str** structure for field descriptions.

REQUEST_TO_SEND

The REQUEST_TO_SEND request notifies the remote LU that the local LU needs to change the conversation direction so that the local LU can begin sending to the remote LU. The local program uses a **readx** subroutine to get the send indication from the remote program (in the `what_control_rcvd` field). When the local program receives this indication from the remote program, it enters the send state.

For this request, the *arg* parameter specifies the resource ID.

SEND_ERROR

The SEND_ERROR request informs the remote transaction program that the local transaction program has detected an error in the information that it received from the remote program.

The *arg* parameter for this request is a pointer to a structure of type **erro_str**, which contains the input parameters for the request. Refer to the **erro_str** structure for field descriptions.

LU 6.2

When this request is issued in send state, the LU:

1. Flushes the local send buffer.
2. Creates and sends an FMH7 request.

When the FMH7 request is issued in receive state, the LU:

1. Generates a negative response.
2. Purges all incoming data.
3. Waits for a send indication to arrive from the remote program.
4. Creates an FMH7 request and sends it.
5. Enters send state to send the error message.

LUs 1, 2, and 3

When the FMH7 request is issued in send state, the LU:

1. Flushes the send buffer.
2. Sends a CANCEL request to the remote session.

When this request is issued in receive state, the LU:

1. Generates a negative response, using the `sense_code` parameter specified in the **erro_str** structure.
2. Purges all incoming data to the end of chain.

SEND_FMH

The SEND_FMH request sends the FM header to the remote LU. Since the SEND_FMH request is used only by LU 1 support, it must be used on a basic conversation.

The *arg* parameter for this request is a pointer to a structure of type **fmh_str**, which contains the input parameters for the request. Refer to the **fmh_str** structure for field descriptions.

The application program must build the complete FM header to be sent. If more than one FM header is to be sent, the application must build all FM headers with the concatenation bit set within a contiguous area. The application program must also enforce concatenation and chaining rules.

SEND_STATUS

The SEND_STATUS request sends status information about the devices on the local session (LUs 1, 2, and 3, only) to the host program. This request can be used on a basic conversation only. When issued in send state, an LUSTAT is sent to the remote session, using the ID to indicate which device the LUSTAT is for. This request is used for LU1 LU–LU sessions only.

The *arg* parameter for this request is a pointer to a structure of type **stat_str**, which contains the input parameters for the status request. Refer to the **stat_str** structure for field descriptions.

Parameters

cid — Specifies the variable that contains the connection ID returned by the **snaopen** subroutine.

request — Specifies the function to be performed as defined in the **luxsna.h** include file.

arg — Specifies the variable that contains one of the following (varies with the function performed as specified in the *request* parameter):

- The resource ID returned by the **snalloc** subroutine.
- A pointer to a structure that contains additional input parameters for the requested function.

c_type — Specifies a character constant that indicates the conversation type:

`'B'` — The request is performed on a basic conversation.

`'M'` — The request is performed on a mapped conversation.

Return Values

When the subroutine completes successfully, it returns a non–negative integer that is equal to 0 for most requests. However, when the CONFIRM and SEND_ERROR requests complete successfully, they return one of the following values:

0 Successful completion, but did not receive a request to send.

1 Successful completion and received a request to send from the remote transaction program.

Additional information, if any, is stored in the structures provided by the specific request. If an error occurs, the subroutine returns a value of –1 and sets the **errno** global variable to indicate the error.

Error Codes

The subroutine sets the **errno** global variable to a value to indicate the cause of any errors that occur. The values that this variable can receive vary with the requested function as shown in the following table. Error Code Constants in *Communications Programming Concepts* contains a brief description of the following error values for AIX SNA Services/6000.

ERRNO VALUE / REQUEST TYPE	CONFIRM	CONFIRMED	FLUSH	GET_ATTR	GET_STATUS	PEEP_RECVUS	CP_STATUS	SEND_ERROR	SEND_STATUS	SEND_FMH	RTS
EBADF	●	●	●	●	●	●	●	●	●	●	●
EINTR	●					●		●			
EINVAL	●	●	●		●	●		●	●	●	
ENOMEM	●	●				●		●			●
ENXIO										●	
SNA_ALFN	●					●		●	●		
SNA_ALFR	●					●		●	●		
SNA_BOUNDARY	●					●					
SNA_CTYPE	●					●		●			
SNA_DEAL									●	●	
SNA_DEALPGM									●	●	
SNA_EXCEED										●	
SNA_INVACC	●					●		●			
SNA_NFMH	●										
SNA_MAPEXEC	●										
SNA_MAP_NOTFND	●										
SNA_NMAP	●										
SNA_NOCONN	●	●		●		●		●			●
SNA_NOTPN	●					●		●			
SNA_NPIP	●					●		●			
SNA_NREC	●	●				●		●			●
SNA_NRMDEAL	●					●		●			
SNA_NRREC	●					●		●			
SNA_NSES	●	●	●			●		●	●	●	●
SNA_NSYC	●	●				●		●			●
SNA_PGMDEAL	●					●		●			
SNA_PGMPURGE									●	●	
SNA_PNREC	●					●		●			
SNA_PNSYC	●					●		●			
SNA_PNTR	●					●		●			
SNA_PPURG	●					●		●			
SNA_PROTOCOL	●	●				●		●			●
SNA_PTR	●					●		●			
SNA_RFN	●					●		●	●	●	
SNA_RFR	●					●		●	●	●	
SNA_RREC	●					●		●			
SNA_SHUT			●			●			●	●	
SNA_SNTR	●					●		●			
SNA_SPURG	●					●		●			
SNA_STATE	●	●	●			●		●	●	●	●
SNA_SVCDEAL	●					●		●			
SNA_TIMDEAL	●					●		●			
SNA_WRGPIP	●					●		●			
SNA_NOTCP							●				

Figure 1. The **snactl Subroutine Error Returns**

File

/usr/include/luxsna.h
Defines constants and structures used by AIX SNA Services/6000 subroutines.

Related Information

Node Verification in Defining LU Type 6.2 Connection Characteristics in *Communication Concepts and Procedures*.

The **snaopen** subroutine, **snalloc** subroutine.

snadeal Subroutine

Purpose

Deallocates the specified conversation from the transaction program.

Syntax

#include <luxsna.h>

int snadeal(*cid*, *ptr*, *c_type*)
int *cid*;
struct deal_str **ptr*;
char *c_type*;

Description

The **snadeal** subroutine removes the allocation of the specified conversation from the local transaction program. Information about the deallocation is supplied in the structure of type **deal_str** pointed to by the *ptr* parameter. Refer to the **deal_str** structure for field descriptions.

LU 6.2

The **snadeal** subroutine ends the conversation but not the session. The LU resource manager determines whether to keep or end the session.

Although a deallocation with a `type` field of `local` occurs in the general SNA specifications, do not use that type with AIX SNA Services/6000. The SNA device driver performs the local deallocation function when the device driver receives a deallocate request from the remote transaction program. The device driver then sets the `what_control_rcvd` field in the **read_out** structure to indicate the type of deallocation the device driver received from the remote program. See the **snaread** subroutine for an explanation of the **read_out** structure.

LUs 1, 2, and 3

Do not use the **snadeal** subroutine with an LU–LU session for either LUs 2 or 3. If used with these sessions, the subroutine returns with an SNA_STATE error. Using **snadeal** with an LU–LU session for LU 1 ends a bracket.

Use the `confirm_deallocate` and `normal_deallocate` parameters only on an LU–LU session to indicate the end of a bracket.

Do not use the following parameters:

- `confirm_deallocate_retain`
- `normal_deallocate_retain`
- `confirm_deallocate` on an SSCP–LU session
- `normal_deallocate` on an SSCP–LU session.

To deallocate an LU–LU session that has a corresponding SSCP–LU session, use the **snadeal** subroutine to deallocate the SSCP–LU session. When the local transaction program issues a **snadeal** routine with `type` set to `B'010'` (flush) for the SSCP–LU session, the local LU issues an RSHUTD to request an UNBIND negotiation to terminate the LU–LU session. The local LU rejects all data from the host on the LU–LU session until it receives the UNBIND request. The local LU rejects all data from the host on the SSCP–LU session until that session is allocated to another application program.

The host can issue an UNBIND request at other times to end the LU–LU session. When the UNBIND request occurs, the local program using the LU–LU session receives a return code of SNA_NSES that notifies this program of the session end. If the application uses a GET_STATUS request of the **snactl** subroutine, the returned status indicates that the session is not active.

The local LU cannot issue a DACTLU request to end the SSCP–LU session. Therefore, the session remains active until the host ends it with a DACTLU, DACTPU or ACTPU request. In this case, the **snadeal** subroutine used on the SSCP–LU session removes the connection to the local program, but does not remove the SSCP–LU session itself.

When used on an LU 1 LU–LU session, the **snadeal** subroutine ends a bracket.

Parameters

cid — Specifies the variable that contains the connection ID returned by the **snaopen** subroutine.

ptr — Specifies a pointer to a structure that contains additional input parameters.

c_type — Specifies a character constant that indicates the conversation type:

- `'B'` — The request is performed on a basic conversation.
- `'M'` — The request is performed on a mapped conversation.

Return Values

When the subroutine completes successfully, it returns a value of `0`. If an error occurs, the subroutine returns a value of `-1` and sets the **errno** global variable to indicate the error.

Error Codes

The subroutine sets the **errno** global variable to a value to indicate the cause of any errors that occur. The values that this variable can receive are shown in the following list. Error Code Constants in *Communications Programming Concepts* contains a brief description of the error values for AIX SNA Services/6000.

EBADF	**SNA_MAP**	**SNA_PPURG**
EINTR	**SNA_NOCONN**	**SNA_PROTOCOL**
EINVAL	**SNA_NOTPN**	**SNA_PTR**
ENOMEM	**SNA_NPIP**	**SNA_RFN**
SNA_ALFN	**SNA_NREC**	**SNA_RFR**
SNA_ALFR	**SNA_NRMDEAL**	**SNA_RREC**
SNA_BOUNDARY	**SNA_NRREC**	**SNA_SNTR**
SNA_CTYPE	**SNA_NSYC**	**SNA_SPURG**
SNA_INVACC	**SNA_PGMDEAL**	**SNA_STATE**
SNA_MAPEXEC	**SNA_PNREC**	**SNA_SVCDEAL**

SNA_MAP_NOTFND	**SNA_PNSYC**	**SNA_TIMDEAL**
SNA_NFMH	**SNA_PNTR**	**SNA_WRGPIP**

File

/usr/include/luxsna.h
Defines constants and structures used by AIX SNA Services/6000 subroutines.

Related Information

The **snactl** subroutine, **snaopen** subroutine, **snalloc** subroutine.

snalloc Subroutine

Purpose

Creates a session and conversation between two transaction programs.

Syntax

#include <luxsna.h>

long snalloc(*cid*, *allo_ptr*, *c_type***)**
int *cid*;
struct allo_str **allo_ptr*;
char *c_type*;

Description

The **snalloc** subroutine allocates a session between the local LU and a remote LU. Then it allocates a conversation between the local transaction program and a remote transaction program using the allocated session. The subroutine returns a resource ID to identify the conversation. Use this subroutine before using any other subroutine that refers to the conversation.

The *allo_ptr* parameter is a pointer to a structure of type **allo_str**, which contains additional information for the subroutine. This structure contains a pointer to an additional structure `pip_str`. Refer to the **allo_str** and **pip_str** structures for field descriptions.

LU 6.2

When this subroutine completes successfully, the local transaction program (the one that used this subroutine) is in the send state and the remote transaction program is in the receive state.

If two LUs that are connected by a session try to allocate a conversation on that session at the same time, one of the LUs is successful and the other is not. Which LU is successful is determined by the BIND negotiation that occurred when the session was established.

For two programs to reconnect to each other, the following events must occur:

1. One program uses the **snadeal** subroutine with the `deal_flag` parameter set to *retain* to deallocate the conversation.
2. The program initiating the reconnection uses the **snalloc** subroutine with the `type` parameter set to *reconnect*. This action sends a reconnection request to the remote LU.
3. The remote program completes the reconnection when it uses the **snaread** subroutine to receive information.

LUs 1, 2, and 3

When this subroutine completes successfully, the appropriate session (SSCP–LU or LU–LU) is established and both LUs are in HDX contention state. The SSCP–LU session must be active and allocated before allocating the LU–LU session. Trying to allocate the LU–LU session before the SSCP–LU session results in a SNA_STATE error return from the **snalloc** subroutine.

If an application program tries to allocate an SSCP–LU session and the ACTLU request has not yet been received, the subroutine returns the SNA_NSES error code (session not active) and does not allocate a session. If the application program then uses the GET_STATUS request of the **snactl** subroutine, the returned status shows that the SSCP–LU session is inactive.

If an application program tries to allocate an LU–LU session, but the BIND request for that session has not yet been received, the result depends upon whether the LU names for both LUs of the requested session are specified in the connection profile:

- LU Names Specified: The local LU sends an INIT_SELF request on the SSCP–LU session to request the needed BIND negotiation. When the LU receives and accepts the BIND request, it then completes the requested allocation of the LU–LU session.
- LU Names Not Specified: The LU–LU session cannot be allocated. The subroutine returns the SNA_NSES error code to indicate that the session is not active.

If the host bid for the session, the **snalloc** subroutine rejects the bid. See the ALLOCATE section of the **ioctl** subroutine for more information.

Parameters

cid
: Specifies the variable that contains the connection ID returned by the **snaopen** subroutine.

allo_ptr
: Specifies a pointer to the structure that contains additional input parameters for the subroutine. If you do not provide this information, the subroutine uses the values contained in the remote transaction profile specified in the **snaopen** subroutine. If no remote transaction profile is specified in the **snaopen** subroutine, the first remote transaction profile in the connection profile associated with the **snaopen** subroutine is used.

c_type
: Specifies a character constant that indicates the conversation type:

  `'B'` The request is performed on a basic conversation. When the `type` field of the **allo_str** structure is `B'11'` (see `type`), this parameter indicates that a basic conversation is to be reconnected.

  `'M'` The request is performed on a mapped conversation. When the `type` field of the **allo_str** structure is `B'11'` (see `type`), this parameter indicates that a mapped conversation is to be reconnected.

Return Values

When the subroutine completes successfully, it returns a positive integer that indicates the resource ID for the conversation. If an error occurs, the subroutine returns a value of `-1` and sets the **errno** global variable to indicate the error.

Error Codes

The subroutine sets the **errno** global variable to a value that indicates the cause of any errors that occur. The values that this variable can receive are shown in the following list. Error Code Constants in *Communications Programming Concepts* contains a brief description of the error values for AIX SNA Services/6000.

EBADF	**SNA_LUNSYC**	**SNA_NRESTART**
EINTR	**SNA_NIMMED**	**SNA_NSYC**
ENOMEM	**SNA_NOCONN**	**SNA_PROTOCOL**
SNA_ALFN	**SNA_NOMODE**	**SNA_RFN**
SNA_ALFR	**SNA_NOTPN**	**SNA_RFR**
SNA_LUNREC	**SNA_NREC**	**SNA_STATE**

File

/usr/include/luxsna.h
Defines constants and structures used by AIX SNA Services/6000 subroutines.

Related Information

The **snadeal** subroutine, **snaopen** subroutine, and **snaread** subroutine.

snaopen Subroutine

Purpose

Opens an SNA connection.

Syntax

#include <luxsna.h>

int snaopen(*resource***)**
**char ****resource*;

Description

The **snaopen** subroutine initializes a connection to a resource described in a specified connection profile. You must use the **snaopen** subroutine before using any other SNA subroutine for a particular connection.

Parameters

The *resource* parameter consists of:

cpn Specifies the connection profile name of the resource to be opened.

tpn Specifies the remote transaction profile name to be used in place of the remote transaction profile name found in the connection profile. The *tpn* parameter is optional. If you do not specify this parameter, the **snaopen** subroutine uses the remote transaction profile name found in the connection profile. If you specify this parameter, separate it from the connection profile name with a / (slash). Remote transaction profiles are used in LU 6.2 only. Do not supply this parameter for LUs 1, 2, or 3.

Return Values

When the subroutine completes successfully, it returns a positive integer that specifies the connection ID (*cid*) for the connection. If an error occurs, the subroutine returns a value of –1 and sets the **errno** global variable to indicate the error.

Error Codes

The subroutine sets the **errno** global variable to a value to indicate the cause of any errors that occur. The values that this variable can receive are shown in the following list. Error Code Constants in *Communications Programming Concepts* contains a brief description of the error values for AIX SNA Services/6000.

EEXIST	**ENOTDIR**	**EROFS**
EINVAL	**ENXIO**	**EMFILE**
ENOENT	**EACCES**	**ETXTBSY**
ENOMEM	**EISDIR**	**EFAULT**

File

/usr/include/luxsna.h
Defines constants and structures used by AIX SNA Services/6000 subroutines.

Related Information

The **snaclse** subroutine.

snaread Subroutine

Purpose

Receives information from a specified conversation.

Syntax

#include <luxsna.h>

int snaread (*cid*, *data*, *length*, *rid*, *fill*, *output_ptr*, *c_type*)
int *cid*;
char **data*;
int *length*;
int *rid*;
int *fill*;
struct read_out **output_ptr*;
char *c_type*;

Description

The **snaread** subroutine waits for information to arrive on the specified conversation and then receives the information. If information is already available, the program receives it without waiting. The information can be data, conversation status, or a request for confirmation.

The program uses this subroutine when in the send state, causing the following actions to occur:

1. The LU flushes its send buffer, sending all buffered information and the send indication to the remote program,
2. The local program enters the receive state and waits for information from the remote program.

When trying to read from a conversation that has no data available, the state of the **O_NDELAY** flag (see the **open** or **fcntl** subroutine) determines what happens to the read operation:

set The read returns a value of 0 and sets the `what_data_rcvd` field (in the **ext_io_str** structure) to indicate whether the received data was complete.

clear The read is blocked until data becomes available.

To perform a `read_any` function, the **O_NDELAY** flag must be set on.

LUs 1, 2, and 3

Use only the `buffer` value for the *fill* option.

Parameters

cid Specifies the variable that contains the connection ID returned by the **snaopen** subroutine.

data Specifies a pointer to the buffer area into which the data will be read.

length Specifies the variable that contains a value indicating the maximum number of bytes of data to be received. This value cannot be larger than 32,764 (32K – 4) bytes.

rid Specifies the variable that contains the resource ID returned by the **snalloc** subroutine that allocated the resource to be read. If you do not specify a value for the *rid* parameter(a null value), the subroutine performs a `read_any` operation. It reads the first resource that is allocated to the program and that has data to be read.

output_ptr Specifies a pointer to a structure of type **read_out**, which contains space for the output parameters from this subroutine.

c_type Specifies a character constant that indicates the conversation type:

`'B'` The request is performed on a basic conversation.

`'M'` The request is performed on a mapped conversation.

fill Specifies whether the program receives data in terms of the logical record format of the data. If you do not specify one of the two following values, the program uses a value of `buffer`. Always use a value of buffer for LUs 1, 2, and 3.

`buffer (0)` Specifies that the program receives data without regard to the logical record format of the data.

`ll (1)` Specifies that the program receives one complete logical record or a logical record that has been truncated to the length specified in the *length* parameter of this subroutine. This value is used for LU 6.2 basic conversations only.

The *output_ptr* parameter for this request is a pointer to a structure of type **read_out**, which receives the information produced by this subroutine. Refer to the **read_out** structure for field descriptions.

Return Values

When the subroutine completes successfully, it returns a positive integer that indicates the number of bytes received and places received data in the user buffer pointed to by the *data* parameter. Additional information is stored in the **read_out** structure. If an error occurs, the subroutine returns a value of –1 and sets the **errno** global variable to indicate the error.

If an interrupt occurs while the subroutine is processing, it returns the number of bytes that have already been transferred to the user buffer. If no data has been moved when the interrupt occurs, it returns a value of –1 and sets the **errno** global variable to EINTR.

Error Codes

The subroutine sets the **errno** global variable to a value that indicates the cause of any errors that occur. The values that this variable can receive are shown in the following list. Error Code Constants in *Communications Programming Concepts* contains a brief description of the error values for AIX SNA Services/6000.

EBADF	**SNA_NMAP**	**SNA_PPURG**
EINTR	**SNA_NOCONN**	**SNA_PROTOCOL**
EINVAL	**SNA_NOTPN**	**SNA_PTR**
ENOMEM	**SNA_NPIP**	**SNA_RFN**
SNA_ALFN	**SNA_NREC**	**SNA_RFR**
SNA_ALFR	**SNA_NRMDEAL**	**SNA_RREC**
SNA_BOUNDARY	**SNA_NRREC**	**SNA_SNTR**
SNA_CTYPE	**SNA_NSES**	**SNA_SPURG**
SNA_EC	**SNA_NSYC**	**SNA_STATE**
SNA_INVACC	**SNA_PGMDEAL**	**SNA_SVCDEAL**
SNA_NFMH	**SNA_PNREC**	**SNA_TIMDEAL**
SNA_MAPEXEC	**SNA_PNSYC**	**SNA_WRGPIP**
SNA_PNTR	**SNA_MAP_NOTFND**	

File

/usr/include/luxsna.h
Defines constants and structures used by AIX SNA Services/6000 subroutines.

Related Information

The **snactl** subroutine, **snadeal** subroutine, **snaopen** subroutine, **snalloc** subroutine.

snawrit Subroutine

Purpose

Sends data to the remote transaction program.

Syntax

#include <luxsna.h>

int snawrit (*cid*, *data*, *length*, *rid*, *write_ptr*, *c_type*)
int *cid*;
char **data*;
int *length*;
long *rid*;
struct *write_out* **write_ptr*;
char *c_type*;

Description

The **snawrit** subroutine sends data to the remote transaction program. The local LU buffers all data to be transmitted until the buffers contain enough data to make a transmission block, or until the local transaction program forces the LU to transmit the data (see the **snactl** and **snaread** subroutines).

Mapped Conversations

For mapped conversations, the data to be sent consists of data records that contain only data and no record length parameter. The *length* parameter of the **snawrit** subroutine defines the length of the data record. In addition, each **snawrit** subroutine can send up to 32,764 (32K – 4) bytes of data. Use several **snawrit** subroutines to send blocks of data that are longer than this limit.

Basic Conversations for LU 6.2

The data to be sent consists of logical records with a length that is determined by the local application program data format. The length is independent from the *length* parameter of this subroutine. A complete logical record contains the 2–byte `ll` field plus all bytes of a logical record from the local application program. The `ll` field contains the length of the complete logical record (the `ll` field plus the logical record). Transmission of a logical record is not complete until the last byte of logical record is sent. Each **snawrit** subroutine can send up to 32,764 (32K – 4) bytes of data.

The local program must finish sending a logical record before using any of the following subroutines:

- **snactl** with a CONFIRM request
- **snactl** with a PREPARE_TO_RECEIVE request
- **snaread**
- **snadeal** using a `type` field that is *not* `B'010'` (indicating an abnormal end).

Using any of these subroutines before the logical record transmission is complete, results in a bad return code from the subroutine.

Basic Conversations for LUs 1, 2 and 3

The data to be sent consists of chain elements with a length that is determined by the maximum request/response unit size specified in the BIND or ACTLU image.

Parameters

cid	Specifies the variable that contains the connection ID returned by the **snaopen** subroutine.
data	Specifies a pointer to the buffer area from which the data will be sent.
length	Specifies the variable that contains a value indicating the number of bytes of data to be sent. This value cannot be larger than 32,764 bytes (32K bytes minus 4 bytes).
rid	Specifies the variable that contains the resource ID returned by the **snalloc** subroutine.
write_ptr	Specifies a pointer to the write_out structure.
c_type	Specifies a character constant that indicates the conversation type:

`'B'`	The request is performed on a basic conversation.
`'M'`	The request is performed on a mapped conversation.

Return Values

When the subroutine completes successfully, it returns a positive integer that indicates the number of bytes sent. If an error occurs, the subroutine returns a value of –1 and sets the **errno** global variable to indicate the error.

If an interrupt occurs while the subroutine is processing, it returns the number of bytes that have already been transferred to the network. If no data has been transferred when the interrupt occurs, it returns a value of –1 and sets the **errno** global variable to **EINTR**.

Error Codes

The subroutine sets the **errno** global variable to a value to indicate the cause of any errors that occur. The values that this variable can receive are shown in the following list. Error Code Constants in *Communications Programming Concepts* contains a brief description of the error values for AIX SNA Services/6000.

EBADF	**SNA_NPIP**	**SNA_PROTOCOL**
EINTR	**SNA_NREC**	**SNA_PTR**
EINVAL	**SNA_NRESTART**	**SNA_RFN**
ENOMEM	**SNA_NRMDEAL**	**SNA_RFR**
SNA_ALFN	**SNA_NRREC**	**SNA_RREC**
SNA_ALFR	**SNA_NSES**	**SNA_SHUT**
SNA_CTYPE	**SNA_NSYC**	**SNA_SNTR**
SNA_INVACC	**SNA_PGMDEAL**	**SNA_SPURG**

SNA_NIMMED	**SNA_PNREC**	**SNA_STATE**
SNA_NOCONN	**SNA_PNSYC**	**SNA_SVCDEAL**
SNA_NOMODE	**SNA_PNTR**	**SNA_TIMDEAL**
SNA_NOTPN	**SNA_PPURG**	**SNA_WRGPIP**

LUs 1, 2, and 3

An **errno** value of SNA_SHUT indicates that a shutdown request had been received. The **snawrit** subroutine failed because the transmission requires a new bracket and new brackets are not allowed when shutdown is active. This value occurs during an LU–LU session only.

File

/usr/include/luxsna.h
Defines constants and structures used by AIX SNA Services/6000 subroutines.

Related Information

The **snactl** subroutine, **snadeal** subroutine, **snalloc** subroutine, **snaopen** subroutine, **snaread** subroutine.

write Subroutine for SNA Services/6000

Purpose

Sends data to the remote transaction program.

Syntax

#include <luxsna.h>

int write(*fildes*, *data*, *length***)**
int *fildes*;
char **data*;
int *length*;

Description

Note: Use this subroutine for LU 6.2 applications that use the limited interface only.

The **write** subroutine sends data to the remote transaction program. The local LU buffers all data to be transmitted until the buffers contain enough data to make a transmission block or until the local transaction program forces the LU to transmit the data (see the **ioctl**(FLUSH) subroutine). The **write** subroutine uses the first conversation if multiple conversations are active.

Data sent by the **write** subroutine consists of logical records. The length of the logical records is not determined by the *length* parameter of this subroutine. A complete logical record contains the 2–byte `ll` (logical length) field plus all bytes of a logical record from the local application program. The `ll` field contains the length of the complete logical record (`ll` field plus the logical record). Transmission of a logical record is not complete until the last byte of the logical record is sent.

The local program must finish sending a logical record before using any of the following subroutines:

- **read**
- **ioctl** with any of the following:
  - CONFIRM request
  - PREPARE_TO_RECEIVE request
  - DEALLOCATE request using a type field that is *not* `B'010'` (indicating an abnormal end).

Using any of these routines before the transmission is complete results in a return code from the routine that indicates that the local program has not finished sending a logical record.

Parameters

fildes Specifies the file descriptor returned by the **open** subroutine.

data Specifies a pointer to the buffer area from which the data will be sent.

length Specifies the variable that contains a value indicating the number of bytes of data to be sent.

Return Values

When the subroutine completes successfully, it returns a positive integer that indicates the number of bytes sent. If an error occurs, the routine returns a value of –1 and sets the **errno** global variable to indicate the error.

If the subroutine is interrupted by a signal, it returns the number of bytes that have already been transferred to the network. If no data has been moved when the interrupt occurs, it returns a value of –1 and sets the **errno** global variable to **EINTR**.

Error Codes

The subroutine sets the **errno** global variable to a value to indicate the cause of any errors that occur. The values that this variable can receive are shown in the following list. Error Code Constants in *Communications Programming Concepts* contains a brief description of the error values for AIX SNA Services/6000.

EBADF	**SNA_NPIP**	**SNA_PROTOCOL**
EINTR	**SNA_NREC**	**SNA_PTR**
EINVAL	**SNA_NRESTART**	**SNA_RFN**
ENOMEM	**SNA_NRMDEAL**	**SNA_RFR**
SNA_ALFN	**SNA_NRREC**	**SNA_RREC**
SNA_ALFR	**SNA_NSES**	**SNA_SHUT**
SNA_BOUNDARY	**SNA_NSYC**	**SNA_SNTR**
SNA_CTYPE	**SNA_PGMDEAL**	**SNA_SPURG**
SNA_INVACC	**SNA_PNREC**	**SNA_STATE**
SNA_NIMMED	**SNA_PNSYC**	**SNA_SVCDEAL**
SNA_NOCONN	**SNA_PNTR**	**SNA_TIMDEAL**
SNA_NOMODE	**SNA_PPURG**	**SNA_WRGPIP**
SNA_NOTPN		

File

/usr/include/luxsna.h
Defines constants and structures used by AIX SNA Services/6000 subroutines.

Related Information

The **write** subroutine.

The **ioctl** subroutine for SNA Services/6000, **open** subroutine for SNA Services/6000, **read** subroutine for SNA Services/6000.

write Subroutine for Generic SNA

Purpose

Sends data to a file descriptor.

Syntax

#include <luxgsna.h>

int write (*fildes, data, length)*
int writex (*fildes, data, length, ext)*
int *fildes*;
char **data*;
int *length*;
int *ext*;

Description

The **write** subroutine sends normal sequenced data, exchange ID (XID) data, network data, or datagram data to a file descriptor. A **write** subroutine can be issued for normal data when the **open** subroutine has completed successfully.

Parameters

fildes	Specifies the file descriptor returned by the **open** subroutine.
data	Specifies a pointer to the buffer area from which the data is sent.
length	Specifies the number of bytes of data to be sent.
ext	Ignored by generic SNA.

Return Values

Upon successful completion, the **write** subroutine returns a non–negative integer that indicates the number of bytes sent. If an error occurs, it returns a value of –1 and sets **errno** to indicate the error.

Error Codes

The call sets **errno** to a value which indicates the cause of any errors that occur, as shown in the following list:

EBADF	An invalid file descriptor was specified.
ENOMEM	No write buffer available.
EFAULT	An invalid address was specified.
EINTR	The **write** subroutine was interrupted.
EINVAL	An invalid parameter was passed.
SNA_HIER_RESET	Hierarchical Reset was received from AIX SNA Services/6000.

SNA_INOP	An INOP was received from AIX SNA Services/6000.
SNA_FAIL	SNA system failure. SNA is not currently running.

Related Information

The **close** Subroutine for Generic SNA, **open** Subroutine for Generic SNA, **read** Subroutine for Generic SNA, **ioctl** Subroutine for Generic SNA, **select** Subroutine for Generic SNA.

Developing Special AIX SNA Services/6000 Function in *Communications Programming Concepts.*

writex Subroutine for SNA Services/6000

Purpose

Sends data to the remote transaction program.

Syntax

#include <luxsna.h>

int writex (*fildes*, *data*, *length*, *ext*)

int *fildes*;
char **data*;
int *length*;
struct ext_io_str **ext*;

Description

Note: Do not use this subroutine for programs that use the limited interface.

The **writex** subroutine sends data to the remote transaction program. The local LU buffers all data to be transmitted until the buffers contain enough data to make a transmission, or until the local transaction program forces the LU to transmit the data (see the **ioctl** subroutine).

The *ext* parameter points to a structure that contains additional input and output parameters for the **writex** subroutine. This same structure is used for the **readx** subroutine. This structure is defined in the **luxsna.h** include file. Refer to the **ext_io_str** structure for a description of the fields in this structure. The **writex** subroutine uses only the following fields:

- `priority`
- `tpn_option`
- `confirm`
- `deallocate`
- `deallo_type`
- `deallo_flag`
- `allocate`
- `sess_type`
- `flush_flag`
- `rq_to_snd_rcvd`
- `rid`
- `usrhdr_len`

User Header Field

In addition to the data provided in the extended I/O structure **ext_io_str**, a program can supply header information about the data being sent. To do this, the program must:

1. Define the length of the header information in the `usrhdr_len` field of the **ext_io_str** structure.
2. Reserve consecutive space following the extended I/O structure (**ext_io_str**) for the header information.
3. Store the header information in the contiguous space following the **ext_io_str** structure.
4. Pass the extended I/O structure pointer (the *ext* parameter) in a **writex** subroutine.

LU 6.2

Data sent to the remote transaction program consists of logical records. The length of the logical records is not determined by the *length* parameter of this subroutine. A complete logical record contains the 2–byte `ll` (logical length) field plus all bytes of a logical record from the local application program. The `ll` field contains the length of the complete logical record (`ll` field plus the logical record). Transmission of a logical record is not complete until the last byte of the logical record is sent.

The local program must finish sending a logical record before using any of the following subroutines:

- **read**
- **ioctl** with any of the following:
  - CONFIRM request
  - PREPARE_TO_RECEIVE request
  - DEALLOCATE request, using a type field that is *not* B&ssq.010&ssq. (indicating an abnormal end).

Using any of these routines before the transmission is complete results in a bad return code from the routine that indicates that the local program has not finished sending a logical record.

LUs 1, 2, and 3

The data to be sent consists of chain elements whose length is determined by the maximum request/response unit size specified in the BIND or ACTLU request.

If an error (negative response) occurs during the write, the subroutine sets the **errno** global variable to SNA_PPURG and sets the `sense_code` field to indicate the sense data received in the negative response.

If the host bids for a session, a **writex** subroutine with the allocate bit on rejects the host bid and any data associated with the bid. For additional information, refer to the ALLOCATE section of the **ioctl** and **readx** subroutines.

Parameters

fildes	Specifies the variable that contains the file descriptor returned by the **open** subroutine.
data	Specifies a pointer to the buffer area from which the data will be sent.

length Specifies the variable that contains a value indicating the number of bytes of data to be sent.

ext Specifies a pointer to an extended I/O structure of type **ext_io_str**. The **ext_io_str** structure allows the user to combine functions into one routine. You can use the **writex**(ALLOCATE and DEALLOCATE) functions on one routine. This structure type is defined in the **luxsna.h** include file. See the **ext_io_str** structure.

Return Values

When the subroutine completes successfully, it returns a positive integer that indicates the number of bytes sent. If an error occurs, the routine returns a value of –1 and sets the **errno** global variable to indicate the error.

If an interrupt occurs while the subroutine is processing, it returns the number of bytes that have already been transferred to the network. If no data has been moved when the interrupt occurs, it returns a value of –1 and sets the **errno** global variable to EINTR.

Error Codes

The subroutine sets the **errno** global variable to a value to indicate the cause of any errors that occur. The values that this variable can receive are shown in the following list. Error Code Constants in *Communications Programming Concepts* contains a brief description of the error values for AIX SNA Services/6000.

EBADF	**SNA_NPIP**	**SNA_PROTOCOL**
EINTR	**SNA_NREC**	**SNA_PTR**
EINVAL	**SNA_NRESTART**	**SNA_RFN**
ENOMEM	**SNA_NRMDEAL**	**SNA_RFR**
SNA_ALFN	**SNA_NRREC**	**SNA_RREC**
SNA_ALFR	**SNA_NSES**	**SNA_SHUT**
SNA_BOUNDARY	**SNA_NSYC**	**SNA_SNTR**
SNA_CTYPE	**SNA_PGMDEAL**	**SNA_SPURG**
SNA_INVACC	**SNA_PNREC**	**SNA_STATE**
SNA_NIMMED	**SNA_PNSYC**	**SNA_SVCDEAL**
SNA_NOCONN	**SNA_PNTR**	**SNA_TIMDEAL**
SNA_NOMODE	**SNA_PPURG**	**SNA_WRGPIP**
SNA_NOTPN		

File

/usr/include/luxsna.h
Defines constants and structures used by AIX SNA Services/6000 subroutines.

Related Information

The **writex** subroutine.

The **readx** subroutine for SNA Services/6000, **ioctl** subroutine for SNA Services/6000, **open** subroutine for SNA Services/6000, **snawrit** subroutine for SNA Services/6000.

Sockets

accept Subroutine

Purpose

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

Syntax

#include <sys/types.h>

#include <sys/socket.h>

int accept (*Socket*, *Address*, *AddressLength*)
int *Socket*;
struct sockaddr **Address*;
int **AddressLength*;

Description

The **accept** subroutine extracts the first connection on the queue of pending connections, creates a new socket with the same properties as the specified socket, and allocates a new file descriptor for that socket.

If the **listen** queue is empty of connection requests, the **accept** subroutine:

- Blocks a calling socket of the blocking type until a connection is present.
- Returns an **EWOULDBLOCK** for sockets marked nonblocking.

The **accepted** socket cannot itself accept more connections. The original socket remains open and can accept more connections.

Parameters

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

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

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

Return Values

Upon successful completion, the **accept** subroutine returns the nonnegative socket descriptor of the accepted socket.

If the **accept** subroutine fails, the subroutine handler performs the following functions:

- Returns a value of –1 (negative one) to the calling program
- Moves an error code, indicating the specific error, into the global variable **ernno**.

Error Codes

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

EBADF	The *Socket* parameter is not valid.
ENOTSOCK	The *Socket* parameter refers to a file, not a socket.
EOPNOTSUPP	The referenced socket is not of type **SOCK_STREAM**.
EFAULT	The *Address* parameter is not in a writable part of the user address space.
EWOULDBLOCK	The socket is marked as nonblocking, and no connections are present to be accepted.

Examples

1. As illustrated in the following program fragment, once a socket is marked as listening, a server process may **accept** a connection:

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

2. The Accepting a UNIX Stream Connection program fragrment illustrates the use of the **accept** subroutine.

Implementation Specifics

The **accept** subroutine is part of AIX Base Operating System (BOS) Runtime.

All applications containing the **accept** subroutine must be compiled with **_BSD** defined. In addition, when applicable, all socket applications must include the BSD library **libbsd**.

Files

/usr/include/netinet/in.h	Contains Internet constants and structures.
/usr/include/sys/socket.h	Contains socket definitions.
/usr/include/sys/socketvar.h	Defines the kernel structure per socket and contains buffer queues.
/usr/include/sys/types.h	Contains unsigned data types.

Related Information

Other socket creation and connection subroutines are the **bind** subroutine, **connect** subroutine, **listen** subroutine, **select** subroutine, and **socket** subroutine.

Sockets Overview, Understanding Socket Creation, and Binding Names to Sockets in *Communications Programming Concepts.*

bind Subroutine

Purpose

Binds a name to a socket.

Syntax

#include <sys/types.h>

#include <sys/socket.h>

#include <netinet/in.h>

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

Description

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

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

Parameters

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

Return Values

Upon successful completion, the **bind** subroutine returns a value of 0 (zero).

If the **bind** subroutine fails, the subroutine handler performs the following actions:

- Returns a value of –1 (negative one) to the calling program
- Moves an error code, indicating the specific error, into the global variable **errno**

Error Codes

The **bind** subroutine fails if any one of the following errors occurs:

EBADF	The *Socket* parameter is not valid.
ENOTSOCK	The *Socket* parameter refers to a file, not a socket.

EADDRNOTAVAIL	The specified address is not available from the local machine.
EADDRINUSE	The specified address is already in use.
EINVAL	The socket is already bound to an address.
EACCESS	The requested address is protected, and the current user does not have permission to access it.
EFAULT	The *Address* parameter is not in a writable part of the *UserAddress* space.

Examples

1. The following program fragment illustrates the use of the **bind** subroutine to bind the name "/tmp/zan/" to a UNIX domain socket.

```
#include <sys/un.h>
.
.
.
struct sockaddr_un.addr
.
.
.
strcpy(addr.sun_path, "/tmp/zan/");
addr.sun_family = AF_UNIX;
bind(s,(struct sockaddr*)&addr, strlen(addr.sun_path)+
     sizeof(addr.sun_family));
```

2. The Reading UNIX Domain Datagrams Example Program fragrment illustrates the use of the **bind** subroutine.

Implementation Specifics

The **bind** subroutine is part of AIX Base Operating System (BOS) Runtime.

All applications containing the **bind** subroutine must be compiled with **_BSD** defined. In addition, when applicable, all socket applications must include the BSD library **libbsd**.

Files

/usr/include/netinet/in.h	Contains internet constants and structures definitions.
/usr/include/sys/socket.h	Contains socket definitions.
/usr/include/sys/socketvar.h	Defines the kernel structure per socket and contains buffer queues.
/usr/include/sys/types.h	Contains definitions of unsigned data types.

Related Information

Other socket creation and connection subroutines are the **connect** subroutine, **listen** subroutine, and **socket** subroutine.

The subroutine to retrieve the socket name is the **getsockname** subroutine.

Sockets Overview, Understanding Socket Connections, Binding Names to Sockets in *Communications Programming Concepts*.

connect Subroutine

Purpose

Connects two sockets.

Syntax

#include <sys/types.h>

#include <sys/socket.h>

#include <netinet/in.h>

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

Description

The **connect** subroutine requests a connection between two sockets. The kernel sets up the communications links between the sockets; both sockets must use the same address format and protocol.

If a **connect** subroutine is issued on an unbound socket, the system automatically binds the socket.

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

- If the initiating socket is **SOCK_DGRAM**, then the **connect** subroutine establishes the peer address. The peer address identifies the socket where all datagrams are sent on subsequent **send** subroutines. No connections are made by this **connect** subroutine.
- If the initiating socket is **SOCK_STREAM**, then the **connect** subroutine attempts to make a connection to the socket specified by the *Name* parameter. Each communication space interprets the *Name* parameter differently.

Parameters

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

Return Value

Upon successful completion, the **connect** subroutine returns a value of 0 (zero).

If the **connect** subroutine fails, the system handler performs the following functions:

- Returns a value of –1 (negative one) to the calling program
- Moves an error code, indicating the specific error, into the global variable **errno**

Error Codes

The **connect** subroutine fails if any one of the following errors occurs:

EBADF	The *Socket* parameter is not valid.
ENOTSOCK	The *Socket* parameter refers to a file, not a socket.
EADDRNOTAVAIL	The specified address is not available from the local machine.
EAFNOSUPPORT	The addresses in the specified address family cannot be used with this socket.
EISCONN	The socket is already connected.
ETIMEDOUT	The establishment of a connection timed out before a connection was made.
ECONNREFUSED	The attempt to connect was rejected.
ENETUNREACH	No route to the network or host is present.
EADDRINUSE	The specified address is already in use.
EFAULT	The *Address* parameter is not in a writable part of the user address space.
EWOULDBLOCK	The socket is marked *nonblocking*, the connection cannot be immediately completed. The application program can select the socket for writing during the connection process.

Examples

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

```
struct sockaddr_un server;
.
.
.
connect(s,(struct sockaddr*)&server, strlen(server.sun_path)+
        sizeof(server.sun_family));
```

2. The Initiating a UNIX Stream Connection program fragrment illustrates the use of the **connect** subroutine.

Implementation Specifics

The **connect** subroutine is part of AIX Base Operating System (BOS) Runtime.

All applications containing the **connect** subroutine must be compiled with **_BSD** defined. In addition, when applicable, all socket applications must include the BSD library **libbsd**.

Files

/usr/include/netinet/in.h	Contains internet constants and structures definitions.
/usr/include/sys/socket.h	Contains socket definitions.

/usr/include/sys/socketvar.h	Defines the kernel structure per socket and contains buffer queues.
/usr/include/sys/types.h	Contains definitions of unsigned data types.

Related Information

Other socket creation subroutines are the **accept** subroutine, **bind** subroutine, and **socket** subroutine.

Socket information retrieval and transmission subroutines are the **getsockname** subroutine, **select** subroutine, and **send** subroutine.

Sockets Overview, Understanding Socket Connections in *Communications Programming Concepts*.

dn_comp Subroutine

Purpose

Compresses a domain name.

Library

(libc.a)

Syntax

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

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

Description

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

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

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

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

Parameters

ExpDomNam	Specifies the address of an expanded domain name.
CompDomNam	Points to an array containing the compressed domain name.

Length	Specifies the size of the array pointed to by the *CompDomNam* parameter.
DomNamPtrs	Specifies a list of pointers to previously compressed names in the current message.
LastDomNamPtr	Points to the end of the array specified to by the *CompDomNam* parameter.

Return Value

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

If unsuccessful, the **dn_comp** subroutine returns a value of –1 (negative one) to the calling program.

Implementation Specifics

The **dn_comp** subroutine is part of AIX Base Operating System (BOS) Runtime.

All applications containing the **dn_comp** subroutine must be compiled with **_BSD** defined. In addition, when applicable, all socket applications must include the BSD library **libbsd**.

Files

/etc/resolv.conf	Defines name server and domain name structures, constants, and values.
/usr/include/netinet/in.h	Contains Internet constants and structures.
/usr/include/arpa/nameser.h	Defines Internet name server structures, constants, and values.
/usr/include/resolv.h	Contains global information used by the resolver subroutines.
/usr/include/sys/types.h	Contains definitions of unsigned data types.

Related Information

Domain name access subroutines are the **res_init** subroutine, **res_mkquery** subroutine, and **res_send** subroutine.

Domain name translation subroutines are the **dn_expand** subroutine, **dn_find** subroutine, and **dn_skipname** subroutine.

Byte stream and byte boundary retrieval subroutines are the **_getshort** subroutine, **_getlong** subroutine, **putshort** subroutine, and **putlong** subroutine.

The **named** daemon.

Sockets Overview, Understanding Domain Name Resolution in *Communications Programming Concepts.*

Understanding Naming for TCP/IP in *Communication Concepts and Procedures.*

dn_expand Subroutine

Purpose

Expands a compressed domain name.

Library

(libc.a)

Syntax

#include <sys/types.h>

#include <netinet/in.h>

#include <arpa/nameser.h>

#include <resolv.h>

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

Description

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

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

Parameters

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

Return Value

Upon successful completion, the **dn_expand** subroutine returns the size of the expanded domain name.

If unsuccessful, the **dn_expand** subroutine returns a value of –1 (negative one) to the calling program.

Implementation Specifics

The **dn_expand** subroutine is part of AIX Base Operating System (BOS) Runtime.

All applications containing the **dn_expand** subroutine must be compiled with **_bsd** defined. In addition, when applicable, all socket applications must include the BSD library **libbsd**.

Files

/etc/resolv.conf	Defines name server and domain name constants, structures, and values.
/usr/include/netinet/in.h	Defines Internet constants and structures.
/usr/include/arpa/nameser.h	Defines Internet name server constants, structures, and values.
/usr/include/sys/socketvar.h	Defines the kernel structure per socket and contains buffer queues.
/usr/include/sys/types.h	Contains definitions of unsigned data types.

Related Information

Domain name access subroutines are the **res_init** subroutine, **res_mkquery** subroutine, and **res_send** subroutine.

Domain name translation subroutines are the **dn_comp** subroutine, **dn_find** subroutine, and **dn_skipname** subroutine.

Byte stream and byte boundary retrieval subroutines are the **getshort** subroutine, **_getlong** subroutine, **putshort** subroutine, and **putlong** subroutine.

Sockets Overview, Understanding Domain Name Resolution in *Communications Programming Concepts.*

Understanding Naming for TCP/IP in *Communication Concepts and Procedures.*

dn_find Subroutine

Purpose

Searches for an expanded domain name.

Library

(libc.a)

Syntax

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

dn_find (*ExpandDomNam*, *Message*, *DomNamPtrs*, *LastDomNamPtr*)
char **ExpDomNam*, **Message*;
char ***DomNamPtrs*, ***LastDomNamPtr*;

Description

The **dn_find** (domain name find) subroutine searches for an expanded domain name from a list of previously compressed names. An expanded domain name is an one that is not compressed. If an expanded domain name is found, the **dn_comp** subroutine returns the offset from *Message*. An application program does not call the **dn_find** subroutine, directly. Instead, the **dn_find** subroutine is called indirectly by the **dn_comp** subroutine.

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

Parameters

ExpandDomNam	Points to an expanded domain name.
Message	Points to the address of a domain name message that contains the name sought by the **dn_find** operation.
DomNamPtrs	Specifies an array of pointers to previously compressed names in the current message.
LastDomNamPtr	Points to the end of an array of pointers. The array is indicated by the *DomNamPtrs* parameter.

Return Values

Upon successful completion, the **dn_find** subroutine returns the offset from the *Message* parameter.

If unsuccessful, the **dn_find** subroutine returns a value of –1 (negative one).

Implementation Specifics

The **dn_find** subroutine is part of AIX Base Operating System (BOS) Runtime.

All applications containing the **dn_find** subroutine must be compiled with **_BSD** defined. In addition, when applicable, all socket applications must include the BSD library **libbsd**.

Files

/etc/resolv.conf	Defines name server and domain name structures and constants.
/usr/include/netinet/in.h	Defines Internet constants and structures.
/usr/include/arpa/nameser.h	Defines Internet name server structures and constants.
/usr/include/resolv.h	Contains global information used by the resolver subroutines.
/usr/include/sys/socketvar.h	Defines the kernel structure per socket and contains buffer queues.
/usr/include/sys/types.h	Contains definitions of unsigned data types

Related Information

Domain name access subroutines are the **res_init** subroutine, **res_mkquery** subroutine, and **res_send** subroutine.

Domain name translation subroutines are the **dn_comp** subroutine, **dn_expand** subroutine, and **dn_skipname** subroutine.

Byte stream and byte boundary retrieval subroutines are the **_getshort** subroutine, **_getlong** subroutine, **putshort** subroutine, and **putlong** subroutine.

The **named** daemon.

Sockets Overview, Understanding Domain Name Resolution in *Communications Programming Concepts*

Understanding Naming for TCP/IP in *Communication Concepts and Procedures*

dn_skipname Subroutine

Purpose

Skips over a compressed domain name.

Library

(libc.a)

Syntax

#include <sys/types.h>

#include <netinet/in.h>

#include <arpa/nameser.h>

#include <resolv.h>

int dn_skipname (*CompDomNam*, *EndOfMessage*)
u_char **CompDomNam*;
u_char **EndOfMessage*;

Description

The **dn_skipname** subroutine skips over a compressed domain name.

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

Parameters

CompDomNam	Specifies a pointer to a compressed domain name.
EndOfMessage	Specifes a pointer to the end of the message string.

Return Value

Upon successful completion, the **dn_skipname** subroutine returns the size of *CompDomNam.*

If the **dn_skipname** subroutine fails, the subroutine returns a –1 (negative one).

Implementation Specifics

The **dn_skipname** subroutine is part of AIX Base Operating System (BOS) Runtime.

All applications containing the **dn_skipname** subroutine must be compiled with **_BSD** defined. In addition, when applicable, all socket applications must include the BSD library **libbsd**.

Files

/etc/resolv.conf	Defines name server and domain name structures, values, and constants.
/usr/include/resolv.h	Contains global information used by the resolver subroutines.

/usr/include/arpa/nameser.h	Defines Internet name server structures, values, and constants.
/usr/include/sys/types.h	Contains definitions of unsigned data types.
/usr/include/netinet/in.h	Defines Internet constants and structures.

Related Information

Domain name access subroutines are the **res_init** subroutine, **res_mkquery** subroutine, and **res_send** subroutine.

Domain name translation subroutines are the **dn_comp** subroutine, **dn_expand** subroutine, and **dn_find** subroutine.

Byte stream and byte boundary retrieval subroutines are the **_getshort** subroutine, **_getlong** subroutine, **putshort** subroutine, and **putlong** subroutine.

Sockets Overview, Understanding Domain Name Resolution in *Communications Programming Concepts.*

endhostent Subroutine

Purpose

Ends retrieval of network host entries.

Library

(libc.a)

Syntax

#include <netdb.h>

void endhostent ( )

Description

The **endhostent** subroutine closes the **/etc/hosts** file. The **/etc/hosts** file is opened by either the **gethostbyaddr** or **gethostbyname** subroutine.

Note: If a previous **sethostent** (STAYOPEN)**;** routine has been performed and the STAYOPEN value does not equal 0 (zero), then the **endhostent( );** routine will *not* close the **/etc/hosts** file. Also, the **sethostent( );** routine does not indicate that it closed the file. A second **sethostent (**STAYOPEN**);** routine has to be issued with the STAYOPEN value equal to 0 (zero) in order for a following **endhostent( );** routine to succeed. If this is not done, the **/etc/hosts** file closes on an **exit**( ); call.

Example

To close the **/etc/hosts** file:

```
endhostent();
```

Implementation Specifics

The **endhostent** subroutine is part of AIX Base Operating System (BOS) Runtime.

All applications containing the **endhostent** subroutine must be compiled with **_BSD** defined. In addition, when applicable, all socket applications must include the BSD library **libbsd**.

Files

/etc/hosts	Contains the host name database.
/usr/include/netdb.h	Contains the network database structures.

Related Information

Additional host information retrieval subroutines are the **gethostbyaddr** subroutine, **gethostbyname** subroutine, and **sethostent** subroutine.

Sockets Overview in *Communications Programming Concepts.*

endnetent Subroutine

Purpose

Closes the **networks** file.

Library

libc.a

Syntax

#include <netdb.h>

void endnetent ( )

Description

The **endnetent** (end network entry) subroutine closes the /**etc/networks** file. Calls made to the **getnetent, getnetbyaddr,** or **getnetbyname** subroutines open the /**etc/networks** file.

Note: If a previous **setnetent** (STAYOPEN) subroutine has been performed and the STAYOPEN value does not equal 0 (zero), then the **endnetent** subroutine will *not* close the /**etc/networks** file. Also, the **setnetent** subroutine does not indicate that it closed the file. A second **setnetent (**STAYOPEN**);** routine has to be issued with the STAYOPEN value equal to 0 (zero) in order for a following **endnetent** subroutine to succeed. If this is not done, the /**etc/networks** file closes on an **exit** call.

Example

To close the /**etc/networks** file:

```
endnetent();
```

Implementation Specifics

The **endnetent** subroutine is part of AIX Base Operating System (BOS) Runtime.

All applications containing the **endnetent** subroutine must be compiled with **_BSD** defined. In addition, when applicable, all socket applications must include the BSD library **libbsd**.

Files

/etc/networks	Contains official network names.
/usr/include/netdb.h	Contains the network database structures.

Related Information

Additional network information retrieval subroutines are the **getnetbyaddr** subroutine, **getnetbyname** subroutine, **getnetent** subroutine, and **setnetent** subroutine.

Sockets Overview, Understanding Network Address Translation in *Communications Programming Concepts.*

endprotoent Subroutine

Purpose

Closes the /**etc/protocols** file.

Library

(libc.a)

Syntax

void endprotoent ( )

Description

The **endprotoent** (end protocol entry) subroutine closes the /**etc/protocols** file.

Calls made to the **getprotoent** subroutine, **getprotobyname** subroutine, or **getprotobynumber** subroutine open the /**etc/protocols** file. An application program can use the **endprotoent** subroutine to close the /**etc/protocols** file.

Note: If a previous **setprotoent** (STAYOPEN**);** routine has been performed and the STAYOPEN value does not equal 0 (zero), then the **endprotoent;** routine will *not* close the /**etc/protocols** file. Also, the **setprotoent;** routine does not indicate that it closed the file. A second **setprotoent** (STAYOPEN**);** routine has to be issued with the STAYOPEN value equal to 0 (zero) in order for a following **endprotoent;** routine to succeed. If this is not done, the /**etc/protocols** file closes on an **exit**; call.

Example

To close the /**etc/protocols** file:

```
endprotoent();
```

Implementation Specifics

The **endprotoent** subroutine is part of AIX Base Operating System (BOS) Runtime.

All applications containing the **endprotoent** subroutine must be compiled with **_BSD** defined. In addition, when applicable, all socket applications must include the BSD library **libbsd**.

File

/**etc/protocols**	Contains protocol names.

Related Information

Additional protocol information retrieval subroutines are the **getprotobynumber** subroutine, **getprotobyname** subroutine, **getprotoent** subroutine, and **setprotoent** subroutine.

Sockets Overview, Understanding Network Address Translation in *Communications Programming Concepts*.

endservent Subroutine

Purpose

Closes the **/etc/service** file entry.

Library

(libc.a)

Syntax

#include <netdb.h>

void endservent ( )

Description

The **endservent** (end service entry) subroutine closes the **/etc/services** file. A call made to the **getservent** subroutine, **getservbyname** subroutine, or **getservbyport** subroutine opens the **/etc/services** file. An application program can use the **endservent** subroutine to close the **/etc/services** file.

Note: If a previous **setservent (STAYOPEN);** routine has been performed and the STAYOPEN value does not equal 0 (zero), then the **endservent( );** routine will *not* close the **/etc/services** file. Also, the **setservent( );** routine does not indicate that it closed the file. A second **setservent (STAYOPEN);** routine has to be issued with the STAYOPEN value equal to 0 (zero) in order for a following **endservent( );** routine to succeed. If this is not done, the **/etc/services** file closes on an **exit**( ); call.

Example

To close the **/etc/services** file:

```
endservent ();
```

Implementation Specifics

The **endservent** subroutine is part of AIX Base Operating System (BOS) Runtime.

All applications containing the **endservent** subroutine must be compiled with **_BSD** defined. In addition, when applicable, all socket applications must include the BSD library **libbsd**.

Files

/etc/services	Contains service names.
/usr/include/netdb.h	Contains network database structures.

Related Information

Additional service information retrieval subroutines are the **getservbyname** subroutine, **getservbyport** subroutine, **getservent** subroutine, and **setservent** subroutine.

Protocol information retrieval subroutines are the **endprotoent** subroutine, **getprotobynumber** subroutine, **getprotobyname** subroutine, **getprotoent** subroutine, and **setprotoent** subroutine.

Sockets Overview, Understanding Network Address Translation in *Communications Programming Concepts*.

getdomainname Subroutine

Purpose

Gets the name of the current domain.

Syntax

int getdomainname (*name*, *namelen* **)**

char **name*;
int *namelen*;

Description

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

The purpose of domains is to enable two distinct networks that may have host names in common to merge. Each network would be distinguished by having a different domain name. At the current time, only the NIS and the **sendmail** command make use of domains

Note: Domain names are restricted to 64 characters.

Parameters

name	Specifies the domain name to be returned.
namelen	Specifies the size of the array pointed to by the *name* parameter.

Return Values

If the call suceeds, a value of 0 (zero) is returned. If the call fails, a value of –1 is returned and an error code is placed in the global location errno.

Error Codes

The following error may be returned by this subroutine:

EFAULT	The *Name* parameter gave an invalid address.

Implementation Specifics

The **getdomainname** subroutine is part of AIX Base Operating System (BOS) Runtime.

All applications containing the **getdomainname** subroutine must be compiled with **_BSD** defined. In addition, when applicable, all socket applications must include the BSD library **libbsd**.

Related Information

The **gethostname** subroutine, **setdomainname** subroutine, **sethostname** subroutine.

Sockets Overview in *Communications Programming Concepts.*

gethostbyaddr Subroutine

Purpose

Gets network host entry by address.

Library

(libc.a)

Syntax

#include<netdb.h>

struct hostent *gethostbyaddr (*Address*, *Length*, *Type*)
char **Address*;
int *Length*, *Type*;

Description

The **gethostbyaddr** subroutine retrieves information about a host using the host address as a search key.

The **gethostbyaddr** subroutine recognizes domain name servers as described in RFC883. If the file /**etc/resolv.conf** exists, the **gethostbyaddr** subroutine queries the domain name server. If the request to the domain name server times out, the **gethostbyaddr** subroutine checks the local /**etc/hosts** file.

The **gethostbyaddr** returns a pointer to a **hostent** structure, which contains information obtained from the domain name server or which contains a field from a line in the /**etc/hosts** file. The **hostent** structure is defined in the **netdb.h** header file.

Parameters

Address	Specifies a host address. The host address is passed as a character string and is assumed to be in dotted IP format.
Length	Specifies the length of host address.
Type	Specifies the domain type of the host address. This currently works only on address family: AF_INET.

Return Values

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

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

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

Error Codes

The **gethostbyaddr** subroutine fails if any one of the following errors occurs:

HOST_NOT_FOUND	The host specified by the *Name* parameter is not found.
TRY_AGAIN	The local server does not receive a response from an authoritative server. Try again later.

NO_RECOVERY	This error code indicates an unrecoverable error.
NO_ADDRESS	The requested *Name* parameter is valid but does not have an Internet address at the name server.

Implementation Specifics

The **gethostbyaddr** subroutine is part of AIX Base Operating System (BOS) Runtime.

All applications containing the **gethostbyaddr** subroutine must be compiled with **_BSD** defined. In addition, when applicable, all socket applications must include the BSD library **libbsd**.

Files

/etc/hosts	Contains the host name data base.
/usr/include/netdb.h	Contains the network data base structures.
/etc/resolv.conf	Contains the name server and domain name information.

Related Information

Additional host information retrieval socket subroutines are the **endhostent** subroutine, **gethostbyname** subroutine, and **sethostent** subroutine.

Sockets Overview, Understanding Network Address Translation in *Communications Programming Concepts.*

gethostbyname Subroutine

Purpose

Gets network host entry by name.

Library

(libc.a)

Syntax

#include <netdb.h>

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

Description

The **gethostbyname** subroutine retrieves host address and name information using a host name as a search key

The **gethostbyname** subroutine recognizes domain name servers as described in RFC883. If the file **/etc/resolv.conf** exists, the **gethostbyname** subroutine queries the domain name server. If the request to the domain name server times out, the **gethostbyname** subroutine checks local **/etc/hosts** file.

The **gethostbyname** subroutine returns a pointer to a **hostent** structure, which contains information obtained from a name server program or contains a field from a line in the **/etc/hosts** file. The **hostent** structure is defined in the **netdb.h** header file.

Use the **endhostent** subroutine to close the **/etc/hosts/** file or the TCP connection.

Parameter

Name	Points to the host name.

Return Values

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

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

If an error occurs or if the end of the file is reached, the **gethostbyname** subroutine returns a **NULL** pointer and sets **h_ernno** variable to indicate the error.

Error Codes

The **gethostbyname** subroutine fails if any one of the following errors occur:

HOST_NOT_FOUND	The host specified by the *Name* parameter was not found.
TRY_AGAIN	The local server did not receive a response from an authoritative server. Try again later.
NO_RECOVERY	This error code indicates an unrecoverable error.
NO_ADDRESS	The requested *Name* is valid but does not have an Internet address at the name server.

Example

1. The following program fragment illustrates the use of the **gethostbyname** subroutine to look up a destination host.

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

Implementation Specifics

The **gethostbyname** subroutine is part of AIX Base Operating System (BOS) Runtime.

All applications containing the **gethostbyname** subroutine must be compiled with **_BSD** defined. In addition, when applicable, all socket applications must include the BSD library **libbsd**.

Files

/etc/hosts	Contains the host name data base.
/etc/resolv.conf	Contains the name server and domain name.
/usr/include/netdb.h	Contains the network data base structures.

Related Information

Additional host information retrieval routines are the **gethostent** subroutine, **endhostent** subroutine, **gethostbyaddr** subroutine, and **sethostent** subroutine.

Sockets Overview in *Communications Programming Concepts.*

gethostent Subroutine

Purpose

Retrieves a network host entry.

Library

(libc.a)

Syntax

#include <netdb.h>

gethostent ( )

struct hostent *gethostent ()

Description

The **gethostent** subroutine allows an application program to retrieve an entry from the **/etc/host** file. The **gethostent** subroutine opens the **/etc/host** file and performs a sequential read of each line in the file starting from the beginning of the file. Each subsequent **gethostent** subroutine call returns information for a different host.

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

Use the **endhostent** subroutine to close the **/etc/hosts** file.

Return Values

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

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

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

Implementation Specifics

The **gethostent** subroutine is part of AIX Base Operating System (BOS) Runtime.

All applications containing the **gethostent** subroutine must be compiled with **_BSD** defined. In addition, when applicable, all socket applications must include the BSD library **libbsd**.

Files

/etc/hosts	Contains the host name database.
/usr/include/netdb.h	Contains the network database structures.

Related Information

Additional host information retrieval subroutines are the **gethostbyaddr** subroutine, **gethostbyname** subroutine, and **sethostent** subroutine.

Sockets Overview in *Communications Programming Concepts.*

gethostid Subroutine

Purpose

Gets the unique identifier of the current host.

Syntax

int gethostid ( )

Description

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

Return Value

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

If the **gethostid** subroutine fails, the system handler performs the following functions:

- Returns a value of –1 (negative one) to the calling program
- Moves an error code, indicating the specific error, into the global variable **errno**

Implementation Specifics

The **gethostid** subroutine is part of AIX Base Operating System (BOS) Runtime.

All applications containing the **gethostid** subroutine must be compiled with **_BSD** defined. In addition, when applicable, all socket applications must include the BSD library **libbsd**.

Related Information

Socket subroutines to set and obtain host names and to set host IDs, respectively, are the **gethostname** subroutine, **sethostname** subroutine, and **sethostid** subroutine.

Sockets Overview in *Communications Programming Concepts.*

gethostname Subroutine

Purpose

Gets the name of the local host.

Syntax

int gethostname (*Name*, *NameLength*)
char **Name*;
int *NameLength*;

Description

The **gethostname** subroutine retrieves the standard host name of the local host. If sufficient space is provided, the returned *Name* parameter is null–terminated. System host names are limited to **MAXHOSTNAMELEN** as defined in **/usr/include/sys/param.h**. The **MAXHOSTNAMELEN** value is set at 32.

The **gethostname** subroutine allows a calling process to determine the internal host name for a machine on a network.

Parameters

Name	Returns the address of an array of bytes where the host name is stored.
NameLength	Returns an integer that specifies the length of the *Name* array.

Return Values

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

If the **gethostname** subroutine fails, the subroutine handler performs the following functions:

- Returns a value of –1 (negative one) to the calling program
- Moves an error code, indicating the specific error, into the global variable **errno**

Error Codes

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

EFAULT	The *Name* parameter or *NameLength* parameter gives an invalid address.

Implementation Specifics

The **gethostname** subroutine is part of AIX Base Operating System (BOS) Runtime.

All applications containing the **gethostname** subroutine must be compiled with **_BSD** defined. In addition, when applicable, all socket applications must include the BSD library **libbsd**.

Related Information

Socket subroutines to obtain and set the host ID are the **gethostid** subroutine and **sethostid** subroutine.

The socket subroutine to set the host name is the **sethostname** subroutine.

Sockets Overview in *Communications Programming Concepts*.

_getlong Subroutine

Purpose

Retrieves long byte quantities.

Library

(libc.a)

Syntax

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

unsigned long _getlong (*MessagePtr*)
u_char **MessagePtr*;

Description

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

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

Parameters

MessagePtr	Specifies a pointer into the byte stream.

Return Value

The **_getlong** subroutine returns an unsigned long (32–bit) value.

Implementation Specifics

The **_getlong** subroutine is part of AIX Base Operating System (BOS) Runtime.

All applications containing the **_getlong** subroutine must be compiled with **_BSD** defined. In addition, when applicable, all socket applications must include the BSD library **libbsd**.

Files

/etc/resolv.conf	Lists name server and domain names.
/usr/include/resolv.h	Contains global information used by the resolver subroutines.
/usr/include/arpa/nameser.h	Defines Internet name server structures and constants.
/usr/include/sys/types.h	Contains definitions of unsigned data types.
/usr/include/netinet/in.h	Contains Internet constants and structures.

Related Information

Domain name access subroutines are the **res_init** subroutine, **res_mkquery** subroutine, and **res_send** subroutine.

Domain name translation subroutines are the **dn_comp** subroutine, **dn_expand** subroutine, **dn_find** subroutine,and **dn_skipname** subroutine.

Byte stream and boundary retrieval subroutines are the **_getshort** subroutine, **putshort** subroutine, and **putlong** subroutine.

Sockets Overview, Understanding Domain Name Resolution in *Communications Programming Concepts*.

getnetbyaddr Subroutine

Purpose

Gets network entry by address.

Library

(libc.a)

Syntax

#include <netdb.h>

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

Description

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

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

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

Parameters

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

Return Values

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

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

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

Implementation Specifics

The **getnetbyaddr** subroutine is part of AIX Base Operating System (BOS) Runtime.

All applications containing the **getnetbyaddr** subroutine must be compiled with **_BSD** defined. In addition, when applicable, all socket applications must include the BSD library **libbsd**.

Files

/etc/networks	Contains official network names.
/usr/include/netdb.h	Contains the network database structures.

Related Information

Additional network information retrieval subroutines are the **endnetent** subroutine, **getnetbyname** subroutine, **getnetent** subroutine, and **setnetent** subroutine.

Sockets Overview in *Communications Programming Concepts*.

getnetbyname Subroutine

Purpose

Gets network entry by name.

Library

(libc.a)

Syntax

#include <netdb.h>

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

Description

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

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

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

Parameter

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

Return Values

Upon successful completion, the **getnetbyname** subroutine returns a pointer to a **netent** structure.

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

If an error occurs or the end of the file is reached, the **getnetbyname** subroutine returns a **NULL** pointer.

Implementation Specifics

The **getnetbyname** subroutine is part of AIX Base Operating System (BOS) Runtime.

All applications containing the **getnetbyname** subroutine must be compiled with **_BSD** defined. In addition, when applicable, all socket applications must include the BSD library **libbsd**.

Files

/etc/networks	Contains official network names.
/usr/include/netdb.h	Contains the network database structures.

Related Information

Additional network information retrieval subroutines are the **endnetent** subroutine, **getnetbyaddr** subroutine, **getnetent** subroutine, and **setnetent** subroutine.

Sockets Overview in *Communications Programming Concepts*.

getnetent Subroutine

Purpose

Gets network entry.

Library

(libc.a)

Syntax

#include <netdb.h>

struct netent *getnetent ( )

Description

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

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

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

Return Values

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

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

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

Implementation Specifics

The **getnetent** subroutine is part of AIX Base Operating System (BOS) Runtime.

All applications containing the **getnetent** subroutine must be compiled with **_BSD** defined. In addition, when applicable, all socket applications must include the BSD library **libbsd**.

Files

/etc/networks	Contains official network names.
/usr/include/netdb.h	Contains the network database structures.

Related Information

Additional network information retrieval subroutines are the **endnetent** subroutine, **getnetbyaddr** subroutine, **getnetbyname** subroutine, and **setnetent** subroutine.

Sockets Overview in *Communications Programming Concepts.*

getpeername Subroutine

Purpose

Gets the name of the peer socket.

Syntax

#include<sys/types.h>

#include <sys/socket.h>

int getpeername (*Socket*, *Name*, *NameLength*);
int *Socket*;
struct sockaddr **Name*;
int **NameLength*;

Description

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

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

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

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

Parameters

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

Return Value

Upon successful completion, a value of 0 (zero) is returned and the *Name* parameter holds the address of the peer socket.

If the **getpeername** subroutine fails, the system handler performs the following functions:

- Returns a value of –1 (negative one) to the calling program
- Moves an error code, indicating the specific error, into the global variable **errno**

Error Codes

The **getpeername** subroutine fails if any one of the following errors occur:

EBADF The *Socket* parameter is not valid.

ENOTSOCK The *Socket* parameter refers to a file, not a socket.

ENOTCONN The socket is not connected.

ENOBUFS Insufficient resources were available in the system to complete the call.

EFAULT The *Address* parameter is not in a writable part of the user address space.

Examples

1. The following program fragment illustrates the use of the **getpeername** subroutine to return the address of the peer connected on the other end of the socket.

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

Implementation Specifics

The **getpeername** subroutine is part of AIX Base Operating System (BOS) Runtime.

All applications containing the **getpeername** subroutine must be compiled with **_BSD** defined. In addition, when applicable, all socket applications must include the BSD library **libbsd**.

Files

/usr/include/sys/socket.h Contains socket definitions.

/usr/include/sys/socketvar.h Defines the kernel structure per socket and contains buffer queues.

/usr/include/sys/types.h Contains definitions of unsigned data types.

Related Information

The socket subroutines to create and name sockets are the **accept** subroutine, **bind** subroutine, and **socket** subroutine.

The socket subroutine used to retrieve a local socket address is the **getsockname** subroutine.

Sockets Overview in *Communications Programming Concepts*.

getprotobyname Subroutine

Purpose

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

Library

(libc.a)

Syntax

#include <netdb.h>

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

Description

The **getprotobyname** (get protocol by name) subroutine retrieves protocol information from the **/etc/protocols** file by protocol name.

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

The **getprotobyname** subroutine searches the protocols file sequentially from the start of the file until it finds a matching protocol name or until it reaches the end of the file.

The subroutine returns a pointer to a **protoent** structure, which contains fields for a line of information in the **protocols** file. The **netdb.h** header file defines the **protoent** structure.

Use the **endprotoent** subroutine to close the protocols file.

Parameters

Name	Specifies the protocol name.

Return Values

Upon successful completion, the **getprotobyname** subroutine returns a pointer to a **protoent** structure.

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

If an error occurs or the end of the file is reached, the **getprotbyname** subroutine returns a **NULL** pointer.

Implementation Specifics

The **getprotobyname** subroutine is part of AIX Base Operating System (BOS) Runtime.

All applications containing the **getprotobyname** subroutine must be compiled with **_BSD** defined. In addition, when applicable, all socket applications must include the BSD library **libbsd**.

Files

/etc/protocols	Contains protocol information.
/usr/include/netdb.h	Contains the network database structures.

Related Information

Additional protocol information retrieval subroutines are the **endprotoent** subroutine, **getprotobynumber** subroutine, **getprotoent** subroutine, and **setprotoent** subroutine.

Sockets Overview in *Communications Programming Concepts*.

getprotobynumber Subroutine

Purpose

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

Library

(libc.a)

Syntax

#include <netdb.h>

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

Description

The **getprotobynumber** (get protocol by number) subroutine retrieves protocol information from the **/etc/protocols** file using a specified protocol number as a search key.

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

The **getprotobynumber** subroutine searches the **/etc/protocols** file sequentially from the start of the file until it finds a matching protocol name or protocol number, or until it reaches the end of the file.

The subroutine returns a pointer to a **protoent** structure, which contains fields for a line of information in the **/etc/protocols** file. The **netdb.h** file defines the **protoent** structure.

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

Parameter

Protocol Specifies the protocol number.

Return Values

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

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

If an error occurs or the end of the file is reached, the **getprotobynumber** subroutine returns a **NULL** pointer.

Implementation Specifics

The **getprotobynumber** subroutine is part of AIX Base Operating System (BOS) Runtime.

All applications containing the **getprotobynumber** subroutine must be compiled with **_BSD** defined. In addition, when applicable, all socket applications must include the BSD library **libbsd**.

Files

/etc/protocols	Contains protocol information.
/usr/include/netdb.h	Contains network database structures.

Related Information

Additional protocol information retrieval subroutines are the **endprotoent** subroutine, **getprotobyname** subroutine, **getprotoent** subroutine, and **setprotoent** subroutine.

Sockets Overview in *Communications Programming Concepts.*

getprotoent Subroutine

Purpose

Gets protocol entry from the /etc/**protocols** file.

Library

(libc.a)

Syntax

#include <netdb.h>

struct protoent *getprotoent ( )

Description

The **getprotoent** (get protocol entry) subroutine retrieves protocol information from the /etc/**protocols** file.

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

The **getprotoent** subroutine opens and performs a sequential read of the /etc/**protocols** file.

The **getprotoent** subroutine returns a pointer to a **protoent** structure, which contains the fields for a line of information in the /etc/**protocols** file. The **netdb.h** header file defines the **protoent** structure.

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

Return Values

Upon successful completion, the **getprotoent** subroutine returns a pointer to a **protoent** structure.

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

If an error occurs or the end of the file is reached, the **getprotoent** subroutine returns a **NULL** pointer.

Implementation Specifics

The **getprotoent** subroutine is part of AIX Base Operating System (BOS) Runtime.

All applications containing the **getprotoent** subroutine must be compiled with **_BSD** defined. In addition, when applicable, all socket applications must include the BSD library **libbsd**.

Files

/etc/protocols	Contains protocol information.
/usr/include/netdb.h	Contains the network database structures.

Related Information

Additional protocol information retrieval subroutines are the **endprotoent** subroutine, **getprotobyname** subroutine, **getprotobynumber** subroutine, and **setprotoent** subroutine.

Sockets Overview in *Communications Programming Concepts*.

getservbyname Subroutine

Purpose

Gets service entry by name.

Library

(libc.a)

Syntax

#include <netdb.h>

struct servent *getservbyname (*Name*, *Protocol*)
char **Name*, **Protocol*;

Description

The **getservbyname** (get service by name) subroutine retrieves an entry from the **/etc/services** file using the service name as a search key.

An application program can use the **getservbyname** subroutine to access a service, service aliases, the protocol for the service, and a protocol port number for the service.

The **getservbyname** subroutine searches the **/etc/services** file sequentially from the start of the file until it finds one of the following:

- a matching name and protocol number
- a matching name when the *Protocol* parameter is set to 0 (zero)
- the end of the file

Upon locating a matching name and protocol, the **getservbyname** returns a pointer to the **servent** structure, which contains fields for a line of information from the **/etc/services** file. The **netdb.h** header file defines the **servent** structure and structure fields.

Use the **endservent** subroutine to close the **/etc/host** file.

Parameters

Name	Specifies the name of a service.
Protocol	Specifies a protocol for use with the specified service.

Return Value

The **getservbyname** subroutine returns a pointer to a **servent** structure when a successful match occurs.

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

If an error occurs or the end of the file is reached, the **getservbyname** subroutine returns a **NULL** pointer.

Implementation Specifics

The **getservbyname** subroutine is part of AIX Base Operating System (BOS) Runtime.

All applications containing the **getservbyname** subroutine must be compiled with **_BSD** defined. In addition, when applicable, all socket applications must include the BSD library **libbsd**.

Files

/etc/services	Contains service names.
/usr/include/netdb.h	Contains network database structures.

Related Information

Additional service information retrieval subroutines are the **endservent** subroutine, **getservbyport** subroutine, **getservent** subroutine, and **setservent** subroutine.

Protocol information retrieval subroutines are the **endprotoent** subroutine, **getprotobynumber** subroutine, **getprotobyname** subroutine, **getprotoent** subroutine, and **setprotoent** subroutine.

Sockets Overview, Understanding Network Address Translation in *Communications Programming Concepts.*

getservbyport Subroutine

Purpose

Gets service entry by port.

Library

(libc.a)

Syntax

#include <netdb.h>

struct servent *getservbyport (*Port*, *Protocol*)
int *Port*;
char **Protocol*;

Description

The **getservbyport** (get service by port) subroutine retrieves an entry from the **/etc/services** file using a port number as a search key.

An application program can use the **getservbyport** subroutine to access a service, service aliases, the protocol for the service, and a protocol port number for the service.

The **getservbyport** subroutine searches the services file sequentially from the beginning of the file until it finds one of the following:

- a matching protocol and port number
- a matching protocol when the *Port* parameter value equals 0 (zero)
- the end of the file

Upon locating a matching protocol and port number or upon locating a matching protocol only if the *Port* parameter value equals 0 (zero), the **getservbyport** subroutine returns a pointer to a **servent** structure, which contains fields for a line of information in the **/etc/services** file. The **netdb.h** header file defines the **servent** structure and structure fields.

Use the **endservent** subroutine to close the **/etc/services** file.

Parameters

Port	Specifies the port where a service resides.
Protocol	Specifies a protocol for use with the service.

Return Value

Upon successful completion, the **getservbyport** subroutine returns a pointer to a **servent** structure.

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

if an error occurs or the end of the file is reached, the **getservbyport** subroutine returns a **NULL** pointer.

Implementation Specifics

The **getservbyport** subroutine is part of AIX Base Operating System (BOS) Runtime.

All applications containing the **getservbyport** subroutine must be compiled with **_BSD** defined. In addition, when applicable, all socket applications must include the BSD library **libbsd**.

Files

/etc/services	Contains service names.
/usr/include/netdb.h	Contains network database structures.

Related Information

Additional service information retrieval subroutines are the **endservent** subroutine, **getservbyname** subroutine, **getservent** subroutine, and **setservent** subroutine.

Protocol information retrieval subroutines are the **endprotoent** subroutine, **getprotobynumber** subroutine, **getprotobyname** subroutine, **getprotoent** subroutine, and **setprotoent** subroutine.

Sockets Overview in *Communications Programming Concepts.*

getservent Subroutine

Purpose

Gets services file entry.

Library

(libc.a)

Syntax

#include <netdb.h>

struct servent *getservent ( )

Description

The **getservent** (get service entry) subroutine opens and reads the next line of the **/etc/services** file.

An application program can use the **getservent** subroutine to retrieve information about network services and the protocol ports they use.

The **getservent** subroutine returns a pointer to a **servent** structure, which contains fields for a line of information from the **/etc/services** file. The **servent** structure is defined in the **netdb.h** header file.

The **/etc/services** file remains open after a call by the **getservent** subroutine. To close the **/etc/services** file after each call, use the **setservent** subroutine. Otherwise, use the **endservent** subroutine to close the **/etc/services** file.

Return Value

The **getservent** subroutine returns a pointer to a **servent** structure when a successful match occurs.

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

If an error occurs or the end of the file is reached, the **getservent** subroutine returns a **NULL** pointer.

Implementation Specifics

The **getservent** subroutine is part of AIX Base Operating System (BOS) Runtime.

All applications containing the **getservent** subroutine must be compiled with **_BSD** defined. In addition, when applicable, all socket applications must include the BSD library **libbsd**.

Files

/etc/services	Contains service names.
/usr/include/netdb.h	Contains network database structures.

Related Information

Additional service information retrieval subroutines are the **endservent** subroutine, **getservbyname** subroutine, **getservbyport** subroutine, and **setservent** subroutine.

Protocol information retrieval subroutines are the **endprotoent** subroutine, **getprotobynumber** subroutine, **getprotobyname** subroutine, **getprotoent** subroutine, and **setprotoent** subroutine.

Sockets Overview, Understanding Network Address Translation in *Communications Programming Concepts.*

_getshort Subroutine

Purpose

Retrieves short byte quantities.

Library

(libc.a)

Syntax

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

unsigned short getshort (*MessagePtr*)
u_char **MessagePtr*;

Description

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

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

Parameters

MessagePtr	Specifies a pointer into the byte stream.

Return Value

The **_getshort** subroutine returns an unsigned short (16–bit) value.

Implementation Specifics

The **_getshort** subroutine is part of AIX Base Operating System (BOS) Runtime.

All applications containing the **_getshort** subroutine must be compiled with **_BSD** defined. In addition, when applicable, all socket applications must include the BSD library **libbsd**.

Files

/etc/resolv.conf	Defines name server and domain names.
/usr/include/resolv.h	Contains global information used by the resolver subroutines.
/usr/include/arpa/nameser.h	Defines Internet name server structures, constants, and values.
/usr/include/sys/types.h	Defines unsigned data types.
/usr/include/netinet/in.h	Defines Internet constants and structures.

Related Information

Domain name access subroutines are the **res_init** subroutine, **res_mkquery** subroutine, and **res_send** subroutine.

Domain name translation subroutines are the **dn_comp** subroutine, **dn_expand** subroutine, **dn_find** subroutine, and **dn_skipname** subroutine.

Byte stream and byte boundary retrieval subroutines are the **_getlong** subroutine, **putshort** subroutine, and **putlong** subroutine.

getsockname Subroutine

Purpose

Gets the socket name.

Syntax

#include<sys/types.h>

#include <sys/socket.h>

int getsockname(*Socket*, *Name*, *NameLength*)

int *Socket*;

struct sockaddr **Name*;

int **NameLength*;

Description

The **getsockname** subroutine retrieves the locally bound address of the specified socket. The socket address represents a port number in the Internet domain and is stored in the **sockaddr** structure pointed to by the *Name* parameter. The **/sys/socket.h file** defines the **sockaddr** data structure.

Note: The **getsockname** subroutine does not perform operations on UNIX domain sockets.

A process created by another process can inherit open sockets. To use the inherited socket, the created process needs to identify their addresses. The **getsockname** subroutine allows a process to retrieve the local address bound to the specified socket.

A process can use the **getpeername** subroutine to determine the address of a destination socket in a socket connection.

Parameters

Socket	Specifies the socket for which the local address is desired.
Name	Points to the structure containing the local address of the specified socket.
NameLength	Specifies the size of the local address in bytes. Initialize the value pointed to by the *NameLength* parameter to indicate the amount of space pointed to by the *Name* parameter.

Return Value

Upon successful completion, a value of 0 (zero) is returned, and the *NameLength* parameter points to the size of the socket address.

If the **getsockname** subroutine fails, the subroutine handler performs the following functions:

- Returns a value of –1 (negative one) to the calling program
- Moves an error code, indicating the specific error, into the global variable **errno**.

Error Codes

The **getsockname** subroutine fails if any one of the following errors occur:

EBADF	The *Socket* parameter is not valid.
ENOTSOCK	The *Socket* parameter refers to a file, not a socket.
ENOBUFS	Insufficient resources are available in the system to complete the call.
EFAULT	The *Address* parameter is not in a writable part of the user address space.

Examples

1. The Reading Internet Domain Datagrams program fragment illustrates the use of the **getsockname** subroutine.
2. The Check for Pending Connections program fragrment illustrates the use of the **getsockname** subroutine.

Implementation Specifics

The **getsockname** subroutine is part of AIX Base Operating System (BOS) Runtime.

All applications containing the **getsockname** subroutine must be compiled with **_BSD** defined. In addition, when applicable, all socket applications must include the BSD library **libbsd**.

Files

/usr/include/sys/socket.h	Contains socket definitions.
/usr/include/sys/socketvar.h	Defines the kernel structure per socket and contains buffer queues.
/usr/include/sys/types.h	Contains definitions of unsigned data types.

Related Information

Socket subroutines to name and create sockets are the **accept** subroutine, the **bind** subroutine, and **socket** subroutine.

The socket subroutine to get the destination address of a connected socket is the **getpeername** subroutine.

Sockets Overview in *Communications Programming Concepts*.

getsockopt Subroutine

Purpose

Gets options on sockets.

Syntax

#include <sys/types.h>

#include <sys/socket.h>

int getsockopt (*Socket*, *Level*, *OptionName*, *OptionValue*, *OptionLength*)
int *Socket*, *Level*, *OptionName*;
char **OptionValue*;
int **OptionLength*;

Description

The **getsockopt** subroutine allows an application program to query socket options. The calling program specifies the name of the socket, the name of the option, and a place to store the requested information. The operating system gets the socket option information from its internal data structures and passes the requested information back to the calling program.

Options may exist at multiple protocol levels. They are always present at the uppermost socket level. When retrieving socket options, specify the level at which the option resides and the name of the option.

Parameters

Socket
Specifies the unique socket name.

Level
Specifies the protocol level at which the option resides. To retrieve options at the:

- Socket level—specify the *Level* parameter as **SOL_SOCKET**.
- Other levels—supply the appropriate protocol number for the protocol controlling the option. For example, to indicate that an option will be interpreted by the TCP protocol, set *Level* to the protocol number of TCP, as defined in the **netinet/in.h** header file.

OptionName
Specifies a single option. The *OptionName* parameter and any specified options are passed uninterpreted to the appropriate protocol module for interpretation. The **sys/socket.h** header file contains definitions for socket level options. The socket level options can be enabled or disabled; they operate in a toggle fashion. The **getsockopt** subroutine retrieves information about the following options:

- **SO_DEBUG** — Specifies the recording of debugging information. This option enables or disables debugging in the underlying protocol modules.
- **SO_ACCEPTCONN** — Socket had a listen call.

- **SO_BROADCAST** Specifies whether transmission of broadcast messages is supported. The option enables or disables broadcast support.
- **SO_REUSEADDR** Specifies that the rules used in validating addresses supplied by a **bind** subroutine should allow reuse of local addresses. This option enables or disables reuse of local addresses.
- **SO_KEEPALIVE** Keeps connections active. Enables or diables the periodic transmission of messages on a connected socket. If the connected socket fails to respond to these messages, the connection is broken and processes using that socket are notified with a **SIGPIPE** signal.
- **SO_DONTROUTE** Does not apply routing on outgoing messages. Indicates outgoing messages should bypass the standard routing facilities. Directs messages to the appropriate network interface according to the network portion of the destination address. This option enables or disables routing of outgoing messages.
- **SO_LINGER** Lingers on a **close** subroutine if data is present. This option controls the action taken when unsent messages queue on a socket and a **close** subroutine is performed. It uses a **struct** *Linger* parameter defined in the **sys/socket.h** file. The parameter specifies the state of the option and linger interval. Specify the linger interval by using the **setsockopt** subroutine when requesting **SO_LINGER**. This option enables or disable lingers on a **close** subroutine.

  If **SO_LINGER** is set, the system blocks the process during the **close** subroutine until it can transmit the data or until the time expires. If **SO_LINGER** is not specified, and a **close** subroutine is issued, the system handles the call in a way that allows the process to continue as quickly as possible.
- **SO_OOBINLINE** Leaves received out–of–band data (data marked urgent) in line. This option enables or disables the receipt of out–of–band data.
- **SO_SNDBUF** Retrieves buffer size information.
- **SO_RCVBUF** Retrieves buffer size information.
- **SO_SNDLOWAT** Retrieves low–water mark information.
- **SO_RCVLOWAT** Retrieves low–water mark information.
- **SO_SNDTIMEO** Retrieves time–out information.

- **SO_RCVTIMEO** Retrieves time–out information.
- **SO_ERROR** Retrieves information about error status and clear
- **SO_TYPE** Retrieves information about a socket type.

OptionValue

Specifies a pointer to the address of a buffer. The *OptionValue* parameter takes an integer parameter. The *OptionValue* parameter should be set to a nonzero value to enable a Boolean option or to a value of 0 (zero) to disable the option. The following options enable and disable in the same manner:

- **SO_DEBUG**
- **SO_REUSEADDR**
- **SO_KEEPALIVE**
- **SO_DONTROUTE**
- **SO_BROADCAST**
- **SO_OOBINLINE**

OptionLength Specifies the length of the *OptionValue*. The *OptionLength* parameter initially contains the size of the buffer pointed to by the *OptionValue* parameter. On return, the *OptionLength* parameter is modified to indicate the actual size of the value returned. If no option value is supplied or returned, the *OptionValue* parameter can be 0 (zero).

Options at other protocol levels vary in format and name.

Return Value

Upon successful completion, the **getsockopt** subroutine returns a value of 0 (zero).

If the **getsockopt** subroutine fails, the subroutine handler performs the following actions:

- Returns a value of –1 (negative one) to the calling program
- Moves an error code, indicating the specific error, into the global variable **errno**

Error Codes

The **getsockopt** subroutine fails if any one of the following errors occur:

EBADF	The *Socket* parameter is not valid.
ENOTSOCK	The *Socket* parameter refers to a file, not a socket.
ENOPROTOOPT	The option is unknown.
EFAULT	The address pointed to by the *OptionValue* parameter is not in a valid (writable) part of the process space, or the *OptionLength* parameter is not in a valid part of the process address space.

Example

1. The following program fragment illustrates the use of the **getsockopt** subroutine to determine an existing socket type.

```
#include <sys/types.h>
#include <sys/socket.h>
int type, size;
size = sizeof(int);
if(getsockopt(s, SOL_SOCKET, SO_TYPE, (char*)&type,&size)<0){
.
.
.
}
```

Implementation Specifics

The **getsockopt** subroutine is part of AIX Base Operating System (BOS) Runtime.

All applications containing the **getsockopt** subroutine must be compiled with **_BSD** defined. In addition, when applicable, all socket applications must include the BSD library **libbsd**.

Files

/usr/include/sys/socket.h	Contains socket definitions.
/usr/include/sys/socketvar.h	Defines the kernel structure per socket and contains buffer queues.
/usr/include/sys/types.h	Contains definitions of unsigned data types.

Related Information

Socket subroutines to manipulate protocol information are the **endprotoent** subroutine, **getprotobynumber** subroutine, **getprotoent** subroutine, **setprotoent** subroutine, and **socket** subroutine.

The socket subroutine to set socket options is the **setsockopt** subroutine.

Socket subroutines to assign names to sockets and end communications, respectively, are the **bind** subroutine and **close** subroutine.

Sockets Overview, Understanding Socket Options in *Communications Programming Concepts*.

htonl Subroutine

Purpose

Converts an unsigned long integer from host byte order to Internet network byte order.

Syntax

#include <sys/types.h>
#include <netinet/in.h>

unsigned long htonl (*HostLong*)
unsigned long *HostLong*;

Description

The **htonl** (host to network long) subroutine converts an unsigned long (32–bit) integer from host byte order to Internet network byte order.

The Internet network requires addresses and ports in network standard byte order. Use the **htonl** subroutine to convert the host integer representation of addresses and ports to Internet network byte order.

The **htonl** subroutine is defined in the **netinet/in.h** file as a macro.

Parameter

HostLong	Specifies a 32–bit integer in host byte order.

Return Values

The **htonl** subroutine returns a 32–bit integer in Internet network byte order (most significant byte first).

Implementation Specifics

The **htonl** subroutine is part of AIX Base Operating System (BOS) Runtime.

All applications containing the **htonl** subroutine must be compiled with **_BSD** defined. In addition, when applicable, all socket applications must include the BSD library **libbsd**.

Files

/usr/include/sys/types.h	Defines unsigned data types.
/usr/include/netinet/in.h	Defines Internet constants and structures.

Related Information

Additional conversion subroutines are the **htons** subroutine, **ntohl** subroutine, and **ntohs** subroutine.

Sockets Overview in *Communications Programming Concepts*.

htons Subroutine

Purpose

Converts an unsigned short integer from host byte order to Internet network byte order.

Syntax

#include <sys/types.h>
#include <netinet/in.h>

unsigned short htons (*HostShort*)
unsigned short *HostShort*;

Description

The **htons** (host to network short) subroutine converts an unsigned short (16–bit) integer from host byte order to Internet network byte order.

The Internet network requires ports and addresses in network standard byte order. Use the **htons** subroutine to convert addresses and ports from their host integer representation to network standard byte order.

The **htons** subroutine is defined in the **netinet/in.h** file as a macro.

Parameter

HostShort	Specifies a 16–bit integer in host byte order that is a host address or port.

Return Values

The **htons** subroutine returns a 16–bit integer in Internet network byte order (most significant byte first).

All applications containing the **htons** subroutine must be compiled with **_BSD** defined. In addition, when applicable, all socket applications must include the BSD library **libbsd**.

Implementation Specifics

The **htons** subroutine is part of AIX Base Operating System (BOS) Runtime.

Files

/usr/include/sys/types.h	Contains definitions of unsigned data types.
/usr/include/netinet/in.h	Defines Internet constants and structures.

Related Information

Additional conversion subroutines are the **htonl** subroutine, **ntohl** subroutine, and **ntohs** subroutine.

Sockets Overview in *Communications Programming Concepts.*

inet_addr Subroutine

Purpose

Converts Internet addresses to Internet numbers.

Library

(libc.a)

Syntax

#include <sys/socket.h>

#include <sys/socketvar.h>

#include <netinet/in.h>

#include <arpa/inet.h>

unsigned long inet_addr (*CharString*)
char **CharString*;

Description

The **inet_addr** subroutine interprets character strings representing numbers expressed in the Internet . (dot) notation, returning numbers suitable for use as Internet addresses.

All Internet addresses are returned in network order, with the first byte being the high–order byte.

Use C language integers when specifying each part of a dot notation.

Parameter

CharString	Represents a string of characters in the Internet address form.

Return Values

Upon successful completion, the **inet_addr** subroutine returns Internet addresses and Internet network numbers.

If the **inet_addr** subroutine fails, the subroutine returns a value of –1 (negative one).

Implementation Specifics

The **inet_addr** subroutine is part of AIX Base Operating System (BOS) Runtime.

All applications containing the **inet_addr** subroutine must be compiled with **_BSD** defined. In addition, when applicable, all socket applications must include the BSD library **libbsd**.

Files

/etc/hosts	Contains host names.
/etc/networks	Contains network names.
/usr/include/sys/socket.h	Contains socket definitions.
/usr/include/sys/socketvar.h	Defines the kernel structure per socket and contains buffer queues.

/usr/include/netinet/in.h	Defines Internet constants and structures.
/usr/include/arpa/inet.h	Contains external definitions for functions in inet.

Related Information

Internet address conversion subroutines are the **inet_lnaof** subroutine, **inet_makeaddr** subroutine, **inet_netof** subroutine, **inet_network** subroutine, and **inet_ntoa** subroutine.

Host information retrieval subroutines are the **endhostent** subroutine, **gethostbyaddr** subroutine, **gethostbyname** subroutine, **sethostent** subroutine.

Network information retrieval subroutines are the **endnetent** subroutine, **getnetbyaddr** subroutine, **getnetbyname** subroutine, **getnetent** subroutine, and **setnetent** subroutine.

Sockets Overview, Understanding Network Address Translation in *Communications Programming Concepts*.

inet_lnaof Subroutine

Purpose

Separates local Internet addresses into their network number and local network address.

Library

(libc.a)

Syntax

#include<sys/socket.h>

#include <sys/socket.h>

#include <netinet/in.h>

#include <arpa/inet.h>

int inet_lnaof (*InternetAddr*)
struct in_addr *InternetAddr*;

Description

The **inet_lnaof** subroutine breaks apart Internet addresses, returning the local network address part.

All Internet addresses are returned in network order, with the first byte being the high–order byte. All network numbers and local addresses are returned as integer values in machine format. Internet addresses are specified using a dot notation.

Use C language integers when specifying each part of a dot notation.

Parameter

InternetAddr	Specifies the Internet address to separate.

Return Values

Upon successful completion, the **inet_network** subroutine returns an Internet network number.

If the **inet_network** subroutine fails, the subroutine returns a –1 (negative one).

Implementation Specifics

The **inet_lnaof** subroutine is part of AIX Base Operating System (BOS) Runtime.

All applications containing the **inet_lnaof** subroutine must be compiled with **_BSD** defined. In addition, when applicable, all socket applications must include the BSD library **libbsd**.

Files

/etc/hosts	Contains host names.
/usr/include/sys/socket.h	Contains socket definitions.
/usr/include/sys/socketvar.h	Defines the kernel structure per socket and contains buffer queues.

/usr/include/netinet/in.h	Defines Internet constants and structures.
/usr/include/arpa/in.h	Contains external definitions for functions in inet.

Related Information

The **inet_addr** subroutine, **inet_makeaddr** subroutine, **inet_netof** subroutine, **inet_network** subroutine, and **inet_ntoa** subroutine.

Host information retrieval subroutines are the **endhostent** subroutine, **gethostbyaddr** subroutine, **gethostbyname** subroutine, and **sethostent** subroutine.

Network information retrieval subroutines are the **endnetent** subroutine, **getnetbyaddr** subroutine, **getnetbyname** subroutine, **getnetent** subroutine, and **setnetent** subroutine.

Sockets Overview, Understanding Network Address Translation in *Communications Programming Concepts*.

inet_makeaddr Subroutine

Purpose

Makes an Internet address.

Library

(libc.a)

Syntax

#include<sys/socket.h>

#include <sys/socket.h>

#include <netinet/in.h>

#include <arpa/inet.h>

struct in_addr inet_makeaddr (*Net*, *LocalNetAddr*)
int *Net*, *LocalNetAddr*;

Description

The **inet_makeaddr** takes an Internet network number and a local network address and constructs an Internet address from it.

All Internet addresses are returned in network order, with the first byte being the high–order byte. All network numbers and local addresses are returned as integer values in machine format. Internet addresses are specified using a dot notation.

Use C language integers when specifying each part of a dot notation.

Parameters

Net	Contains an Internet network number.
LocalNetAddr	Contains a local network address.

Return Values

Upon successful completion, the **inet_addr** subroutine returns an Internet address.

If the **inet_addr** subroutine is unsuccessful, the subroutine returns a –1 (negative one).

Implementation Specifics

The **inet_makeaddr** subroutine is part of AIX Base Operating System (BOS) Runtime.

All applications containing the **inet_makeaddr** subroutine must be compiled with **_BSD** defined. In addition, when applicable, all socket applications must include the BSD library **libbsd**.

Files

/etc/hosts	Contains host names.
/usr/include/netinet/in.h	Contains Internet constants and structures.
/usr/include/arpa/inet.h	Contains external definitions for functions in inet.

/usr/include/sys/socketvar.h	Defines the kernel structure per socket and contains buffer queues.
/usr/include/sys/socket.h	Contains socket definitions.

Related Information

Internet address conversion subroutines are the **inet_addr** subroutine, **inet_lnaof** subroutine, **inet_netof** subroutine, **inet_network** subroutine, and **inet_ntoa** subroutine.

Host information retrieval subroutines are the **endhostent** subroutine, **gethostbyaddr** subroutine, **gethostbyname** subroutine, **sethostent** subroutine.

Network information retrieval subroutines are the **endnetent** subroutine, **getnetbyaddr** subroutine, **getnetbyname** subroutine, **getnetent** subroutine, and **setnetent** subroutine.

Sockets Overview, Understanding Network Address Translation in *Communications Programming Concepts.*

inet_netof Subroutine

Purpose

Separates network Internet addresses into their network number and local network address.

Library

(libc.a)

Syntax

#include <sys/socket.h>

#include <sys/socketvar.h>

#include <netinet/in.h>

#include <arpa/inet.h>

int inet_netof (*InternetAddr*)
struct in_addr *InternetAddr*;

Description

The **inet_netof** subroutine breaks apart Internet addresses, returning the network number. Internet addresses are specified using a dot notation.

All Internet addresses are returned in network order, with the first byte being the high–order byte.

Use C language integers when specifying each part of a dot notation.

Parameter

InternetAddr	Specifies the Internet address to separate.

Return Values

Upon successful completion, the **inet_netof** subroutine returns a network number.

If the **inet_netof** subroutine fails, the subroutine returns a –1 (negative one).

Implementation Specifics

The **inet_netof** subroutine is part of AIX Base Operating System (BOS) Runtime.

All applications containing the **inet_netof** subroutine must be compiled with **_BSD** defined. In addition, when applicable, all socket applications must include the BSD library **libbsd**.

Files

/etc/hosts	Contains host names.
/etc/networks	Contains network names.
/usr/include/sys/socket.h	Contains socket definitions.
/usr/include/sys/socketvar.h	Defines the kernel structure per socket and contains buffer queues.
/usr/include/netinet/in.h	Defines Internet constants and structures.
/usr/include/arpa/inet.h	Contains external definitions for functions in inet.

Related Information

Internet address conversion subroutines are the **inet_addr** subroutine, **inet_lnaof** subroutine, **inet_makeaddr** subroutine, **inet_network** subroutine, and **inet_ntoa** subroutine.

Host information retrieval subroutines are the **endhostent** subroutine, **gethostbyaddr** subroutine, **gethostbyname** subroutine, and **sethostent** subroutine.

Network information retrieval subroutines are the **endnetent** subroutine, **getnetbyaddr** subroutine, **getnetbyname** subroutine, **getnetent** subroutine, and **setnetent** subroutine.

Sockets Overview, Understanding Network Address Translation in *Communications Programming Concepts*.

inet_network Subroutine

Purpose

Converts Internet network addresses in . (dot) notation to Internet numbers.

Library

(libc.a)

Syntax

#include <sys/socket.h>

#include <sys/socket.h>

#include <netinet/in.h>

#include <arpa/inet.h>

unsigned long inet_network (*CharString*)
char **CharString*;

Description

The **inet_network** subroutine interprets character strings representing numbers expressed in the Internet . (dot) notation and returns numbers suitable for use as Internet addresses and Internet network numbers.

All Internet addresses are returned in network order, with the first byte being the high–order byte.

Use C language integers when specifying each part of a dot notation.

Parameter

CharString	Represents a string of characters in the Internet address form.

Return Values

Upon successful completion, the **inet_network** subroutine returns numbers suitable for use as Internet network numbers.

If the **inet_network** subroutine fails, the subroutine returns a value of –1 (negative one).

Implementation Specifics

The **inet_network** subroutine is part of AIX Base Operating System (BOS) Runtime.

All applications containing the **inet_network** subroutine must be compiled with **_BSD** defined. In addition, when applicable, all socket applications must include the BSD library **libbsd**.

Files

/etc/hosts	Contains host names.
/etc/networks	Contains network names.
/usr/include/sys/socket.h	Contains socket definitions.
/usr/include/sys/socketvar.h	Defines the kernel structure per socket and contains buffer queues.

/usr/include/netinet/in.h	Defines Internet constants and structures.
/usr/include/arpa/inet.h	Contains external definitions for functions in Internet.

Related Information

Internet address conversion subroutines are the **inet_addr** subroutine, **inet_lnaof** subroutine, **inet_makeaddr** subroutine, **inet_netof** subroutine, and **inet_ntoa** subroutine.

Host information retrieval subroutines are the **endhostent** subroutine, **gethostbyaddr** subroutine, **gethostbyname** subroutine, **sethostent** subroutine.

Network information retrieval subroutines are the **endnetent** subroutine, **getnetbyaddr** subroutine, **getnetbyname** subroutine, **getnetent** subroutine, and **setnetent** subroutine.

Sockets Overview, Understanding Network Address Translation in *Communications Programming Concepts*.

inet_ntoa Subroutine

Purpose

Converts an Internet address into an ASCII string.

Library

(libc.a)

Syntax

#include <sys/socket.h>

#include <netinet/in.h>

#include <arpa/inet.h>

char *inet_ntoa (*InternetAddr*)
struct in_addr *InternetAddr*;

Description

The **inet_ntoa** subroutine takes an Internet address and returns an ASCII string representing the Internet address in dot notation. All Internet addresses are returned in network order, with the first byte being the high–order byte.

Use C language integers when specifying each part of a dot notation.

Parameter

InternetAddr	Contains the Internet address to be converted to ASCII.

Return Values

Upon successful completion, the **inet_ntoa** subroutine returns an Internet address.

If the **inet_ntoa** subroutine fails, the subroutine returns a –1 (negative one).

Implementation Specifics

The **inet_ntoa** subroutine is part of AIX Base Operating System (BOS) Runtime.

All applications containing the **inet_ntoa** subroutine must be compiled with **_BSD** defined. In addition, when applicable, all socket applications must include the BSD library **libbsd**.

Files

/etc/hosts	Contains host names.
/etc/networks	Contains network names.
/usr/include/sys/socket.h	Contains socket definitions.
/usr/include/sys/socketvar.h	Defines the kernel structure per socket and contains buffer queues.
/usr/include/netinet/in.h	Defines Internet constants and structures.

Related Information

Internet address conversion subroutines are the **inet_addr** subroutine, **inet_lnaof** subroutine, **inet_makeaddr** subroutine, **inet_network** subroutine, and **inet_ntoa** subroutine.

Host information retrieval subroutines are the **endhostent** subroutine, **gethostbyaddr** subroutine, **gethostbyname** subroutine, and **sethostent** subroutine.

Network information retrieval subroutines are the **endnetent** subroutine, **getnetbyaddr** subroutine, **getnetbyname** subroutine, **getnetent** subroutine, and **setnetent** subroutine.

Sockets Overview, Understanding Network Address Translation in *Communications Programming Concepts.*

listen Subroutine

Purpose

Listens for socket connections and limits the backlog of incoming connections.

Syntax

int listen (*Socket*, *Backlog*)

int *Socket*, *Backlog*;

Description

The **listen** subroutine performs the following activities:

1. Identifies the socket that receives the connections.
2. Marks the socket as **accepting** connections.
3. Limits the number (*Backlog)* of outstanding connection requests in the system queue.

The maximum queue length (*Backlog*) that the **listen** subroutine can specify is ten (10). The maximum queue length is indicated by the **SOMAXCONN** value in the **/include/sys/socket.h** file.

Parameters

Socket	Specifies the unique name for the socket.
Backlog	Specifies the maximum number of outstanding connection requests.

Return Value

Upon successful completion, the **listen** subroutine returns a value 0 (zero).

If the **listen** subroutine fails, the subroutine handler performs the following functions:

- Returns a value of –1 (negative one) to the calling program
- Moves an error code, indicating the specific error, into the global variable **errno**.

Error Codes

The subroutine fails if any one of the following errors occurs:

EBADF	The *Socket* parameter is not valid.
ECONNREFUSED	A connection request arrived exceeding the backlog amount.
ENOTSOCK	The *Socket* parameter refers to a file, not a socket.
EOPNOTSUPP	The referenced socket is not a type that supports the **listen** subroutine.

Examples

1. The following program fragment illustrates the use of the **listen** subroutine with five (5) as the maximum number of outstanding connections which may be queued awaiting acceptance by the server process.

```
listen(s,5)
```

Implementation Specifics

The **listen** subroutine is part of AIX Base Operating System (BOS) Runtime.

All applications containing the **listen** subroutine must be compiled with **_BSD** defined. In addition, when applicable, all socket applications must include the BSD library **libbsd**.

Files

/usr/include/sys/socket.h	Contains socket definitions.
/usr/include/sys/socketvar.h	Defines the kernel structure per socket and contains buffer queues.
/usr/include/sys/types.h	Contains definitions for unsigned data types.

Related Information

Other creation and connection socket subroutines are the **accept** subroutine, **connect** subroutine, and **socket** subroutine.

Sockets Overview, Understanding Socket Connections in *Communications Programming Concepts*.

ntohl Subroutine

Purpose

Converts an unsigned long integer from Internet network standard byte order to host byte order.

Syntax

#include<sys/types.h>
#include <netinet/in.h>

unsigned long ntohl (*NetLong*)
unsigned long *NetLong*;

Description

The **ntohl** (network to host long) subroutine converts an unsigned long (32–bit) integer from Internet network standard byte order to host byte order.

Receiving hosts require addresses and ports in host byte order. Use the **ntohl** subroutine to convert Internet addresses and ports to the host integer representation.

The **ntohl** subroutine is defined in the **netinet/in.h** file as a macro.

Parameter

NetLong	Requires a 32–bit integer in network byte order.

Return Values

The **ntohl** subroutine returns a 32–bit integer in host byte order.

Implementation Specifics

The **ntohl** subroutine is part of AIX Base Operating System (BOS) Runtime.

All applications containing the **ntohl** subroutine must be compiled with **_BSD** defined. In addition, when applicable, all socket applications must include the BSD library **libbsd**.

Files

/usr/include/sys/types.h	Contains definitions of unsigned data types.
/usr/include/netinet/in.h	Defines Internet constants and structures.

Related Information

Additional conversion subroutines are the **htonl** subroutine, **htons** subroutine, and **ntohs** subroutine.

Host address retrieval subroutines are the **endhostent** subroutine, **gethostbyaddr** subroutine, **gethostbyname** subroutine, and **sethostent** subroutine.

Port retrieval subroutines are the **endservent** subroutine, the **getservbyname** subroutine, **getservbyport** subroutine, **getservent** subroutine, and **setservent** subroutine.

Sockets Overview in *Communications Programming Concepts*.

ntohs Subroutine

Purpose

Converts an unsigned short integer from Internet network byte order to host byte order.

Syntax

#include<sys/types.h>
#include <netinet/in.h>

unsigned short ntohs (*NetShort*)
unsigned short *NetShort*;

Description

The **ntohs** (network to host short) subroutine converts an unsigned short (16–bit) integer from Internet network byte order to the host byte order.

Receiving hosts require Internet addresses and ports in host byte order. Use the **ntohs** subroutine to convert Internet addresses and ports to the host integer representation.

The **ntohs** subroutine is defined in the **netinet/in.h** file as a macro.

Parameter

NetShort	Requires a 16–bit integer in network standard byte order.

Return Value

The **ntohs** subroutine returns the supplied integer in host byte order.

Implementation Specifics

The **ntohs** subroutine is part of AIX Base Operating System (BOS) Runtime.

All applications containing the **ntohs** subroutine must be compiled with **_BSD** defined. In addition, when applicable, all socket applications must include the BSD library **libbsd**.

Files

/usr/include/sys/types.h	Contains definitions of unsigned data types.
/usr/include/netinet/in.h	Defines Internet constants and structures.

Related Information

Additional conversion subroutines are the **htonl** subroutine, **htons** subroutine, and **ntohl** subroutine.

Host address retrieval subroutines are the **endhostent** subroutine, **gethostbyaddr** subroutine, **gethostbyname** subroutine,and **sethostent** subroutine.

Port retrieval subroutines are the **endservent** subroutine, **getservbyname** subroutine, **getservbyport** subroutine, **getservent** subroutine, and **setservent** subroutine.

Sockets Overview in *Communications Programming Concepts.*

_putlong Subroutine

Purpose

Places long byte quantities into the byte stream.

Library

(libc.a)

Syntax

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

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

Description

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

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

Parameters

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

Implementation Specifics

The **_putlong** subroutine is part of AIX Base Operating System (BOS) Runtime.

All applications containing the **_putlong** subroutine must be compiled with **_BSD** defined. In addition, when applicable, all socket applications must include the BSD library **libbsd**.

Files

/etc/resolv.conf	Lists the name server and domain name.
/usr/include/resolv.h	Contains global information used by the resolver subroutines.
/usr/include/arpa/nameser.h	Defines the Internet name server structure.
/usr/include/sys/types.h	Contains definitions of unsigned data types.
/usr/include/netinet/in.h	Defines Internet constants and structures.

Related Information

Domain name access subroutines are the **res_init** subroutine, **res_mkquery** subroutine, and **res_send** subroutine.

Domain name translation subroutines are the **dn_comp** subroutine, **dn_expand** subroutine, **dn_find** subroutine, and **dn_skipname** subroutine.

Byte stream and byte boundary retrieval subroutines are the **_getlong** subroutine, **_getshort** subroutine, and **putshort** subroutine.

Sockets Overview, Understanding Domain Name Resolution in *Communications Programming Concepts.*

_putshort Subroutine

Purpose

Places short byte quantities into the byte stream.

Library

(libc.a)

Syntax

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

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

Description

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

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

Parameters

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

Implementation Specifics

The **__putshort** subroutine is part of AIX Base Operating System (BOS) Runtime.

All applications containing the **__putshort** subroutine must be compiled with **_BSD** defined. In addition, when applicable, all socket applications must include the BSD library **libbsd**.

Files

/etc/resolv.conf	Lists the name server and domain name.
/usr/include/resolv.h	Contains global information used by the resolver subroutines.
/usr/include/arpa/nameser.h	Contains the Internet name server.
/usr/include/sys/types.h	Defines unsigned data types.
/usr/include/netinet/in.h	Contains Internet constants and structures.

Related Information

Domain name access subroutines are the **res_init** subroutine, **res_mkquery** subroutine, and **res_send** subroutine.

Domain name translation subroutines are the **dn_comp** subroutine, **dn_expand** subroutine, **dn_find** subroutine, and **dn_skipname** subroutine.

Byte stream and byte boundary retrieval subroutines are the **_getlong** subroutine, **_getshort** subroutine, and **putlong** subroutine.

Sockets Overview, Understanding Domain Name Resolution in *Communications Programming Concepts.*

rcmd Subroutine

Purpose

Allows execution of commands on a remote host

Library

(libc.a)

Syntax

int rcmd (*Host*, *Port*, *LocalUser*, *RemoteUser*, *Command*, *ErrFileDesc*)
char ***Host*;
u_short *Port*;
char **LocalUser*, **RemoteUser*, **Command*;
int **ErrFileDesc*;

Description

The **rcmd** (remote command) subroutine allows execution of certain commands on a remote host that supports **rshd, rlogin**, and **rpc** among others.

Only processes with an effective user ID of root user can use the **rcmd** subroutine. An authentication scheme based on remote port numbers is used to verify permissions. Ports in the range from 0 to 1023 can only be used by a root user.

The **rcmd** subroutine looks up a host via the nameserver or if the local nameserver isn't running, via the /**etc/hosts** file.

If the connection succeeds, a socket in the Internet domain of type **SOCK_STREAM** is returned to the calling process and given to the remote command as standard input (**stdin)** and standard output (**stdout**).

Always specify the *Host* name. If the local domain and remote domain are the same, specifying the domain parts is optional.

Parameters

Host Specifies the name of a remote host that is listed in the /**etc/hosts** file. If the specified name of the host is not found in this file, the **rcmd** subroutine fails.

Port Specifies the well–known port to use for the connection. The /**etc/services** file contains the DARPA Internet services, their ports, and socket types.

LocalUser and *RemoteUser*
Points to user names that are valid at the local and remote host, respectively. Any valid user name can be given.

Command Specifies the name of the command to be executed at the remote host.

ErrFileDesc Specifies an integer controlling the set up of communications channels. Integer options are as follows:

- Not 0 (zero) = an auxiliary channel to a control process is set up, and the *ErrFileDesc* parameter points to the file descriptor for the channel. The control process provides diagnostic output from the remote command on this channel and also accepts bytes as signal numbers to be forwarded to the process group of the command.

- 0 (zero) = the standard error (**stderr**) of the remote command is the same as standard output (**stdout**), and no provision is made for sending arbitrary signals to the remote process. However, it is possible to send out-of-band data to the remote command.

Return Values

Upon successful completion, the **rcmd** subroutine returns a valid socket descriptor.

Upon unsuccessful completion, the **rcmd** subroutine returns a value of –1 (negative one). The subroutine returns a –1 (negative one), if the effective user ID of the calling process is not root user or if the subroutine fails to resolve the host.

Implementation Specifics

The **rcmd** subroutine is part of AIX Base Operating System (BOS) Runtime.

All applications containing the **rcmd** subroutine must be compiled with **_BSD** defined. In addition, when applicable, all socket applications must include the BSD library **libbsd**.

Files

/etc/services	Contains the service names, ports, and socket type.
/etc/hosts	Contains host names and their addresses for hosts in a network.
/etc/resolv.config	Contains the name server and domain name.

Related Information

Additional remote command execution subroutines are the **rresvport** subroutine and **ruserok** subroutine.

TCP/IP Interface Program commands are the **rlogind** command and **rshd** command.

TCP/IP daemons are the **named** daemon.

System calls to get and set the host name are, respectively, the **gethostname** subroutine and **sethostname** subroutine.

Sockets Overview in *Communications Programming Concepts*.

recv Subroutine

Purpose

Receives messages from connected sockets.

Syntax

#include <sys/types.h>
#include <sys/socket.h>

#include <sys/socketvar.h>

int recv (*Socket*, *Buffer*, *Length*, *Flags*)
int *Socket*;
char **Buffer*;
int *Length*, *Flags*;

Description

The **recv** (receive) subroutine receives messages from a connected socket. The **recvfrom** and **recvmsg** subroutines receive messages from both connected and unconnected sockets. However, they are usually used for unconnected sockets only.

The **recv** subroutine returns the length of the message. If a message is too long to fit in the supplied buffer, excess bytes may be truncated depending on the type of socket that issued the message.

If no messages are available at the socket, the **recv** subroutine waits for a message to arrive, unless the socket is nonblocking. If a socket is nonblocking, the system returns an error.

Use the **select** subroutine to determine when more data arrives.

Parameters

Socket — Specifies the socket descriptor.

Buffer — Specifies an address where the message should be placed.

Length — Specifies the size of the *Buffer* parameter.

Flags — Points to a value controlling the message reception. The **/sys/socket.h** file defines the *Flags* value. The argument to receive a call is formed by logically ORing one or more of the following values:

- **MSG_PEEK** — Peeks at incoming message. The data is treated as unread and the next **recv** (or similar call) will still return this data.
- **MSG_OOB** — Processes out-of-band data.

Return Value

Upon successful completion, the **recv** subroutine returns the length of the message in bytes.

If the **recv** subroutine fails, the subroutine handler performs the following functions:

- Returns a value of –1 (negative one) to the calling program
- Moves an error code, indicating the specific error, into the global variable **errno**

Error Codes

The **recv** subroutine fails if any one of the following errors occurs:

EBADF	The *Socket* parameter is not valid.
ENOTSOCK	The *Socket* parameter refers to a file, not a socket.
EWOULDBLOCK	The socket is marked nonblocking, and no connections are present to be accepted.
EINTR	A signal interrupted the **recv** subroutine before any data was available.
EFAULT	The data was directed to be received into a non–existent or protected part of the process address space. The *Buffer* is invalid.

Implementation Specifics

The **recv** subroutine is part of AIX Base Operating System (BOS) Runtime.

All applications containing the **recv** subroutine must be compiled with **_BSD** defined. In addition, when applicable, all socket applications must include the BSD library **libbsd**.

Files

/usr/include/sys/socket.h	Contains socket definitions.
/usr/include/sys/socketvar.h	Defines the kernel structure per socket and contains buffer queues.
/usr/include/sys/types.h	Contains definitions for unsigned data types.

Related Information

The **fgets** subroutine and **fputs** subroutine.

Additional receive subroutines are the **recvfrom** subroutine and **recvmsg** subroutine.

Subroutines for sending messages over sockets are the **send** subroutine, **sendmsg** subroutine, and **sendto** subroutine.

Socket subroutines for disabling communications, creating sockets and monitoring data reception are, respectively, the **select** subroutine, **shutdown** subroutine, and **socket** subroutine.

The **read** subroutine and **write** subroutine.

Sockets Overview, Understanding Socket Data Transfer in *Communications Programming Concepts.*

recvfrom Subroutine

Purpose

Receives messages from sockets.

Syntax

#include <sys/types.h>

#include <sys/socket.h>

#include <sys/socket.h>

int recvfrom (*Socket*, *Buffer*, *Length*, *Flags*, *From*, *FromLength*)
int *Socket*;
char **Buffer*;
int *Length*, *Flags*;
struct sockaddr **From*;
int **FromLength*;

Description

The **recvfrom** subroutine allows an application program to receives messages from unconnected sockets. The **recvfrom** subroutine is normally applied to unconnected sockets as it includes parameters that allow the calling program to specify the source point of the data to be received.

To return the source address of the message, specify a non–Null value for the *From* parameter. The **recvfrom** subroutine initializes the *FromLength* parameter to the size of the buffer associated with the *From* parameter. On return, the **recvfrom** subroutine modifies the *FromLength* parameter to indicate the actual size of the stored address. The **recvfrom** subroutine returns the length of the message. If a message is too long to fit in the supplied buffer, excess bytes may be truncated depending on the type of socket that issued the message.

If no messages are available at the socket, the **recvfrom** subroutine waits for a message to arrive, unless the socket is nonblocking. If the socket is nonblocking, the system returns an error.

Parameters

Socket Specifies the socket descriptor.

Buffer Specifies an address where the message should be placed.

Length Specifies the size of the *Buffer* parameter.

Flags Points to a value controlling the message reception. The argument to receive a call is formed by logically ORing one or more of the values shown in the following list:

MSG_PEEK Peeks at incoming message.

MSG_OOB Processes out–of–band data.

From	Points to a socket structure, filled in with source's address.
FromLength	Specifies the length of the sender's or source's address.

Return Value

If the **recvfrom** subroutine is successful, the subroutine returns the length of the message in bytes.

If the call is unsuccessful, the subroutine handler performs the following functions:

- Returns a value of –1 (negative one) to the calling program
- Moves an error code, indicating the specific error, into the global variable **errno**

Error Codes

The **recvfrom** subroutine fails if any one of the following errors occurs:

EBADF	The *Socket* parameter is not valid.
ENOTSOCK	The *Socket* parameter refers to a file, not a socket.
EWOULDBLOCK	The socket is marked nonblocking, and no connections are present to be accepted.
EFAULT	The data was directed to be received into a non–existent or protected part of the process address space. The buffer is invalid.

Implementation Specifics

The **recvfrom** subroutine is part of AIX Base Operating System (BOS) Runtime.

All applications containing the **recvfrom** subroutine must be compiled with **_BSD** defined. In addition, when applicable, all socket applications must include the BSD library **libbsd.**

Files

/usr/include/sys/socket.h	Contains socket definitions.
/usr/include/sys/socketvar.h	Defines the kernel structure per socket and contains buffer queues.
/usr/include/sys/types.h	Contains definitions for unsigned data types.

Related Information

The **fgets** subroutine and **fputs** subroutine.

Additional socket receive subroutines are the **recv** subroutine and **recvmsg** subroutine.

Subroutines for sending messages over sockets are the **send** subroutine, **sendmsg** subroutine, and **sendto** subroutine.

Socket subroutines for disabling communications, creating sockets and monitoring data reception are, respectively, the **select** subroutine, **shutdown** subroutine, and **socket** subroutine.

The **read** subroutine and **write** subroutine.

Sockets Overview, Understanding Socket Data Transfer in *Communications Programming Concepts*.

recvmsg Subroutine

Purpose

Receives a message from any socket.

Syntax

#include <sys/types.h>
#include <sys/socket.h>

#include <sys/socketvar.h>

int recvmsg (*Socket*, *Message*, *Flags*)
int *Socket*;
struct msghdr *Message*[ ];
int *Flags*;

Description

The **recvmsg** subroutine receives messages from unconnected or connected sockets. The **recvmsg** subroutine returns the length of the message. If a message is too long to fit in the supplied buffer, excess bytes may be truncated depending on the type of socket that issued the message.

If no messages are available at the socket, the **recvmsg** subroutine waits for a message to arrive. If the socket is nonblocking and no messages are available, the **recvmsg** subroutine fails.

Use the **select** subroutine to determine when more data arrives.

The **recvmsg** subroutine uses a **msghdr** structure to decrease the number of directly supplied parameters. The **msghdr** structure is defined in the **sys/socket.h** header file.

Parameters

Socket Specifies the unique name of the socket.

Message Points to the address of the **msghdr** structure which contains both the address for the incoming message and the space for the sender address.

Flags Permits the subroutineer to exercise control over the reception of messages. The *Flags* parameter to receive a call is formed by logically ORing one or more of the values shown in the following list:

MSG_PEEK Peeks at incoming message.

MSG_OOB Processes out–of–band data.

The /**sys/socket.h** file contains the possible values for the *Flags* parameter.

Return Value

Upon successful completion, the length of the message in bytes is returned.

If the **recvmsg** subroutine fails, the subroutine handler performs the following functions:

- Returns a value of –1 (negative one) to the calling program
- Moves an error code, indicating the specific error, into the global variable **errno**

Error Codes

The **recvmsg** subroutine fails if any one of the following error codes occurs:

EBADF	The *Socket* parameter is not valid.
ENOTSOCK	The *Socket* parameter refers to a file, not a socket.
EWOULDBLOCK	The socket is marked nonblocking, and no connections are present to be accepted.
EINTR	The **recvmsg** subroutine was interrupted by delivery of a signal before any data was available for the receive.
EFAULT	The *Address* parameter is not in a writable part of the user address space.

Implementation Specifics

The **recvmsg** subroutine is part of AIX Base Operating System (BOS) Runtime.

All applications containing the **recvmsg** subroutine must be compiled with **_BSD** defined. In addition, when applicable, all socket applications must include the BSD library **libbsd**.

Files

/usr/include/sys/socket.h	Contains socket definitions.
/usr/include/sys/socketvar.h	Defines the kernel structure per socket and contains buffer queues.
/usr/include/sys/types.h	Contains definitions for unsigned data types.

Related Information

Additional socket receive subroutines are the **recv** subroutine and **recvfrom** subroutine.

Socket send subroutines are the **send** subroutine, **sendmsg** subroutine, and **sendto** subroutine.

Socket subroutines for closing communications, monitoring data broadcasts, and creating sockets are, respectively, the **select** subroutine, **shutdown** subroutine, and **socket** subroutine.

res_init Subroutine

Purpose

Searches for a default domain name and Internet address.

Library

(libc.a)

Syntax

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

void res_init ( )

Description

The **res_init** subroutine reads the **/etc/resolv.conf** file for the default domain name and the Internet address of the initial hosts running the name server.

Note: If the **/etc/resolv.conf** file does not exist, the **res_init** subroutine attempts name resolution using the local **/etc/hosts** file. If the system is not using a domain name server, the **/etc/resolv.conf** file should not exist. The **/etc/host** file should be present on the system even if the system is using a name server. In this instance, the file should contain the host ids that the system requires to function even if the name server is not functioning.

The **res_init** subroutine is one of a set of subroutines that form the resolver, a set of functions that translate domain names to Internet addresses. All resolver subroutines use the **/usr/include/resolv.h** header file, which defines the **_res** structure. The **res_init** subroutine stores domain name information in the **_res** structure.

Implementation Specifics

The **res_init** subroutine is part of AIX Base Operating System (BOS) Runtime.

All applications containing the **res_init** subroutine must be compiled with **_BSD** defined. In addition, when applicable, all socket applications must include the BSD library **libbsd**.

Files

/etc/resolv.conf	Contains the name server and domain name.
/etc/hosts	Contains host names and their addresses for hosts in a network. This file is used to resolve a host name into an Internet address.
/usr/include/arpa/nameser.h	Contains the Internet name server.
/usr/include/netinet/in.h	Contains Internet constants and structures.
/usr/include/resolv.h	Contains global information used by the resolver subroutines.
/usr/include/sys/types.h	Contains definitions of unsigned data types.

Related Information

Domain name access subroutines are the **res_mkquery** subroutine and **res_send** subroutine.

Domain name translation subroutines are the **dn_comp** subroutine, **dn_expand** subroutine, **dn_find** subroutine, and **dn_skipname** subroutine.

Byte stream and byte boundary retrieval subroutines are the **_getlong** subroutine, **_getshort** subroutine, **putlong** subroutine, and **putshort** subroutine.

Sockets Overview, Understanding Domain Name Resolution in *Communications Programming Concepts*.

res_mkquery Subroutine

Purpose

Makes query messages for name servers.

Library

(libc.a)

Syntax

#include <sys/types.h>

#include <netinet/in.h>

#include <arpa/nameser.h>

#include <resolv.h>

int res_mkquery (*Operation*, *DomName*, *Class*, *Type*, *Data*, *DataLength*, *Reserved*,
Buffer, *BufferLength*)
int *Operation*;
char **DomName*;
int *Class*, *Type*;
char **Data*;
int *DataLength*;
struct rrec **Reserved*;
char **Buffer*;
int *BufferLength*;

Description

The **res_mkquery** subroutine makes packets for name servers in the Internet domain. The **res_mkquery** subroutine makes a standard query message and places this message in the location pointed to by the *Buffer* parameter.

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

Parameters

Operation — Specifies a query type. The usual type is **QUERY**, but the parameter can be set to any of the query types defined in the **arpa/nameser.h** file.

DomName — Points to the name of the domain. If the *DomName* parameter points to a single label and the **RES_DEFNAMES** bit is set, as it is by default, the subroutine appends *DomName* to the current domain name. The current domain name is defined by the name server in use or in the **/etc/resolv.conf** file.

Class	Specifies one of the following parameters:
	C_IN Specifies the ARPA Internet.
	C_CHAOS Specifies the Chaos network at MIT.
Type	Requires one of the following values:
	T_A Host address
	T_NS Authoritative server
	T_MD Mail destination
	T_MF Mail forwarder
	T_CNAME Canonical name
	T_SOA Start of authority zone
	T_MB Mailbox domain name
	T_MG Mail group member
	T_MR Mail rename name
	T_NULL **NULL** resource record
	T_WKS Wellknown service
	T_PTR Domain name pointer
	T_HINFO Host information
	T_MINFO Mailbox information
	T_MX Mail routing information
	T_UINFO User (**finger**) information
	T_UID User ID
	T_GID Group ID.
Data	Points to the data that is sent to the name server as a search key. The data is stored as a character array.
DataLength	Defines the size of the array pointed to by the *Data* parameter.
Reserved	Specifies a reserved and currently unused parameter.
Buffer	Points to a location containing the query message.
BufferLength	Specifies the length of the message pointed to by the *Buffer* parameter.

Return Value

Upon successful completion, the **res_mkquery** subroutine returns the size of the query. If the query is larger than the value of the *BufferLength* parameter, the subroutine fails and returns a value of –1 (negative one).

Implementation Specifics

The **res_mkquery** subroutine is part of AIX Base Operating System (BOS) Runtime.

All applications containing the **res_mkquery** subroutine must be compiled with **_BSD** defined. In addition, when applicable, all socket applications must include the BSD library **libbsd**.

Files

/etc/resolv.conf	Contains the name server and domain name.
/usr/include/resolv.h	Contains global information used by the resolver subroutines.
/usr/include/arpa/nameser.h	Contains Internet name servers.
/usr/include/sys/types.h	Contains definitions of unsigned data types.
/usr/include/netinet/in.h	Contains Internet constants and structures.

Related Information

Domain name access subroutines are the **res_init** subroutine and **res_send** subroutine.

Domain name translation subroutines are the **dn_comp** subroutine, **dn_expand** subroutine, **dn_find** subroutine, and **dn_skipname** subroutine.

Byte stream and byte boundary retrieval subroutines are the **_getlong** subroutine, **_getshort** subroutine, **putlong** subroutine, and **putshort** subroutine.

Sockets Overview, Understanding Domain Name Resolution in *Communications Programming Concepts.*

res_send Subroutine

Purpose

Sends a query to a name server and retrieves a response.

Library

(libc.a)

Syntax

#include <sys/types.h>

#include <netinet/in.h>

#include <arpa/nameser.h>

#include <resolv.h>

int res_send (*MessagePtr*, *MessageLength*, *Answer*, *AnswerLength*)
char **MsgPtr*;
int *MsgLength*;
char **Answer*;
int *AnswerLength*;

Description

The **res_send** subroutine sends a query to name servers and calls the **res_init** subroutine if the **RES_INIT** option of the **_res** structure is not set. This subroutine sends the query to the local name server and handles timeouts and retries.

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

Parameters

MessagePtr	Points to the beginning of a message.
MessageLength	Specifies the length of the message.
Answer	Points to an address where the response is stored.
AnsLength	Specifies the size of the answer area.

Return Value

Upon successful completion, the **res_send** subroutine returns the length of the message.

If the **res_send** subroutine fails, the subroutine returns a –1 (negative one).

Implementation Specifics

The **res_send** subroutine is part of AIX Base Operating System (BOS) Runtime.

All applications containing the **res_send** subroutine must be compiled with **_BSD** defined. In addition, when applicable, all socket applications must include the BSD library **libbsd**.

Files

/etc/resolv.conf	Contains general name server and domain name information.
/usr/include/resolv.h	Contains global information used by the resolver subroutines.
/usr/include/arpa/nameser.h	Contains general Internet name server information.
/usr/include/sys/types.h	Contains definitions of unsigned data types.
/usr/include/netinet/in.h	Contains Internet constants and structures.

Related Information

Domain name access subroutines are the **res_init** subroutine and **res_mkquery** subroutine.

Domain name translation subroutines are the **dn_comp** subroutine, **dn_expand** subroutine, **dn_find** subroutine, and **dn_skipname** subroutine.

Byte stream and byte boundary retrieval subroutines are the **_getlong** subroutine, **_getshort** subroutine, **putlong** subroutine, and **putshort** subroutine.

Sockets Overview, Understanding Domain Name Resolution in *Communications Programming Concepts.*

rexec Subroutine

Purpose

Allows command execution on a remote host.

Library

(libc.a)

Syntax

int rexec (*Host*, *Port*, *User*, *Passwd*, *Command*, *ErrFileDescParam*)

char ***Host*;

int *Port*; **char** **User*, **Passwd*, **Command*;

int **ErrFileDescParam*;

Description

The **rexec** (remote execution) subroutine allows the calling process to execute commands on a remote host.

If the **rexec** connection succeeds, a socket in the Internet domain of type **SOCK_STREAM** is returned to the calling process and is given to the remote command as standard input and standard output.

Parameters

Host
: Contains the name of a remote host that is listed in the /**etc/hosts** file or /**etc/resolv.config** file. If the name of the host is not found in either file, the **rexec** fails.

Port
: Specifies the well-known DARPA Internet port to use for the connection. A pointer to the structure that contains the necessary port can be obtained by issuing the following library call:

  getservbyname("exec","tcp")

User and *Passwd*
: Points to a user ID and password valid at the host. If these parameters are not supplied, the **rexec** subroutine takes the following actions until finding a user ID and password to send to the remote host:

  1. Searches the current environment for the user ID and password on the remote host.
  2. Searches the user's home directory for a file called **$HOME/.netrc** that contains a user ID and password.
  3. Prompts the user for a user ID and password.

Command
: Points to the name of the command to be executed at the remote host.

ErrFileDescParam
: Specifies one of the following values:

  - Not 0 (zero) = an auxiliary channel to a control process is set up, and a descriptor for it is placed in the *ErrFileDescParam* parameter. The control process provides diagnostic output from the remote command on this channel and also accepts bytes as signal numbers to be forwarded to the process group of the command. This diagnostic information does not include remote authorization failure, since this connection is set up after authorization has been verified.

- 0 (zero) = the standard error of the remote command is the same as standard output, and no provision is made for sending arbitrary signals to the remote process. In this case, however, it may be possible to send out–of–band data to the remote command.

Return Value

Upon successful completion, the system returns a socket to the remote command.

If the **rexec** subroutine is unsuccessful, the system returns a –1 (negative one) indicating that the specified host name does not exist.

Implementation Specifics

The **rexec** subroutine is part of AIX Base Operating System (BOS) Runtime.

All applications containing the **rexec** subroutine must be compiled with **_BSD** defined. In addition, when applicable, all socket applications must include the BSD library **libbsd**.

Files

/etc/hosts	Contains host names and their addresses for hosts in a network. This file is used to resolve a host name into an Internet address.
/etc/resolv.config	Contains the name server and domain name.
$HOME/.netrc	Contains Automatic login information.

Related Information

The **rexecd** command.

The **getservbyname** subroutine.

Additional remote command execution subroutines are the **rcmd** subroutine, **rresvport** subroutine, and **ruserok** subroutine.

Sockets Overview in *Communications Programming Concepts*.

The TCP/IP Overview for System Management in *Communication Concepts and Procedures*.

rresvport Subroutine

Purpose

Retrieves a socket with a privileged address.

Library

(libc.a)

Syntax

int rresvport (*Port*)
int **Port*;

Description

The **rresvport** subroutine obtains a socket with a privileged address bound to the socket. A privileged Internet port is one that falls in the range of 0 to 1023.

Only processes with an effective user ID of root user can use the **rresvport** subroutine. An authentication scheme based on remote port numbers is used to verify permissions.

If the connection succeeds, a socket in the Internet domain of type **SOCK_STREAM** is returned to the calling process.

Parameters

Port Specifies the port to use for the connection.

Return Values

Upon successful completion, the **rresvport** subroutine returns a valid, bound socket descriptor.

If the **rresvport** subroutine fails, the subroutine handler performs the following functions:

- Returns a value of –1 (negative one) to the calling program.
- Moves an error code, indicating the specific error, into the global variable **errno**

Error Codes

The **rresvport** subroutine fails if any one of the following errors occur:

EAGAIN All network ports are in use.

EAFNOSUPPORT
The addresses in the specified address family cannot be used with this socket.

EMFILE Two hundred (200) file descriptors are currently open.

ENFILE The system file table is full.

ENOBUFS Insufficient buffers are available in the system to complete the subroutine.

Implementation Specifics

The **rresvport** subroutine is part of AIX Base Operating System (BOS) Runtime.

All applications containing the **rresvport** subroutine must be compiled with **_BSD** defined. In addition, when applicable, all socket applications must include the BSD library **libbsd**.

File

/etc/services	Contains the service names.

Related Information

Additional remote command execution subroutines are the **rcmd** subroutine and **ruserok** subroutine.

Sockets Overview in *Communications Programming Concepts.*

ruserok Subroutine

Purpose

The **ruserok** subroutine allows servers to authenticate clients.

Library

(libc.a)

Syntax

int ruserok (*Host*, *RootUser*, *RemoteUser*, *LocalUser*)
char **Host*;
int *RootUser*;
char **RemoteUser*, **LocalUser*;

Description

The **ruserok** (remote command user OK) subroutine allows servers to authenticate clients requesting services.

Always specify the host name. If the local domain and remote domain are the same, specifying the domain parts is optional. To determine the domain of the host, use the **gethostname** subroutine.

Parameters

Host
: Specifies the name of a remote host. The **ruserok** subroutine checks for this host in the **/etc/host.equiv** file. Then, if necessary, the subroutine checks a file in the user's home directory at the server called **/$HOME/.rhosts** for a host and remote user ID.

RootUser
: Specifies a value to indicate whether the effective user ID of the calling process is that of a root user. A value of 0 (zero) indicates the process does not have a root user ID. A value of 1 (one) indicates that the process has local root user privileges, and the **/etc/host.equiv** file is not checked.

RemoteUser
: Points to a user name that is valid at the remote host. Any valid user name can be specified.

LocalUser
: Points to a user name that is valid at the local host. Any valid user name can be specified.

Return Values

The **ruserok** subroutine returns a 0 (zero), if the subroutine successfully locates the name specified by the *Host* parameter in the **/etc/hosts.equiv** file or the IDs specified by the *Host* and *RemoteUser* parameters are found in the **/$HOME/.rhosts** file.

If the name specified by the *Host* parameter was not found, the **ruserok** subroutine returns a –1 (negative one).

Implementation Specifics

The **ruserok** subroutine is part of AIX Base Operating System (BOS) Runtime.

All applications containing the **ruserok** subroutine must be compiled with **_BSD** defined. In addition, when applicable, all socket applications must include the BSD library **libbsd**.

Files

/etc/services	Contains service names.
/etc/host.equiv	Specifies foreign host names.
/$HOME/.rhosts	Specifies the remote users of a local user account.

Related Information

Additional remote command execution subroutines are the **rcmd** subroutine and **rresvport** subroutine.

subroutines to get and set the host name, respectively, are the **gethostname** subroutine and **sethostname** subroutine.

TCP/IP Interface Program commands are the **rlogind** command and **rshd** command.

Sockets Overview in *Communications Programming Concepts.*

send Subroutine

Purpose

Sends messages from a connected socket.

Syntax

#include <sys/types.h>

#include <sys/socketvar.h>

#include <sys/socket.h>

int send (*Socket*, *Message*, *Length*, *Flags*)
int *Socket*;
char **Message*;
int *Length*, *Flags*;

Description

The **send** subroutine sends a message only when the socket is connected. The **sendto** and **sendmsg** subroutines can be used with unconnected or connected sockets.

To broadcast on a socket, first issue a **setsockopt** subroutine using the **SO_BROADCAST** option to gain broadcast permissions.

Specify the length of the message with the *Length* parameter. If the message is too long to pass through the underlying protocol, the system returns an error and does not transmit the message.

No indication of failure to deliver is implied in a **send** subroutine. A return value of –1 (negative one) indicates some locally detected errors.

If no space for messages is available at the sending socket to hold the message to be transmitted, the **send** subroutine blocks unless the socket is in a nonblocking I/O mode. Use the **select** subroutine to determine when it is possible to send more data.

Parameters

Socket — Specifies the unique name for the socket.

Message — Points to the address of the message to send.

Length — Specifies the length of the message in bytes.

Flags — Allows the sender to control the transmission of the message. The *Flags* parameter to send a call is formed by logically ORing one or both of the values shown in the following list:

MSG_OOB — Processes out–of–band data on sockets that support **SOCK_STREAM** communication.

MSG_DONTROUTE — Sends without using routing tables.

The /**sys/socket.h** file defines the *Flags* values.

Return Value

Upon successful completion, the **send** subroutine returns the number of characters sent.

If the **send** subroutine fails, the subroutine handler performs the following functions:

- Returns a value of –1 (negative one) to the calling program
- Moves an error code, indicating the specific error, into the global variable **errno**

Error Codes

The subroutine fails if any one or of the following errors occurs:

EBADF	The *Socket* parameter is not valid.
ENOTSOCK	The *Socket* parameter refers to a file, not a socket.
EFAULT	The *Address* parameter is not in a writable part of the user address space.
EMSGSIZE	The message is too large be sent all at once, as the socket requires.
EWOULDBLOCK	The socket is marked nonblocking, and no connections are present to be accepted.

Implementation Specifics

The **send** subroutine is part of AIX Base Operating System (BOS) Runtime.

All applications containing the **send** subroutine must be compiled with **_BSD** defined. In addition, when applicable, all socket applications must include the BSD library **libbsd**.

Files

/usr/include/sys/socket.h	Contains socket definitions.
/usr/include/sys/types.h	Contains definitions of unsigned data type.

Related Information

Subroutines to receive and send data over sockets are the **recv** subroutine, **recvfrom** subroutine, **recvmsg** subroutine, **sendmsg** subroutine, **sendto** subroutine, and **shutdown** subroutine.

Socket creation and connection subroutines are the **connect** subroutine and **socket** subroutine.

Subroutines for monitoring data broadcasts and manipulating socket options are the **getsockopt** subroutine, **select** subroutine, and **setsockopt** subroutine.

Sockets Overview, Understanding Socket Data Transfer in *Communications Programming Concepts*.

sendmsg Subroutine

Purpose

Sends a message from a socket using a message structure.

Syntax

#include<sys/types.h>

#include <sys/socketvar.h>

#include <sys/socket.h>

int sendmsg (*Socket*, *Message*, *Flags*)
int *Socket*;
struct msghdr *Message*[ ];
int *Flags*;

Description

The **sendmsg** subroutine sends messages through connected or unconnected sockets using the **msghdr** message structure. The **/sys/socket.h** file contains the **msghdr** structure and defines the structure members.

To broadcast on a socket, the application program must first issue a **setsockopt** subroutine using the **SO_BROADCAST** option to gain broadcast permissions.

Parameters

Socket Specifies the socket descriptor.

Message Points to the **msghdr** message structure containing the message to be sent, the message length, the destination address, and the size of the destination address.

Flags Allows the sender to control the message transmission. The **/sys/socket.h** file contains the *Flags* values. The *Flags* value to send a call is formed by logically ORing one or both of the following values:

MSG_OOB Processes out–of–band data on sockets that support **SOCK_STREAM**.

Note: The following value is not for general use. It is an administrative tool used for debugging or for routing programs.

MSG_DONTROUTE Sends without using routing tables.

Return Value

Upon successful completion, the **sendmsg** subroutine returns the number of characters sent.

If the **sendmsg** subroutine fails, the system handler performs the following functions:

- Returns a value of –1 (negative one) to the calling program
- Moves an error code, indicating the specific error, into the global variable **errno**

Error Codes

The **sendmsg** subroutine fails if any one of the following errors occurs:

EBADF	The *Socket* parameter is not valid.
ENOTSOCK	The *Socket* parameter refers to a file, not a socket.
EMSGSIZE	The message is too large to be sent all at once, as the socket requires.
EWOULDBLOCK	The socket is marked nonblocking, and no connections are present to be accepted.

Implementation Specifics

The **sendmsg** subroutine is part of AIX Base Operating System (BOS) Runtime.

All applications containing the **sendmsg** subroutine must be compiled with **_BSD** defined. In addition, when applicable, all socket applications must include the BSD library **libbsd**.

Files

/usr/include/sys/socket.h	Contains socket definitions.
/usr/include/sys/socketvar.h	Defines the kernel structure per socket and contains buffer queues.
/usr/include/sys/types.h	Contains definitions of unsigned data types.

Related Information

Subroutines to receive and send data over sockets are the **recv** subroutine, **recvfrom** subroutine, **recvmsg** subroutine, **send** subroutine, **sendto** subroutine, and **shutdown** subroutine.

Subroutines are to create sockets, the **socket** subroutine; to monitor data broadcasts, the **select** subroutine; to manipulate socket options, the **getsockopt** subroutine and **setsockopt** subroutine.

Sockets Overview, Understanding Socket Data Transfer in *Communications Programming Concepts*.

sendto Subroutine

Purpose

Sends messages through a socket.

Syntax

#include <sys/types.h>

#include <sys/socket.h>

int sendto (*Socket*, *Message*, *Length*, *Flags*, *To*, *ToLength*)
int *Socket*;
char **Message*;
int *Length, Flags*;
struct sockaddr **To*;
int *ToLength*;

Description

The **sendto** subroutine allows an application program to sends messages through an unconnected sockets by specifying a destination address.

To broadcast on a socket, first issue a **setsockopt** subroutine using the **SO_BROADCAST** option to gain broadcast permissions.

Provide the address of the target using the *To* parameter. Specify the length of the message with the *Length* parameter. If the message is too long to pass through the underlying protocol, the error **EMSGSIZE** is returned and the message is not transmitted.

If the **sending** socket has no space to hold the message to be transmitted, the **sendto** subroutine blocks the message unless the socket is in a nonblocking I/O mode.

Use the **select** subroutine to determine when it is possible to send more data.

Parameters

Socket Specifies the unique name for the socket.

Message Specifies the address containing the message to be sent.

Length Specifies the size of the message in bytes.

Flags Allows the sender to control the message transmission. The *Flags* value to send a call is formed by logically ORing one or both of the following values:

MSG_OOB Processes out–of–band data on sockets that support **SOCK_STREAM**.

Note: The following value is not for general use.

MSG_DONTROUTE Sends without using routing tables.

The /**sys/socket.h** file defines the *Flags* arguments.

To	Specifies the destination address for the message. The destination address is a **sockaddr** structure defined in the **/sys/socket.h** header file.
ToLength	Specifies the size of the destination address.

Return Value

Upon successful completion, the **sendto** subroutine returns the number of characters sent.

If the **sendto** subroutine fails, the system returns a value of –1 (negative one), and **errno** is set to indicate the error.

Error Codes

The subroutine fails if any one of the following errors occurs:

EBADF	The *Socket* parameter is not valid.
ENOTSOCK	The *Socket* parameter refers to a file, not a socket.
EFAULT	The *Address* parameter is not in a writable part of the user address space.
EMSGSIZE	The message is too large be sent all at once, as the socket requires.
EWOULDBLOCK	The socket is marked nonblocking, and no connections are present to be accepted.

Examples

1. The Sending UNIX Domain Datagrams program fragment illustrates the use of the **sendto** subroutine.
2. The Sending Internet Domain Datagrams program fragment illustrates the use of the **sendto** subroutine.

Implementation Specifics

The **sendto** subroutine is part of AIX Base Operating System (BOS) Runtime.

All applications containing the **sendto** subroutine must be compiled with **_BSD** defined. In addition, when applicable, all socket applications must include the BSD library **libbsd**.

Files

/usr/include/sys/socket.h	Contains socket definitions.
/usr/include/sys/socketvar.h	Defines the kernel structure per socket and contains buffer queues.
/usr/include/sys/types.h	Contains definitions of unsigned data types.

Related Information

Subroutines to receive and send data over sockets are the **recv** subroutine, **recvfrom** subroutine, **recvmsg** subroutine, **send** subroutine, **sendmsg** subroutine, and **shutdown** subroutine.

Subroutines are: to create sockets, the **socket** subroutine; to monitor data broadcasts, **select** subroutine; to manipulate socket options, the **getsockopt** subroutine and **setsockopt** subroutine.

Sockets Overview, Understanding Socket Data Transfer in *Communications Programming Concepts*.

setdomainname Subroutine

Purpose

Sets the name of the current domain.

Syntax

int setdomainname (*name*, *namelen*)

char **name*;
int *namelen*;

Description

The **setdomainname** subroutine sets the name of the domain for the host machine. It is normally used when the system is bootstrapped. You must have root user authority to run this subroutine.

The purpose of domains is to enable two distinct networks that may have host names in common to merge. Each network would be distinguished by having a different domain name. At the current time, only the NIS and the **sendmail** command make use of domains

Note: Domain names are restricted to 64 characters.

Parameters

name	Specifies the domain name to be set.
namelen	Specifies the size of the array pointed to by the *name* parameter.

Return Values

If the call suceeds, a value of 0 (zero) is returned. If the call fails, a value of –1 is returned and an error code is placed in the global location errno.

Error Codes

The following error may be returned by this subroutine:

EFAULT	The *name* parameter gave an invalid address.
EPERM	The caller was not the root user.

Implementation Specifics

The **setdomainname** subroutine is part of AIX Base Operating System (BOS) Runtime.

All applications containing the **setdomainname** subroutine must be compiled with **_BSD** defined. In addition, when applicable, all socket applications must include the BSD library **libbsd**.

Related Information

The **getdomainname** subroutine, **gethostname** subroutine, **sethostname** subroutine.

Sockets Overview in *Communications Programming Concepts*.

sethostent Subroutine

Purpose

Opens network host file.

Library

(libc.a)

Syntax

#include <netdb.h>

void sethostent (*StayOpen*)
int *StayOpen*;

Description

The **sethostent** (set host entry) subroutine opens the **/etc/hosts** file and resets the file marker to the beginning of the file.

Passing a nonzero value to the *StayOpen* parameter establishes a connection with a name server and allows a client process to retrieve one entry at a time from the **/etc/hosts** file. The client process can close the connection with the **endhostent** subroutine.

Parameter

StayOpen — Contains a value used to indicate when to close the host file.

Specifying a value of 0 (zero) closes the **/etc/hosts** file after each call to the **gethostbyname** or **gethostbyaddr** subroutine.

Specifying a nonzero value allows the **/etc/hosts** file to remain open after each call.

Return Values

If an error occurs or if the end of the file is reached, the **sethostent** subroutine returns a **NULL** (**0**) pointer to the calling program. The subroutine handler moves an error code, indicating the specific error, into the **h_errno** variable. The calling program must examine **h_errno**, to determine the error.

Error Code

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

NO_RECOVERY — This error code indicates an unrecoverable error.

Implementation Specifics

The **sethostent** subroutine is part of AIX Base Operating System (BOS) Runtime.

All applications containing the **sethostent** subroutine must be compiled with **_BSD** defined. In addition, when applicable, all socket applications must include the BSD library **libbsd**.

Files

/etc/hosts	Contains the host name database.
/etc/resolv.conf	Contains the name server and domain name.
/usr/include/netdb.h	Contains the network database structures.

Related Information

Additional host information retrieval subroutines are the **endhostent** subroutine, **gethostbyaddr** subroutine, and **gethostbyname** subroutine.

Sockets Overview, Understanding Network Address Translation in *Communications Programming Concepts*.

sethostid Subroutine

Purpose

Sets the unique identifier of the current host.

Syntax

intsethostid (*HostID*)

int *HostID*;

Description

The **sethostid** subroutine allows a calling process with a root user ID to set a new 32–bit identifier for the current host. The **sethostid** subroutine enables an application program to reset the host ID.

Parameters

HostID	Specifies the unique 32–bit identifier for the current host.

Return Value

Upon successful completion, the **sethostid** subroutine returns a value of 0 (zero).

If the **sethostid** subroutine fails, the subroutine handler performs the following functions:

- Returns a value of –1 (negative one) to the calling program
- Moves an error code, indicating the specific error, into the global variable **errno**

Error Code

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

EPERM	The calling process did not have an effective user ID of root user.

Implementation Specifics

The **sethostid** subroutine is part of AIX Base Operating System (BOS) Runtime.

All applications containing the **sethostid** subroutine must be compiled with **_BSD** defined. In addition, when applicable, all socket applications must include the BSD library **libbsd**.

Related Information

Socket subroutines to obtain host names, IDs, and socket names, respectively, are the **gethostid** subroutine, **gethostname** subroutine, and **getsockname** subroutine.

sethostname Subroutine

Purpose

Sets the name of the current host.

Syntax

int sethostname (*Name*, *NameLength*)
char **Name*;
int *NameLength*;

Description

The **sethostname** subroutine sets the name of a host machine. Only programs with a root user ID can use this subroutine.

The **sethostname** subroutine allows a calling process with root user authority to set the internal host name of a machine on a network.

Parameters

Name	Returns the address of an array of bytes where the host name is stored.
NameLength	Returns an integer that specifies the length of the *Name* array.

Return Values

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

If the **sethostname** subroutine fails, the subroutine handler performs the following functions:

- Returns a value of −1 (negative one) to the calling program
- Moves an error code, indicating the specific error, into the global variable **errno**

Error Codes

The **sethostname** subroutine fails if any one of the following errors occur:

EFAULT	The *Name* parameter or *NameLength* parameter gives an address that is not valid.
EPERM	The calling process did not have an effective root user ID.

Implementation Specifics

The **sethostname** subroutine is part of AIX Base Operating System (BOS) Runtime.

All applications containing the **sethostname** subroutine must be compiled with **_BSD** defined. In addition, when applicable, all socket applications must include the BSD library **libbsd**.

Related Information

Socket subroutines to obtain and set the host ID are the **gethostid** subroutine and **sethostid** subroutine.

The socket subroutine to obtain the host name is the **gethostname** subroutine.

Sockets Overview, Understanding Network Address Translation in *Communications Programming Concepts*.

setnetent Subroutine

Purpose

Opens and rewinds the networks file.

Library

(libc.a)

Syntax

#include <netdb.h>

void setnetent (*StayOpen*)
int *StayOpen*;

Description

The **setnetent** (set network entry) subroutine opens the **/etc/networks** file and sets the file marker at the beginning of the file.

Parameter

StayOpen Contains a value used to indicate when to close the networks file.

Specifying a value of 0 (zero) closes the networks file after each call to the **getnetent** subroutine.

Specifying a nonzero values leaves the **/etc/networks** file open after each call.

Return Values

If an error occurs or the end of the file is reached, the **setnetent** subroutine returns a **NULL** pointer.

Implementation Specifics

The **setnetent** subroutine is part of AIX Base Operating System (BOS) Runtime.

All applications containing the **setnetent** subroutine must be compiled with **_BSD** defined. In addition, when applicable, all socket applications must include the BSD library **libbsd**.

Files

/etc/networks	Contains official network names.
/usr/include/netdb.h	Contains the network database structures.

Related Information

Additional network information retrieval subroutines are the **endnetent** subroutine, **getnetbyaddr** subroutine, **getnetbyname** subroutine, and **getnetent** subroutine.

Sockets Overview, Understanding Network Address Translation in *Communications Programming Concepts*.

setprotoent Subroutine

Purpose

Opens and rewinds the /**etc/protocols** file.

Library

(libc.a)

Syntax

#include <netdb.h>

void setprotoent (*StayOpen*)
int *StayOpen*;

Description

The **setprotoent** (set protocol entry) subroutine opens the /**etc/protocols** file and sets the file marker to the beginning of the file.

Parameter

StayOpen Indicates when to close the protocols file.

Specifying a value of 0 (zero) closes the file after each call to **getprotoent**.

Specifying a nonzero value allows the /**etc/protocols** file to remain open after each subroutine.

Return Value

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

Implementation Specifics

The **setprotoent** subroutine is part of AIX Base Operating System (BOS) Runtime.

All applications containing the **setprotoent** subroutine must be compiled with **_BSD** defined. In addition, when applicable, all socket applications must include the BSD library **libbsd**.

Files

/etc/protocols	Contains the protocol names.
/usr/include/netdb.h	Contains the network database structures.

Related Information

Additional protocol information retrieval subroutines are the **endprotoent** subroutine, **getprotobynumber** subroutine, **getprotobyname** subroutine, and **getprotoent** subroutine.

Sockets Overview, Understanding Network Address Translation in *Communications Programming Concepts.*

setservent Subroutine

Purpose

Gets service file entry.

Library

(libc.a)

Syntax

#include <netdb.h>

void setservent (*StayOpen*)
int *StayOpen*;

Description

The **setservent** (set service entry) subroutine opens the **/etc/services** file and sets the file marker at the beginning of the file.

Parameters

StayOpen Indicates when to close the services file.

Specifying a value of 0 (zero) closes the file after each call to the **getservent** subroutine.

Specifying a nonzero value allows the file to remain open after each call.

Return Value

If an error occurs or the end of the file is reached, the **setservent** subroutine returns a **NULL** (**0**) pointer.

Implementation Specifics

The **setservent** subroutine is part of AIX Base Operating System (BOS) Runtime.

All applications containing the **setservent** subroutine must be compiled with **_BSD** defined. In addition, when applicable, all socket applications must include the BSD library **libbsd.**

Files

/etc/services	Contains service names.
/usr/include/netdb.h	Contains network database structures.

Related Information

Additional service information retrieval subroutines are the **endservent** subroutine, **getservbyport** subroutine, **getservbyname** subroutine,and **getservent** subroutine.

Protocol information retrieval subroutines are the **endprotoent** subroutine, **getprotobyname** subroutine, **getprotobynumber** subroutine, **getprotoent** subroutine, and **setprotoent** subroutine.

Sockets Overview, Understanding Network Address Translation in *Communications Programming Concepts*.

setsockopt Subroutine

Purpose

Sets socket options.

Syntax

#include <sys/types.h>
#include <sys/socket.h>
#include <sys/sockevar.h>

int setsockopt (*Socket*, *Level*, *OptionName*, *OptionValue*, *OptionLength*)
int *Socket*, *Level*, *OptionName*;
char **OptionValue*;
int *OptionLength*;

Description

The **setsockopt** subroutine sets options associated with a socket. Options may exist at multiple protocol levels. The options are always present at the uppermost socket level.

The **setsockopt** subroutine provides an application program with the means to control a socket communication. An application program can use the **setsockopt** subroutine to enable debugging at the protocol level, allocate buffer space, control timeouts, or permit socket data broadcasts. The /**sys/socket.h** file defines all the options available to the **setsockopt** subroutine.

When setting socket options, specify the protocol level at which the option resides and the name of the option.

Use the parameters *OptionValue* and *OptionLength* to access option values for the **setsockopt** subroutine. These parameters identify a buffer in which the value for the requested option or options is returned.

Parameters

Socket Specifies the unique socket name.

Level Specifies the protocol level at which the option resides. To set options at:

- Socket level: specify *Level* as **SOL_SOCKET**.
- Other levels: supply the appropriate protocol number for the protocol controlling the option. For example, to indicate that an option will be interpreted by the TCP protocol, set *Level* to the protocol number of TCP, as defined in the **netinet/in.h** file.

OptionName Specifies the option to set. The *OptionName* parameter and any specified options are passed uninterpreted to the appropriate protocol module for interpretation. The **sys/socket.h** header file defines the socket level options. The socket level options can be enabled or disabled; they operate in a toggle fashion. The options are:

SO_DEBUG Turns on recording of debugging information. This option enables or disables debugging in the underlying protocol modules.

SO_REUSEADDR Specifies that the rules used in validating addresses supplied by a **bind** subroutine should allow reuse of local addresses.

SO_KEEPALIVE Keeps connections active. Enables the periodic transmission of messages on a connected socket. If the connected socket fails to respond to these messages, the connection is broken and processes using that socket are notified with a **SIGPIPE** signal.

SO_DONTROUTE Does not apply routing on outgoing messages. Indicates that outgoing messages should bypass the standard routing facilities. Instead, they are directed to the appropriate network interface according to the network portion of the destination address.

SO_BROADCAST Permits sending of broadcast messages.

SO_LINGER Lingers on a **close** subroutine if data is present. This option controls the action taken when unsent messages queue on a socket and a **close** subroutine is performed. It uses a **struct** *linger* parameter defined in the **sys/socket.h** file. The parameter specifies the state of the option and linger interval. Specify the linger interval by using the **setsockopt** subroutine when requesting **SO_LINGER**.

If **SO_LINGER** is set, the system blocks the process during the **close** subroutine until it can transmit the data or until the time expires. If **SO_LINGER** is not specified and a **close** subroutine is issued, the system handles the call in a way that allows the process to continue as quickly as possible.

SO_OOBINLINE Leaves received out–of–band data (data marked urgent) in line.

SO_SNDBUF	Sets send buffer size.
SO_RCVBUF	Sets receive buffer size.
SO_SNDLOWAT	Sets send low–water mark.
SO_RCVLOWAT	Sets receive low–water mark.
SO_SNDTIMEO	Sets send time out.
SO_RCVTIMEO	Sets receive time out.
SO_ERROR	Sets the retrieval of error status and clear
SO_TYPE	Sets the retrieval of a socket type.

OptionValue The *OptionValue* parameter takes an *Int* parameter. To enable a Boolean option, set the *OptionValue* parameter to a nonzero value. To disable an option, set the *OptionValue* parameter to 0 (zero).

The following options enable and disable in the same manner:

SO_DEBUG

SO_REUSEADDR

SO_KEEPALIVE

SO_DONTROUTE

SO_BROADCAST

SO_OOBINLINE

SO_LINGER.

OptionLength The *OptionLength* parameter initially contains the size of the buffer pointed to by the *OptionValue* parameter. On return, the *OptionLength* parameter is modified to indicate the actual size of the value returned. If no option value is supplied or returned, the *OptionValue* parameter can be 0 (zero).

Options at other protocol levels vary in format and name.

Return Value

Upon successful completion, a value of 0 (zero) is returned.

If the **setsockopt** subroutine fails, the subroutine handler performs the following functions:

- Returns a value of –1 (negative one) to the calling program
- Moves an error code, indicating the specific error, into the global variable **errno**

Error Codes

The **setsockopt** subroutine fails if any one of the following errors occur:

EBADF	The *Socket* parameter is not valid.
ENOTSOCK	The *Socket* parameter refers to a file, not a socket.
ENOPROTOOPT	The option is unknown.
EFAULT	The *Address* parameter is not in a writable part of the user address space.

Example

1. To mark a socket for broadcasting:

```
int on=1;
setsockopt(s, SOL_SOCKET, SO_BROADCAST, &on, sizeof(on));
```

Implementation Specifics

The **setsockopt** subroutine is part of AIX Base Operating System (BOS) Runtime.

All applications containing the **setsockopt** subroutine must be compiled with **_BSD** defined. In addition, when applicable, all socket applications must include the BSD library **libbsd**.

Files

/usr/include/sys/socket.h	Contains socket definitions.
/usr/include/sys/socketvar.h	Defines the kernel structure per socket and contains buffer queues.
/usr/include/sys/types.h	Contains definitions of unsigned data types.

Related Information

The socket subroutine used for retrieving socket option data is the **getsockopt** subroutine.

subroutines used for creating and naming sockets are, respectively, the **bind** subroutine and **socket** subroutine.

Socket subroutines used to retrieve protocol data are the **endprotoent** subroutine, **getprotobynumber** subroutine, **getprotoent** subroutine, and **setprotoent** subroutine.

Sockets Overview, Understanding Socket Options in *Communications Programming Concepts*.

shutdown Subroutine

Purpose

Shuts down all socket send and receive operations.

Syntax

intshutdown (*Socket*, *How*)
int *Socket*, *How*;

Description

The **shutdown** subroutine disables all receive and send operations on the specified socket.

Parameters

Socket	Specifies the unique name of the socket
How	Specifies the type of subroutine shutdown. Use the following values:

0	To disable further receive operations.
1	To disable further send operations.
2	To disable further send operations and receive operations.

Return Values

Upon successful completion, a value of 0 (zero) is returned.

If the **shutdown** subroutine fails, the subroutine handler performs the following functions:

- Returns a value of –1 (negative one) to the calling program
- Moves an error code, indicating the specific error, into the global variable **errno**

Error Codes

The **shutdown** subroutine fails if any one of the following errors occurs:

EBADF	The *Socket* parameter is not valid.
ENOTSOCK	The *Socket* parameter refers to a file, not a socket.
ENOTCONN	The socket is not connected.

Implementation Specifics

The **shutdown** subroutine is part of AIX Base Operating System (BOS) Runtime.

All applications containing the **shutdown** subroutine must be compiled with **_BSD** defined. In addition, when applicable, all socket applications must include the BSD library **libbsd**.

Files

/usr/include/sys/socket.h	Contains socket definitions.
/usr/include/sys/types.h	Contains definitions of unsigned data types.

Related Information

Subroutines to receive and send data over sockets are the **read** subroutine, **recv** subroutine, **recvfrom** subroutine, **recvmsg** subroutine, **send** subroutine, **sendto** subroutine, and **write** subroutine.

Subroutines to create sockets, monitor data broadcasts, and manipulate socket options are the **getsockopt** subroutine, **select** subroutine, **setsockopt** subroutine, and **socket** subroutine.

socket Subroutine

Purpose

Creates an end point for communication and returns a descriptor.

Syntax

#include <sys/types.h>

#include <sys/socket.h>

#include <sys/socketvar.h>

int socket (*AddressFamily*, *Type*, *Protocol*)

int *Domain*, *Type*, *Protocol*;

Description

The **socket** subroutine creates a socket in the specified *AddressFamily* and of the specified *Type*. A protocol can be specified or assigned by the system. If the protocol is left unspecified (a value of 0), the system selects an appropriate protocol from those protocols in the address family that can be used to support the requested socket type.

The **socket** subroutine returns a descriptor (an integer) that can be used in later subroutines that operate on sockets.

Socket level options control socket operations. The **getsockopt** and **setsockopt** subroutines are used to get and set these options, which are defined in the **sys/socket.h** file.

Parameters

AddressFamily Specifies an address family with which addresses specified in later socket operations should be interpreted. The **/sys/socket.h** file contains the definitions of the address families. Commonly used families are:

AF_UNIX AIX path names

AF_INET ARPA Internet addresses.

Type Specifies the semantics of communication. The **/sys/socket.h** file defines the socket types. AIX supports the following types:

SOCK_STREAM Provides sequenced, two–way byte streams with a transmission mechanism for out–of–band data.

SOCK_DGRAM Provides datagrams, which are connectionless messages of a fixed maximum length (usually short).

SOCK_RAW Provides access to internal network protocols and interfaces. This type of socket is available only to the root user.

Protocol Specifies a particular protocol to be used with the socket. Specifying a *Protocol* of 0 (zero) causes the **socket** subroutine to default to the typical protocol for the requested type of returned socket.

Return Value

Upon successful completion, the **socket** subroutine returns an integer (the socket descriptor).

If the **socket** subroutine fails, the subroutine handler performs the following functions:

- Returns a value of –1 (negative one) to the calling program
- Moves an error code, indicating the specific error, into the global variable **errno**.

Error Codes

The **socket** subroutine fails if any one of the following errors occurs:

EAFNOSUPPORT	The addresses in the specified address family cannot be used with this socket.
ESOCKTNOSUPPORT	The socket in the specified address family is not supported.
EMFILE	The per–process descriptor table is full.
ENOBUFS	Insufficient resources were available in the system to complete the call.

Example

1. The following program fragment illustrates the use of the **socket** subroutine to create a datagram socket for on–machine use.

```
s = socket(AF_UNIX, SOCK_DGRAM,0);
```

Implementation Specifics

The **socket** subroutine is part of AIX Base Operating System (BOS) Runtime.

All applications containing the **socket** subroutine must be compiled with **_BSD** defined. In addition, when applicable, all socket applications must include the BSD library **libbsd**.

Files

/usr/include/sys/socket.h	Contains socket definitions.
/usr/include/sys/socketvar.h	Defines the kernel structure per socket and contains buffer queues.
/usr/include/sys/types.h	Contains definitions for unsigned data types.

Related Information

Other socket creation and connection subroutines are the **accept** subroutine, **bind** subroutine, **connect** subroutine, **listen** subroutine, and **socketpair** subroutine.

Subroutines for retrieving socket information and setting socket options are the **getsockname** subroutine, **getsockopt** subroutine, and **setsockopt** subroutine.

Subroutines for receiving and sending data over sockets are the **recv** subroutine, **recvfrom** subroutine, **recvmsg** subroutine, **send** subroutine, **sendto** subroutine, **sendmsg** subroutine, and **shutdown** subroutine.

The **ioctl** subroutine and **select** subroutine.

Sockets Overview, Understanding Socket Creation in *Communications Programming Concepts*.

socketpair Subroutine

Purpose

Creates a pair of connected sockets.

Syntax

#include <sys/types.h>
#include <sys/socket.h>
#include <sys/socketvar.h>

socketpair (*Domain*, *Type*, *Protocol*, *SocketVector*)
int *Domain*, *Type*, *Protocol*;
int *SocketVector*[**2**];

Description

The **socketpair** subroutine creates an unnamed pair of connected sockets in a specified *Domain*, of a specified *Type*, and using the optionally specified *Protocol*. The two sockets are identical.

Note: Create sockets with this subroutine only in the AF_UNIX domain.

The descriptors used in referencing the new sockets are returned in *SocketVector*[0] and *SocketVector*[1].

The /**sys/socket.h** file contains the definitions for socket domains, types, and protocols.

Parameters

Domain	Specifies the communications domain within which the sockets are created. This subroutine does not create sockets in the Internet domain.
Type	Specifies the communications method, whether **SOCK_DGRAM** or **SOCK_STREAM**, that the socket uses.
Protocol	Points to an optional identifier used to specify which standard set of rules (such as UDP/IP and TCP/IP) governs the transfer of data.
SocketVector	Points to a two–element vector that contains the integer descriptors of a pair of created sockets.

Return Value

Upon successful completion, the **socketpair** subroutine returns a value of 0 (zero).

If the **socketpair** subroutine fails, the subroutine handler performs the following functions:

- Returns a value of –1 (negative one) to the calling program
- Moves an error code, indicating the specific error, into the global variable **errno.**

Error Codes

The **socketpair** subroutine fails for any one of the following errors occurs:

EMFILE	This process has too many descriptors in use.
EAFNOSUPPORT	The addresses in the specified address family cannot be used with this socket.
EPROTONOSUPPORT	The specified protocol cannot be used on this system.
EOPNOSUPPORT	The specified protocol does not allow creation of socket pairs.
EFAULT	The *SocketVector* parameter is not in a writable part of the user address space.

Implementation Specifics

The **socketpair** subroutine is part of AIX Base Operating System (BOS) Runtime.

All applications containing the **socketpair** subroutine must be compiled with **_BSD** defined. In addition, when applicable, all socket applications must include the BSD library **libbsd**.

Files

/usr/include/sys/socket.h	Contains socket definitions.
/usr/include/sys/socketvar.h	Defines the kernel structure per socket and contains buffer queues.
/usr/include/sys/types.h	Contains definitions for unsigned data types.

Related Information

An additional socket creation method is the **socket** subroutine.

Sockets Overview, Understanding Socket Creation in *Communications Programming Concepts*.

X.25 Application

x25_ack Subroutine

Purpose

Acknowledges data received with the D-bit set.

Library

The X.25 Communications Library (**libx25s.a**)

C Syntax

int x25_ack(
int conn_id
);

Description

The **x25_ack** subroutine sends an acknowledgement for the data packet most recently received with the D-bit set for the call specified by **conn_id**.

Control is returned to the calling application when the adapter has queued the packet for transmission.

Parameter

conn_id	Connection identifier of the call.

Return Value

If successful, **x25_ack** returns a value of 0. If an error occurs, **x25_ack** returns –1 and sets **x25_errno** to one of the error codes shown below.

Error Codes

X25BADCONNID, **X25NOACKREQ**, **X25NOCARD**, **X25NOLINK**, **X25NOTINIT**, **X25PROTOCOL**, **X25RESETCLEAR**, **X25SYSERR**, **X25TRUNCTX**.

If **x25_errno** is set to **X25SYSERR**, **errno** is set to one of the following values:

EINTR, **EIO**, **ENOSPC**.

Implementation Specifics

This subroutine is part of X.25 Application in AIX BOS Extensions 2.

Related Information

The **x25_send** subroutine.

x25_call Subroutine

Purpose

Makes an X.25 call, by setting up a switched virtual circuit (SVC).

Library

The X.25 Communications Library (**libx25s.a**)

C Syntax

int x25_call(
struct cb_call_struct *cb_call,
int ctr_id
);

Description

The **x25_call** subroutine sets up a switched virtual circuit (SVC) for the X.25 port specified in **cb_call_struct**, for an X.25 call between the calling address and called address, also specified in **cb_call_struct**.

Control is returned to the application as soon as the call-request packet has been transmitted, but the SVC is not actually established until a call-connected packet is received (using **x25_receive**).

Optional facilities, such as fast-select calls, can be requested by entering the correct values in **cb_fac_struct**. If the facilities requested are not allowed by the network, the call is cleared and an appropriate error code is made available in **cb_clear_struct**, which can be received using **x25_receive**.

Parameters

cb_call Pointer to **cb_call_struct**.

ctr_id Identifier of a counter allocated by a previous **x25_ctr_get**.

Return Value

If successful, **x25_call** returns the connection identifier to be used by other subroutines for the duration of the call. If an error occurs, or the call is cleared, **x25_call** returns –1 and sets **x25_errno** to one of the error codes shown below.

Error Codes

X25CALLED, **X25CALLING**, **X25INVCTR**, **X25INVFAC**, **X25LONG**, **X25NOCARD**, **X25NOLINK**, **X25NOSUCHLINK**, **X25NOTINIT**, **X25PROTOCOL**, **X25SYSERR**, **X25TOOMANYVCS**, **X25TRUNCTX**.

If **x25_errno** is set to **X25SYSERR**, **errno** is set to one of the following values:

EINTR, **EIO**, **ENOSPC**.

Implementation Specifics

This subroutine is part of X.25 Application in AIX BOS Extensions 2.

Related Information

The **x25_call_accept** and **x25_call_clear** subroutines.

x25_call_accept Subroutine

Purpose

Accepts an incoming call.

Library

The X.25 Communications Library (**libx25s.a**)

C Syntax

int x25_call_accept(
int conn_id,
struct cb_call_struct *cb_call,
int ctr_id
);

Description

The **x25_call_accept** subroutine accepts an incoming call, by generating and sending a call-accepted packet. It then returns control to the application. If the facilities requested are not allowed by the network, the call is cleared and an appropriate error code is made available in a later **cb_clear_struct** control block.

Parameters

conn_id Connection identifier of the call

cb_call Pointer to the call control block, **cb_call_struct**.

ctr_id Identifier of a counter allocated by a previous **x25_ctr_get**, to be associated with this call.

Return Value

If successful, **x25_call_accept** returns a value of 0. If an error occurs, **x25_call_accept** returns –1 and sets **x25_errno** to one of the error codes shown below.

Error Codes

X25BADCONNID, X25CALLED, X25CALLING, X25INVCTR, X25INVFAC, X25LONG, X25NOCARD, X25NOLINK, X25NOTINIT, X25PROTOCOL, X25RESETCLEAR, X25SYSERR, X25TRUNCTX.

If **x25_errno** is set to **X25SYSERR**, **errno** is set to one of the following values:

EINTR, EIO, ENOSPC.

Implementation Specifics

This subroutine is part of X.25 Application in AIX BOS Extensions 2.

Related Information

The **x25_call** and **x25_call_clear** subroutines.

x25_call_clear Subroutine

Purpose

Clears a call.

Library

The X.25 Communications Library (**libx25s.a**)

C Syntax

int x25_call_clear (
int conn_id,
struct cb_clear_struct *cb_clear,
struct cb_msg_struct *cb_msg
);

Description

The **x25_call_clear** subroutine clears a call by generating and sending a clear-request packet. Control is not returned to the application until a clear-confirmation or a clear-indication packet has been received.

The effect of clearing a call is to disconnect a connected call, or to reject a call that has not been accepted.

Parameters

conn_id Connection identifier of the call

cb_clear Pointer to the clear structure, **cb_clear_struct**.

cb_msg Pointer to the message structure, **cb_msg_struct**. This structure is used to return information from the clear–confirmation packet. The application must interpret the appropriate structure to access the message. This structure is allocated by the API; it is the responsibility of the application to free this memory. If you set **cb_msg** value to NULL, no clear confirmation information is returned.

Return Value

If successful, **x25_call_clear** returns a value of 0. If an error occurs, **x25_call_clear** returns –1 and sets **x25_errno** to one of the error codes shown below.

Error Codes

X25BADCONNID, X25CALLED, X25CALLING, X25LONG, X25NOCARD, X25NOLINK, X25NOTINIT, X25PROTOCOL, X25SYSERR, X25RESETCLEAR, X25TRUNCTX.

If **x25_errno** is set to **X25SYSERR**, **errno** is set to one of the following values:

EINTR, EIO, ENOSPC.

Example

Terminate (clear) a call: example program svcxmit in *Communications Programming Concepts.*

Implementation Specifics

This subroutine is part of X.25 Application in AIX BOS Extensions 2.

Related Information

The **x25_call** and **x25_call_accept** subroutines.

x25_circuit_query Subroutine

Purpose

Returns configuration information about a virtual circuit.

Library

The X.25 Communications Library (**libx25s.a**)

C Syntax

struct cb_circuit_info_struct *x25_circuit_query(
int conn_id
);

Description

The **x25_circuit_query** subroutine returns the current information about the specified virtual circuit in **cb_circuit_info_struct**.

Parameter

conn_id	Connection identifier of the call currently using the virtual circuit.

Return Values

If successful, **x25_circuit_query** returns a pointer to **cb_circuit_info_struct**, the structure containing the information. Storage for this structure is allocated by the API; it is the responsibility of the application to free it. If an error occurs, **x25_circuit_query** returns NULL and sets **x25_errno** to one of the error codes shown below.

Error Codes

X25BADCONNID, X25NOLINK, X25NOTINIT, X25SYSERR.

If **x25_errno** is set to **X25SYSERR**, **errno** is set to:

ENOMEM

Example

1. Print current information for the virtual circuit identified by **conn_id**.

```
struct cb_circuit_info_struct *cct_ptr;
cct_ptr = x25_circuit_query(conn_id);
if (cct_ptr == NULL)
  (void)printf("Error %d from x25_circuit_query.",x25_errno);
else
{
  if (cct_ptr -> flags & X25FLG_LCN)
    (void)printf("Logical Channel Number (LCN) : %d\n",cct_ptr ->
lcn);
  if (cct_ptr -> flags & X25FLG_INCOMING_PACKET_SIZE)
    (void)printf("Incoming Packet Size : %d\n",
                 cct_ptr -> incoming_packet_size);
  if (cct_ptr -> flags & X25FLG_OUTGOING_PACKET_SIZE)
    (void)printf("Outgoing Packet Size : %d\n",
                 cct_ptr -> outgoing_packet_size);
  if (cct_ptr -> flags & X25FLG_INCOMING_THROUGHPUT_CLASS)
    (void)printf("Incoming throughput class : %d\n",
                 cct_ptr -> incoming_throughput_class);
  if (cct_ptr -> flags & X25FLG_OUTGOING_THROUGHPUT_CLASS)
    (void)printf("Outgoing throughput class : %d\n",
                 cct_ptr -> outgoing_throughput_class);
  if (cct_ptr -> flags & X25FLG_INCOMING_WINDOW_SIZE)
    (void)printf("Incoming window size : %d\n",
                 cct_ptr -> incoming_window_size);
  if (cct_ptr -> flags & X25FLG_OUTGOING_WINDOW_SIZE)
    (void)printf("Outgoing window size : %d\n",
                 cct_ptr -> outgoing_window_size);

  free(cct_ptr);
}
```

Implementation Specifics

This subroutine is part of X.25 Application in AIX BOS Extensions 2.

Related Information

The **x25_device_query** and **x25_link_query** subroutines.

x25_ctr_get Subroutine

Purpose

Gets a counter.

Library

The X.25 Communications Library (**libx25s.a**)

C Syntax

int x25_ctr_get(
void
);

Description

The **x25_ctr_get** subroutine allocates a counter whose value will be incremented whenever a message associated with it arrives and decremented whenever a message associated with it is received by an application.

Return Value

If successful, **x25_ctr_get** returns the counter identifier. If an error occurs, **x25_ctr_get** returns –1 and sets **x25_errno** to one of the error codes shown below.

Error Codes

X25NOCTRS, X25NOTINIT, X25SYSERR.

Example

Get a counter: example program svcxmit in *Communications Programming Concepts*.

Implementation Specifics

This subroutine is part of X.25 Application in AIX BOS Extensions 2.

Related Information

The **x25_ctr_remove**, **x25_ctr_test** and **x25_ctr_wait** subroutines.

x25_ctr_remove Subroutine

Purpose

Removes a counter.

Library

The X.25 Communications Library (**libx25s.a**)

C Syntax

int x25_ctr_remove(
int ctr_id
);

Description

The **x25_ctr_remove** subroutine removes the specified counter from the system. The counter identifier may be reused by a future **x25_ctr_get**. Only the application that requested the counter can remove the counter from the system. The counter cannot be removed if it has a non-zero value, which indicates that some data is still waiting to be read from an associated call.

Parameter

ctr_id Identifier of a counter allocated by a previous **x25_ctr_get**.

Return Values

If successful, **x25_ctr_remove** returns 0. If an error occurs, **x25_ctr_remove** returns –1 and sets **x25_errno** to one of the error codes shown below.

Error Codes

X25AUTHCTR, X25CTRUSE, X25INVCTR, X25NOTINIT, X25SYSERR.

Example

Remove a counter: example program svcxmit in *Communications Programming Concepts*.

Implementation Specifics

This command is part of X.25 Application in AIX BOS Extensions 2.

Related Information

The **x25_ctr_get**, **x25_ctr_test** and **x25_ctr_wait** subroutines.

x25_ctr_test Subroutine

Purpose

Returns the current value of a counter.

Library

The X.25 Communications Library (**libx25s.a**)

C Syntax

int x25_ctr_test(
int ctr_id
);

Description

The **x25_ctr_test** subroutine returns the current value of an active counter, so that it can be tested.

Parameter

ctr_id Counter identifier allocated by a previous **x25_ctr_get**.

Return Values

If successful, **x25_ctr_test** returns the current value of the counter. If an error occurs, **x25_ctr_test** returns –1 and sets **x25_errno** to one of the error codes shown below.

Error Codes

X25INVCTR, X25NOTINIT, X25SYSERR.

Example

To find out how many messages for a call are waiting to be received, assuming we have an array of information about calls in our application:

```
ctr_id = calls[i].counter_id;
number_of_messages = x25_ctr_test(ctr_id);
if (number_of_messages != -1)
  (void) printf("The number of messages waiting is %d",
                number_of_messages);
```

Note that the array used here is *not* part of the X.25 API.

Implementation Specifics

This subroutine is part of X.25 Application in AIX BOS Extensions 2.

Related Information

The **x25_ctr_get**, **x25_ctr_remove** and **x25_ctr_wait** subroutines.

x25_ctr_wait Subroutine

Purpose

Waits for counters to change in value.

Library

The X.25 Communications Library (**libx25s.a**)

C Syntax

int x25_ctr_wait(
int ctr_num,
struct ctr_array_struct ctr_array[ ]
);

Description

The **x25_ctr_wait** subroutine waits for the values of active counters to change in value. The process is suspended until the value of one of the counters is greater than the specified value. Setting this value in the application is optional, but recommended.

Parameters

ctr_num Number of elements in **ctr_array_struct**.

ctr_array An array of structures containing:

ctr_id Counter identifier allocated by a previous **x25_ctr_get**.

ctr_value The value that must be exceeded by this counter.

Return Values

If successful, **x25_ctr_wait** returns the **ctr_id** of the counter that satisfied the condition by exceeding the specified value. (If more than one counter exceeded its specified value, only one of the counter identifiers is returned.) If an error occurs, **x25_ctr_wait** returns –1 and sets **x25_errno** to one of the error codes shown below.

Error Codes

X25INVCTR, X25NOTINIT, X25SYSERR.

Examples

1. Wait for a call to be connected (or cleared): example program svcxmit in *Communications Programming Concepts*.
2. Wait for an incoming call: example program svcrcv in *Communications Programming Concepts*.
3. Wait for data (or some other message): example program svcrcv in *Communications Programming Concepts*.

Implementation Specifics

This command is part of X.25 Application in AIX BOS Extensions 2.

Related Information

The **x25_ctr_get**, **x25_ctr_remove** and **x25_ctr_test** subroutines.

x25_deafen Subroutine

Purpose

Turns off listening.

Library

The X.25 Communications Library (**libx25s.a**)

C Syntax

int x25_deafen(
int listen_id
);

Description

The **x25_deafen** subroutine turns off listening for incoming calls. In other words, it stops routing the calls that this application was listening for using the specified **listen_id**.

Parameter

listen_id The listen identifier returned from a previous **x25_listen**.

Return Values

If successful, **x25_deafen** returns 0. If an error occurs, **x25_deafen** returns –1 and sets **x25_errno** to one of the error codes shown below.

Error Codes

X25BADLISTENID, X25NOTINIT, X25SYSERR, X25TIMEOUT.

Example

Stop listening: example program svcrcv in *Communications Programming Concepts.*

Implementation Specifics

This subroutine is part of X.25 Application in AIX BOS Extensions 2.

Related Information

The **x25_listen** subroutine.

x25_device_query Subroutine

Purpose

Returns configuration information about a device.

Library

The X.25 Communications Library (**libx25s.a**)

C Syntax

struct cb_dev_info_struct *x25_device_query(
struct cb_link_name_struct *link_name
);

Description

The **x25_device_query** subroutine returns information about the X.25 adapter in **cb_dev_info_struct**.

The information returned is the information entered when you configured the adapter. Changes made to a particular switched virtual circuit (SVC) by requests entered in the facilities fields of X.25 API structures are *not* reflected by this subroutine; these values can be obtained by using the **x25_circuit_query** subroutine.

Parameter

link_name A pointer to **cb_link_name_struct**, which gives the name of the X.25 port.

Return Values

If successful, **x25_device_query** returns a pointer to **cb_dev_info_struct**, the structure containing the information. The storage for this structure is allocated by the API; it is the responsibility of the application to free it. If an error occurs, **x25_device_query** returns NULL and sets **x25_errno** to one of the error codes shown below.

Error Codes

X25NOTINIT, X25SYSERR.

If **x25_errno** is set to **X25SYSERR**, **errno** is set to one of the following values:

ENOMEM.

Example

1. Print out the number of PVCs and the default and maximum packet sizes for an X.25 port:

```
struct cb_dev_info_struct *dev_ptr;
dev_ptr = x25_device_query(&link_name);
if (dev_ptr == NULL)
  (void)printf("Error %d from x25_device_query.",x25_errno);
else
{
  if (dev_ptr -> flags & X25FLG_NUA)
  {
    (void)printf("NUA : %s\n",dev_ptr -> nua);
    free(dev_ptr -> nua);
  }
  if (dev_ptr -> flags & X25FLG_NO_OF_VCS)
    (void)printf("Number of PVCs : %d\n",dev_ptr -> no_of_vcs);
  if (dev_ptr -> flags & X25FLG_MAX_RX_PACKET_SIZE)
    (void)printf("Max receive pkt size : %d\n",
      dev_ptr -> max_rx_packet_size);
  if (dev_ptr -> flags & X25FLG_MAX_TX_PACKET_SIZE)
    (void)printf("Max transmit pkt size : %d\n",
      dev-ptr -> max_tx_packet_size);
  if (dev_ptr -> flags & X25FLG_DEFAULT_SVC_RX_PACKET_SIZE)
    (void)printf("Default receive pkt size : %d\n",
      dev_ptr -> default_svc_rx_packet_size);
  if (dev_ptr -> flags & X25FLG_DEFAULT_SVC_TX_PACKET_SIZE)
    (void)printf("Default transmit pkt size : %d\n",
      dev_ptr -> default_svc_tx_packet_size);

  free(dev_ptr);
}
```

Implementation Specifics

This subroutine is part of X.25 Application in AIX BOS Extensions 2.

Related Information

The **x25_circuit_query** and **x25_link_query** subroutines.

x25_init Subroutine

Purpose

Initialize the X.25 application programming interface (API).

Library

The X.25 Communications Library (**libx25s.a**)

C Syntax

int x25_init(
struct cb_link_name_struct *link_name
);

Description

The **x25_init** subroutine sets up X.25 communications with the X.25 port named by **link_name**, by establishing communication with the X.25 device driver. The application must invoke **x25_init** before any other X.25 subroutines. Note that initializing a port does not guarantee that the port is connected (see **x25_link_query** and **x25_link_connect**).

Parameter

link_name A pointer to **cb_link_name_struct**, which gives the name of the X.25 port.

Return Values

If successful, **x25_init** returns 0. If an error occurs, **x25_init** returns –1 and sets **x25_errno** to one of the error codes shown below.

Error Codes

X25BADDEVICE, **X25INIT**, **X25MAXDEVICE**, **X25NOSUCHLINK**, **X25SYSERR**.

Example

Initialize the API for an X.25 port: example program svcxmit in *Communications Programming Concepts*.

Implementation Specifics

This subroutine is part of X.25 Application in AIX BOS Extensions 2.

Related Information

The **x25_term** subroutine.

x25_interrupt Subroutine

Purpose

Sends an interrupt packet.

Library

The X.25 Communications Library (**libx25s.a**)

C Syntax

int x25_interrupt(
int conn_id,
struct cb_int_data_struct *cb_int
);

Description

The **x25_interrupt** subroutine sends an interrupt message. Control is returned to the application when the message has been received by the adapter.

Parameters

conn_id	Connection identifier of the call.
cb_int	Pointer to **cb_int_data_struct**, which contains the interrupt data.

Return Values

If successful, **x25_interrupt** returns 0. If an error occurs, **x25_interrupt** returns –1 and sets **x25_errno** to one of the error codes shown below.

Error Codes

X25BADCONNID, X25NOCARD, X25NOLINK, X25NOTINIT, X25PROTOCOL, X25RESETCLEAR, X25SYSERR, X25TRUNCTX.

If **x25_errno** is set to **X25SYSERR**, **errno** is set to one of the following values:

EINTR, EIO, ENOSPC.

Example

1. Send an interrrupt:

```
struct cb_int_struct int_data;
int_data.flags = X25FLG_INT_DATA;
int_data.data_len = 20;
int_data.int_data = "This is an interrupt";
rc = x25_interrupt(conn_id,&int_data);
if (rc < 0)
  (void)printf("Error %d from x25_interrupt.",x25_errno);
```

Implementation Specifics

This subroutine is part of X.25 Application in AIX BOS Extensions 2.

x25_link_connect Subroutine

Purpose

Connects an X.25 port to the X.25 network.

Library

The X.25 Communications Library (**libx25s.a**)

C Syntax

int x25_link_connect(
struct cb_link_name_struct *link_name
);

Description

The **x25_link_connect** subroutine initializes the X.25 port. Control is returned to the calling application when communications have been established at link level. NET_CONFIG permission is required to use this subroutine. Note that the connection may take 30 seconds to complete.

Parameter

link_name A pointer to **cb_link_name_struct**, which gives the name of the X.25 port.

Return Values

If successful, **x25_link_connect** returns 0. If an error occurs, **x25_link_connect** returns –1 and sets **x25_errno** to one of the error codes shown below.

Error Codes

X25AUTH, **X25LINKUP**, **X25NOCARD**, **X25NOLINK**, **X25NOTINIT**, **X25SYSERR**, **X25TIMEOUT**.

If **x25_errno** is set to **X25SYSERR**, **errno** is set to one of the following values:

EINTR, EIO.

Example

1. Connect the x25s1 port to the network and print a message if an error occurs:

```
struct cb_link_name_struct link_name;
link_name.flags = X25FLG_LINK_NAME;
link_name.link_name = "x25s1";
rc = x25_link_connect(&link_name);
if (rc < 0)
  (void)printf("Error %d occurred while connecting the link.",
  x25_errno);
```

Implementation Specifics

This subroutine is part of X.25 Application in AIX BOS Extensions 2.

Related Information

The **x25_link_disconnect**, **x25_link_monitor**, **x25_link_statistics**, and **x25_link_query** subroutines.

The **xmanage** command.

x25_link_disconnect Subroutine

Purpose

Disconnects an X.25 port.

Library

The X.25 Communications Library (**libx25s.a**)

C Syntax

int x25_link_disconnect(
struct cb_link_name_struct *link_name,
int override
);

Description

The **x25_link_disconnect** subroutine disconnects the X.25 port from the X.25 network. NET_CONFIG permission is required to use this subroutine. Note that the disconnection may take 30 seconds to complete.

Parameters

link_name A pointer to **cb_link_name_struct**, which gives the name of the X.25 port.

override 0 means that the X.25 port is disconnected if all calls have been cleared and all permanent virtual circuits (PVCs) freed. 0 is assumed if the **override** parameter is not used.

A value other than 0 means that the X.25 port is disconnected immediately. Set the **override** parameter to 1 if you want immediate disconnection.

Return Values

If successful, **x25_link_disconnect** returns 0. If an error occurs, **x25_link_disconnect** returns –1 and sets **x25_errno** to one of the error codes shown below.

Error Codes

X25AUTH, X25LINKUSE, X25NOCARD, X25NOLINK, X25NOTINIT, X25SYSERR, X25TIMEOUT.

If **x25_errno** is set to **X25SYSERR**, **errno** is set to one of the following values:

EINTR, EIO.

Examples

1. Disconnect port x25s1 when all calls have been cleared:

```
struct cb_link_name_struct link_name;
link_name.flags = X25FLG_LINK_NAME;
link_name.link_name = "x25s1";
override = 0;
rc = x25_link_disconnect(&link_name,override);
if (rc < 0)
  (void)printf("Error %d from x25_link_disconnect.",x25_errno);
```

2. Disconnect port x25s2 without waiting for calls to be cleared:

```
struct cb_link_name_struct link_name;
link_name.flags = X25FLG_LINK_NAME;
link_name.link_name = "x25s2";
override = 1;
rc = x25_link_disconnect(&link_name,override);
if (rc < 0)
  (void)printf("Error %d from x25_link_disconnect.",x25_errno);
```

Implementation Specifics

This subroutine is part of X.25 Application in AIX BOS Extensions 2.

Related Information

The **x25_link_connect**, **x25_link_monitor**, **x25_link_statistics**, and **x25_link_query** subroutines.

The **xmanage** command.

x25_link_monitor Subroutine

Purpose

Controls monitoring of the activity on an X.25 port.

Library

The X.25 Communications Library (**libx25s.a**)

C Syntax

int x25_link_monitor(
struct cb_link_name_struct *link_name,
long mode,
int ctr_id
);

Description

The **x25_link_monitor** subroutine turns on or off monitoring for an X.25 port. NET_CONFIG and RAS_CONFIG permissions are required to use this subroutine. The application must use the **x25_receive** subroutine to get the monitoring data obtained by **x25_link_monitor**.

Parameters

link_name A pointer to **cb_link_name_struct**, which gives the name of the X.25 port.

mode This consists of a long formed by ORing the values specified using the monitoring flags, `X25_MON_PACKET` and `X25_MON_FRAME`, which enable packet-level and frame-level monitoring respectively. If the mode is set to 0, both frame-level and packet-level monitoring are turned off.

ctr_id Identifier of a counter allocated by a previous **x25_ctr_get**. Although you must pass this parameter, you need *use* it only if you want to wait for notification before receiving the monitoring data.

Return Values

If successful, **x25_link_monitor** returns the connection identifier of the channel on which the monitoring data must be received. If an error occurs, **x25_link_monitor** returns –1 and sets **x25_errno** to one of the error codes shown below.

Error Codes

X25INVMON, X25MONITOR, X25NOCARD, X25NOLINK, X25NOTINIT, X25SYSERR.

If **x25_errno** is set to **X25SYSERR**, **errno** is set to one of the following values:

EINTR, EIO, EPERM.

Examples

1. Start monitoring port x25s1 at both packet-level and frame-level; then wait for and receive one packet of monitoring data:

```
cb_link_name.link_name = "x25s1";
ctr_id = x25ctr_get();
mode = X25_MON_PACKET;          /* For packet-level monitoring */
mode |= X25_MON_FRAME;           /* For frame-level monitoring  */
conn_id = x25_link_monitor(&link_name,mode,ctr_id);
if (conn_id < 0)
  (void)printf("Error %d from x25_link_monitor.",x25_errno);
else
{
  /* Wait for and receive a packet of monitoring data. */
  ctr_array[0].ctr_id = ctr_id;
  ctr_array[0].ctr_value = 0;

  rc = x25_ctr_wait(ctr_array,1);
  rc = x25_receive(&conn_id,&cb_msg);

  /* cb_msg will now contain relevant monitor information. */
}
```

2. Stop monitoring port x25s1:

```
cb_link_name.link_name = "x25s1";
mode = 0;
conn_id = x25_link_monitor(&link_name,mode,ctr_id);
if (conn_id < 0)
  (void)printf("Error %d from x25_link_monitor.",x25_errno);
```

Implementation Specifics

This subroutine is part of X.25 Application in AIX BOS Extensions 2.

Related Information

The **x25_link_connect**, **x25_link_disconnect**, **x25_link_statistics**, and **x25_link_query** subroutines.

The **xmonitor** command.

x25_link_query Subroutine

Purpose

Returns information about the current status of an X.25 port.

Library

The X.25 Communications Library (**libx25s.a**)

C Syntax

int x25_link_query(
struct cb_link_name_struct *link_name
);

Description

The **x25_link_query** subroutine returns the status of the X.25 port as an integer.

Parameter

link_name A pointer to **cb_link_name_struct**, which gives the name of the X.25 port.

Return Values

If successful, **x25_link_query** returns an integer that indicates the status, one of **X25_LINK_CONNECTED, X25_LINK_DISCONNECTED, X25_LINK_CONNECTING**. If an error occurs, **x25_link_query** returns –1 and sets **x25_errno** to one of the error codes shown below.

Error Codes

X25NOCARD, X25NOTINIT, X25SYSERR.

If **x25_errno** is set to **X25SYSERR**, **errno** is set to one of the following values:

EINTR, EIO.

Example

1. Find out whether port x25s1 is connected, disconnected, or connecting:

```
struct cb_link_name_struct link_name;
link_name.flags = X25FLG_LINK_NAME;
link_name.link_name = "x25s1";
rc = x25_link_query(&link_name);
switch (rc)
{
  case X25_LINK_CONNECTED:
    (void)printf("Link is connected\n");
    break;
  case X25_LINK_DISCONNECTED:
    (void)printf("Link is disconnected\n");
    break;
  case X25_LINK_CONNECTING:
    (void)printf("Link is connecting\n");
    break;
    case -1;
      switch (x25_errno);
      {
        case X25SYSERR:
          (void)printf("System error : errno = %d\n",errno);
           perror();
           break;
        case X25NOCARD:
          (void)printf("The X.25 adapter is either not
installed\n");
          (void)printf("or not functioning:");
          (void)printf("Call your system administrator.\n");
           break;
        case X25NOTINIT:
          (void)printf("The application has not initialized\n",
          (void)printf("X.25 communications:");
          (void)printf("Call your system administrator.\n");
           break;
        }
    break;
}
```

Implementation Specifics

This subroutine is part of X.25 Application in AIX BOS Extensions 2.

Related Information

The **x25_circuit_query**, **x25_device_query**, **x25_link_connect**, **x25_link_disconnect**, **x25_link_statistics**, and **x25_link_monitor** subroutines.

The **xmanage** command.

x25_link_statistics Subroutine

Purpose

Request statistics for an X.25 port.

Library

The X.25 Communications Library (**libx25s.a**)

C Syntax

struct cb_link_stats_struct *x25_link_statistics(
struct cb_link_name_struct *link_name,
unsigned short reset
);

Description

The **x25_link stats** subroutine obtains statistics about the X.25 activity on an X.25 port.

Parameters

link_name A pointer to **cb_link_name_struct**, which gives the name of the X.25 port.

reset If **reset** is set to 1, statistics are reset to 0.

Return Values

If successful, **x25_links_statistics** returns a pointer to **cb_link_stats_struct**. The storage for **cb_link_stats_struct** is allocated by the API; it is the responsibility of the application to free it. If an error occurs, **x25_link stats** returns NULL and sets **x25_errno** to one of the error codes shown below.

Error Codes

X25NOCARD, **X25NOTINIT**, **X25SYSERR**.

If **x25_errno** is set to **X25SYSERR**, **errno** is set to one of the following values:

EINTR, **EIO**.

Example

1. Find out the number of virtual circuits currently in use for a port:

```
struct cb_link_stats_struct *link_ptr;
reset = 0;
link_ptr = x25_link_statistics(&link_name,reset);
if (link_ptr == NULL)
  (void)printf("Error %d from x25_link_statistics.",x25_errno);
else
{
  if (link_ptr -> flags & X25FLG_NO_OF_VCS)
    (void)printf("Number of virtual circuits : %d\n",
                 link_ptr -> no_of_vcs);
  if (link_ptr -> flags & X25FLG_LINK_STATS)
    printf ("link statistics returned in x25_query data
structure\n");

  free(link_ptr);
}
```

Implementation Specifics

This subroutine is part of X.25 Application in AIX BOS Extensions 2.

Related Information

The **x25_link_connect**, **x25_link_disconnect**, **x25_link_monitor**, and **x25_link_query** subroutines.

The **xmanage** command.

x25_listen Subroutine

Purpose

Starts listening for incoming calls.

Library

The X.25 Communications Library (**libx25s.a**)

C Syntax

```
int  x25_listen(
NLchar *name,
int ctr_id
    );
```

Description

The **x25_listen** subroutine tells the API that this application is interested in incoming calls that fit the criteria in the routing list entry that has the specified **name.** It also tells the API to associate such calls with the counter identifier specified. It returns a listen identifier to be used by **x25_receive**.

Parameters

name Pointer to a name that is specified in the routing list.

ctr_id Identifier of a counter, allocated by a previous **x25_ctr_get**.

Return Values

If successful, **x25_listen** returns the listen identifier. If an error occurs, **x25_listen** returns –1 and sets **x25_errno** to one of the error codes shown below.

Error Codes

X25AUTHLISTEN, X25INVCTR, X25NAMEUSED, X25NOLINK, X25NONAME, X25NOTINIT, X25SYSERR, X25TABLE, X25TIMEOUT.

Example

Start listening for incoming calls: example program svcrcv in *Communications Programming Concepts.*

Implementation Specifics

This subroutine is part of X.25 Application in AIX BOS Extensions 2.

Related Information

The **x25_deafen** subroutine.

x25_pvc_alloc Subroutine

Purpose

Allocates a permanent virtual circuit (PVC) for use by an application.

Library

The X.25 Communications Library (**libx25s.a**)

C Syntax

```
int  x25_pvc_alloc(
struct cb_pvc_alloc_struct *pvc_ptr,
int ctr_id
    );
```

Description

The **x25_pvc_alloc** subroutine reserves the use of the specified permanent virtual circuit (PVC) for this application only.

Parameters

pvc_ptr A pointer to **cb_pvc_alloc_struct**, which contains the name of the X.25 port and the logical channel number of the PVC to be used. (Together, these identify the PVC.)

ctr_id Identifier of a counter allocated by a previous **x25_ctr_get**.

Return Values

If successful, **x25_pvc_alloc** returns the connection identifier to be used by other subroutines. If an error occurs, **x25_pvc_alloc** returns –1 and sets **x25_errno** to one of the error codes shown below.

Error Codes

X25INVCTR, X25NOCARD, X25NOLINK, X25NOSUCHLINK, X25NOTINIT, X25NOTPVC, X25PVCUSED, X25SYSERR.

If **x25_errno** is set to **X25SYSERR**, **errno** is set to one of the following values:

EINTR, EIO.

Example

Allocate a PVC: example program pvcxmit in *Communications Programming Concepts.*

Implementation Specifics

This subroutine is part of X.25 Application in AIX BOS Extensions 2.

Related Information

The **x25_pvc_free** subroutine.

x25_pvc_free Subroutine

Purpose

Frees a permanent virtual circuit (PVC).

Library

The X.25 Communications Library (**libx25s.a**)

C Syntax

int x25_pvc_free(
int conn_id
);

Description

The **x25_pvc_free** subroutine frees the permanent virtual circuit (PVC) used for the specified connection, so that it can be used by another application. Any data queued for **x25_receive** is lost. It is the responsibility of the application to check the counter identifier for queued data before freeing the PVC.

Parameter

conn_id Connection identifier, returned by the previous **x25_pvc_alloc**.

Return Values

If successful, **x25_pvc_free** returns 0. If an error occurs, **x25_pvc_free** returns –1 and sets **x25_errno** to one of the error codes shown below.

Error Codes

X25BADCONNID, X25NOCARD, X25NOLINK, X25NOTINIT, X25SYSERR.

If **x25_errno** is set to **X25SYSERR**, **errno** is set to one of the following values:

EINTR, EIO.

Example

Free a PVC: example program pvcxmit in *Communications Programming Concepts*.

Implementation Specifics

This subroutine is part of X.25 Application in AIX BOS Extensions 2.

Related Information

The **x25_pvc_alloc** subroutine.

x25_receive Subroutine

Purpose

Receives an incoming packet and indicates the packet type.

Library

The X.25 Communications Library (**libx25s.a**)

C Syntax

int x25_receive(
int *conn_id,
struct cb_msg_struct *cb_msg
);

Description

The **x25_receive** subroutine is used to receive both incoming calls and messages or monitoring data for already-connected calls. One **x25_receive** receives a complete packet sequence. In the event of an interrupt packet being received, an interrupt confirmation is sent automatically by the system.

Parameters

conn_id To receive an incoming call, a pointer to an integer that contains the listen identifier.

To receive a message for any already-connected call, a pointer to an integer that contains 0.

To receive a message for a specific already-connected call, a pointer to an integer that contains the connection identifier of the call.

To receive monitoring data for a call, a pointer to an integer that contains the connection identifier returned by **x25_link_monitor**.

On return from this subroutine, in all cases, a pointer to an integer that *now* contains the actual connection identifier.

cb_msg Pointer to the message structure, **cb_msg_struct**, which includes the **msg_type**. This structure is allocated by the API; it is the responsibility of the application to free this memory.

Return Value

If successful, **x25_receive** returns a non-negative value. If an error occurs, **x25_receive** returns –1 and sets **x25_errno** to one of the error codes shown below.

Error Codes

X25BADID, X25NOACK, X25NOCARD, X25NODATA, X25NOLINK, X25NOTINIT, X25RESETCLEAR, X25SYSERR, X25TRUNCTX.

If **x25_errno** is set to **X25SYSERR**, **errno** is set to one of the following values:

EINTR.

Examples

1. Receive an incoming call: example program svcrcv in *Communications Programming Concepts*.
2. Receive data (or some other message): example program svcrcv in *Communications Programming Concepts*.
3. Receive an acknowledgment that data has been received: example program svcxmit in *Communications Programming Concepts*.

Implementation Specifics

This subroutine is part of X.25 Application in AIX BOS Extensions 2.

Related Information

The **x25_send** subroutine.

x25_reset Subroutine

Purpose

Resynchronizes communications on a virtual circuit.

Library

The X.25 Communications Library (**libx25s.a**)

C Syntax

int x25_reset(
int conn_id,
struct cb_res_struct *cb_res
);

Description

The **x25_reset** subroutine sends out a reset-indication packet to reset the virtual circuit, using the specified connection identifier.

If the application was sending any data at the time of calling this subroutine, the data is flushed from the system, and the **x25_send** subroutine returns an appropriate error code. Incoming data not already passed to the application will be flushed. As resets can cause data to be lost, it is the responsibility of the application to provide higher-level protocol to protect data.

Parameters

conn_id Connection identifier of the call.

cb_res Pointer to **cb_res_struct**, which is used to pass the reset cause and diagnostic codes.

Return Values

If successful, **x25_reset** returns 0. If an error occurs, **x25_reset** returns –1 and sets **x25_errno** to one of the error codes shown below.

Error Codes

X25BADCONNID, X25NOCARD, X25NOLINK, X25NOTINIT, X25PROTOCOL, X25RESETCLEAR, X25SYSERR.

If **x25_errno** is set to **X25SYSERR**, **errno** is set to one of the following values:

EINTR, EIO, ENOSPC.

Example

Reset a call: example program pvcxmit in *Communications Programming Concepts.*

Implementation Specifics

This subroutine is part of X.25 Application in AIX BOS Extensions 2.

Related Information

The **x25_reset_confirm** subroutine.

x25_reset_confirm Subroutine

Purpose

Confirms that a reset-indication has been received.

Library

The X.25 Communications Library (**libx25s.a**)

C Syntax

int x25_reset_confirm(
int conn_id,
);

Description

The **x25_reset_confirm** subroutine sends a reset-confirmation packet. After an reset-indication packet has been received, by **x25_receive**, no further data can be sent or received until the reset-confirmation has been sent. Any data currently in transmission is discarded with an appropriate return code.

Parameter

conn_id Connection identifier of the call.

Return Values

If successful, **x25_reset_confirm** returns 0. If an error occurs, **x25_reset_confirm** returns –1 and sets **x25_errno** to one of the error codes shown below.

Error Codes

EINTR, EIO, ENOSPC, X25BADCONNID, X25NOACK, X25NOCARD, X25NOLINK, X25NOTINIT, X25PROTOCOL, X25RESETCLEAR, X25SYSERR, X25TRUNCTX.

Example

Confirm that a reset indication has arrived: example program pvcrcv in *Communications Programming Concepts.*

Implementation Specifics

This subroutine is part of X.25 Application in AIX BOS Extensions 2.

Related Information

The **x25_reset** subroutine.

x25_send Subroutine

Purpose

Sends a data packet.

Library

The X.25 Communications Library (**libx25s.a**)

C Syntax

int x25_send(
int conn_id,
struct cb_data_struct *cb_data
);

Description

The **x25_send** subroutine transfers the data packet to the adapter for transmission across the network. Control is returned to the calling application as soon as the device driver has indicated successful transferral of the data to the adapter.

Parameters

conn_id	Connection identifier of the call.
cb_data	Pointer to data structure, **cb_data_struct**.

Return Values

If successful, **x25_send** returns 0. If an error occurs, **x25_send** returns –1 and sets **x25_errno** to one of the error codes shown below.

Error Codes

X25BADCONNID, X25NOACK, X25NOCARD, X25NOLINK, X25NOTINIT, X25PROTOCOL, X25RESETCLEAR, X25SYSERR, X25TRUNCTX.

If **x25_errno** is set to **X25SYSERR**, **errno** is set to one of the following values:

EFAULT, EINTR, EIO, ENOSPC.

Examples

1. Send data without the D-bit set: example program svcxmit in *Communications Programming Concepts*.
2. Send data with the D-bit set to request acknowledgment: example program svcxmit in *Communications Programming Concepts*.

Implementation Specifics

This subroutine is part of X.25 Application in AIX BOS Extensions 2.

Related Information

The **x25_receive** and **x25_ack** subroutines.

x25_term Subroutine

Purpose

Terminates the X.25 API for a specified X.25 port.

Library

The X.25 Communications Library (**libx25s.a**)

C Syntax

int x25_term(
struct cb_link_name_struct *link_name
);

Description

The **x25_term** subroutine stops X.25 communications with the X.25 port named by **link_name**, by terminating communication with the X.25 device driver. If this is the last X.25 port open for this process, X.25 resources are freed.

x25_term clears any virtual circuits that are still being used by the application. Nevertheless, you should clear the virtual circuits and tidy up in a controlled way before invoking **x25_term**.

Parameter

link_name A pointer to **cb_link_name_struct**, which gives the name of the X.25 port.

Return Values

If successful, **x25_term** returns 0. If an error occurs, **x25_term** returns –1 and sets **x25_errno** to one of the error codes shown below.

Error Codes

X25BADDEVICE, X25SYSERR.

Example

Terminate the API: example program svcxmit in *Communications Programming Concepts*.

Implementation Specifics

This subroutine is part of X.25 Application in AIX BOS Extensions 2.

Related Information

The **x25_init** subroutine.

Devices Services

SYS_CFGDD sysconfig Operation

Purpose

Calls a previously loaded device driver at its module entry point.

Description

The SYS_CFGDD **sysconfig** operation calls a previously loaded device driver at its module entry point. The device driver's module entry point, by convention, is its **ddconfig** entry point. The SYS_CFGDD operation is typically invoked by device configure or unconfigure methods to initialize or terminate a device driver, or to request device vital product data.

The **sysconfig** subroutine puts no restrictions on the command code passed to the device driver. This allows the device driver's **ddconfig** entry point to provide additional services, if desired.

The *parmp* parameter on the SYS_CFGDD **sysconfig** operation points to a **cfg_dd** structure defined in the **sys/sysconfig.h** header file. The *parmlen* parameter on the **sysconfig** system call should be set to the size of this structure.

If the **kmid** variable in the **cfg_dd** structure is 0, the desired device driver is assumed to be already installed in the device switch table. The major portion of the device number (passed in the **devno** field in the **cfg_dd** structure) is used as an index into the device switch table. The device switch table entry indexed by this **devno** field contains the device driver's **ddconfig** entry point to be called.

If the **kmid** variable is not 0, it contains the module ID to use in calling the device driver. A **uio** structure is used to pass the address and length of the device–dependent structure, specified by the **cfg_dd.ddsptr** and **cfg_dd.ddslen** fields, to the device driver being called.

The **ddconfig** device driver entry point provides information on how to define the **ddconfig** routine.

The device driver to be called is responsible for using the appropriate routines to copy the device–dependent structure (DDS) from user to kernel space.

Return Values

If the SYS_CFGDD **sysconfig** operation successfully calls the specified device driver, the return code from the **ddconfig** routine determines the value returned by this subroutine. If the **ddconfig** routine's return code is 0, then the value returned by the **sysconfig** subroutine is 0. Otherwise the value returned is a –1, and the **errno** global variable is set to the return code provided by the device driver's **ddconfig** routine.

Errors detected by the SYS_CFGDD **sysconfig** operation result in the following values for the **errno** variable:

EACESS The calling process does not have the required privilege.

EFAULT The calling process does not have sufficient authority to access the data area described by the *parmp* and *parmlen* parameters provided on the system call. This error is also returned if an I/O error occurred when accessing data in this area.

EINVAL Invalid module ID.

ENODEV Module ID specified by the **cfg_dd.kmid** field was 0, and an invalid or undefined **devno** value was specified.

Related Information

The **sysconfig** subroutine.

The **ddconfig** device driver entry point.

The Device Switch Table.

The **uio** structure.

The Device–Dependent (DDS) structure.

Understanding Major and Minor Numbers For A Special File in *Kernel Extensions and Device Support Programming Concepts.*

System Call Kernel Extension Overview in *Kernel Extensions and Device Support Programming Concepts.*

Device Driver Kernel Extension Overview in *Kernel Extensions and Device Support Programming Concepts.*

Virtual File System Introduction in *Kernel Extensions and Device Support Programming Concepts.*

Device Configuration Subsystem: Programming Introduction in *Kernel Extensions and Device Support Programming Concepts.*

Programming in the Kernel Environment in *Kernel Extensions and Device Support Programming Concepts.*

Understanding Kernel Extension Binding in *Kernel Extensions and Device Support Programming Concepts.*

SYS_CFGKMD sysconfig Operation

Purpose

Invokes a previously loaded kernel object file at its module entry point.

Description

The SYS_CFGKMD **sysconfig** operation invokes a previously loaded kernel object file at its module entry point, typically for initialization or termination functions. The SYS_CFGDD operation performs a similar function for device drivers.

The *parmp* parameter on the **sysconfig** subroutine points to a **cfg_kmod** structure, which is defined in the **sys/sysconfig.h** header file. The **kmid** field in this structure specifies the kernel module ID of the module to invoke. This value is returned when using the SYS_KLOAD or SYS_SINGLELOAD **sysconfig** operation to load the object file.

The **cmd** field in the **cfg_kmod** structure is a module–dependent parameter specifying the action that the routine at the module's entry point should perform. This is typically used for initialization and termination commands after loading and prior to unloading the object file.

The **mdiptr** field in the **cfg_kmod** structure points to a module–dependent structure whose size is specified by the **mdilen** field. This field is used to provide module–dependent information to the module to be called. If no such information is needed, the **mdiptr** field can be NULL.

If the **mdiptr** field is not NULL, then the SYS_CFGKMD operation builds a **uio** structure describing the address and length of the module–dependent information in the caller's address space. The **mdiptr** and **mdilen** fields are used to fill in the fields of this **uio** structure. The module is then called at its module entry point with the *cmd* parameter and a pointer to the **uio** structure. If there is no module–dependent information to be provided, the *uiop* parameter passed to the module's entry point is set to NULL.

The module's entry point should be defined as follows:

int module_entry(*cmd*, *uiop***)**
int *cmd*;
struct uio **uiop*;

The definition of the module–dependent information and its length is specific to the module being configured. The module to be called is responsible for using the appropriate routines to copy the module–dependent information from user to kernel space.

Return Values

If the kernel module to be invoked is successfully called, its return code determines the value that is returned by the SYS_CFGKMOD **sysconfig** operation. If the called module's return code is 0, then the value returned by the **sysconfig** subroutine is 0. Otherwise the value returned is –1 and the **errno** global variable is set to the called module's return code.

Errors detected by the SYS_CFGKMOD **sysconfig** operation result in the following values for the **errno** variable:

EINVAL	Invalid module ID.

EACESS The calling process does not have the required privilege.

EFAULT The calling process does not have sufficient authority to access the data area described by the *parmp* and *parmlen* parameters provided on the system call. This error is also returned if an I/O error occurred when accessing data in this area.

Related Information

The **sysconfig** subroutine.

The SYS_CFGDD **sysconfig** subroutine, SYS_KLOAD **sysconfig** subroutine, SYS_SINGLELOAD **sysconfig** operation.

The **uio** structure.

System Call Kernel Extension Overview in *Kernel Extensions and Device Support Programming Concepts.*

Device Driver Introduction in *Kernel Extensions and Device Support Programming Concepts.*

Device Driver Kernel Extension Overview in *Kernel Extensions and Device Support Programming Concepts.*

Virtual File System Introduction in *Kernel Extensions and Device Support Programming Concepts.*

Device Configuration Subsystem: Programming Introduction in *Kernel Extensions and Device Support Programming Concepts.*

Programming in the Kernel Environment in *Kernel Extensions and Device Support Programming Concepts.*

Understanding Kernel Extension Binding in *Kernel Extensions and Device Support Programming Concepts.*

sysconfig Subroutine

Purpose

Provides a service for controlling system/kernel configuration.

Syntax

#include <sys/types.h>
#include <sys/sysconfig.h>

int sysconfig (*cmd*, *parmp*, *parmlen*)
int *cmd*;
void **parmp*;
int *parmlen*;

Parameters

cmd Specifies the function that the **sysconfig** subroutine is to perform.

parmp Specifies a user-provided structure.

parmlen Specifies the length of the user–provided structure indicated by the *parmp* parameter.

Description

The **sysconfig** subroutine is used to customize the AIX Operating System. This subroutine provides a means of loading, unloading, and configuring kernel extensions. These kernel extensions can be additional kernel services, additional system calls, device drivers, or file systems. The **sysconfig** subroutine also provides the ability to read and set system runtime operating parameters.

Use of the **sysconfig** subroutine requires appropriate privilege.

The particular operation that the **sysconfig** subroutine provides is defined by the value of the *cmd* parameter. The following operations are defined:

SYS_KLOAD Loads a kernel extension object file into kernel memory.

SYS_SINGLELOAD
Loads a kernel extension object file only if it is not already loaded.

SYS_QUERYLOAD
Determines if a specified kernel object file is loaded.

SYS_KULOAD
Unloads a previously loaded kernel object file.

SYS_CFGKMOD
Calls the specified module at its module entry point for configuration purposes.

SYS_CFGDD Calls the specified device driver configuration routine (module entry point).

SYS_QDVSW
Checks the status of a device switch entry in the device switch table.

SYS_GETPARMS
Returns a structure containing the current values of runtime system parameters found in the **var** structure.

SYS_SETPARMS
Sets runtime system parameters from a caller–provided structure.

Loader Symbol Binding Support, described with the **sysconfig** SYS_KLOAD operation, explains the symbol binding support provided when loading kernel object files.

Return Values

These **sysconfig** operations return a value of 0 upon successful completion of the subroutine. Otherwise, a value of –1 is returned and the **errno** global variable is set to indicate the error.

Any **sysconfig** operation requiring a structure from the caller fails if the structure is not entirely within memory addressable by the calling process. A return value of –1 is passed back and the **errno** global variable is set to EFAULT.

Related Information

The **ddconfig** device driver entry point.

Understanding the device switch table.

Loader Symbol Binding Support in the SYS_KLOAD **sysconfig** operation.

System Call Kernel Extension Overview in *Kernel Extensions and Device Support Programming Concepts.*

Device Driver Kernel Extension Overview in *Kernel Extensions and Device Support Programming Concepts.*

Virtual File System Introduction in *Kernel Extensions and Device Support Programming Concepts.*

Device Configuration Subsystem: Programming Introduction in *Kernel Extensions and Device Support Programming Concepts.*

Programming in the Kernel Environment in *Kernel Extensions and Device Support Programming Concepts.*

Understanding Kernel Extension Binding in *Kernel Extensions and Device Support Programming Concepts.*

SYS_GETPARMS sysconfig Operation

Purpose

Copies the system parameter structure into a user–specified buffer.

Description

The SYS_GETPARMS **sysconfig** operation copies the system parameter **var** structure into a user–allocated buffer. This structure may be used for informational purposes alone or prior to setting specific system parameters.

In order to set system parameters, the required fields in the **var** structure must be modified, and then the SYS_SETPARMS **sysconfig** operation can be called to change the system runtime operating parameters to the desired state.

The *parmp* parameter on the **sysconfig** subroutine points to a buffer that is to contain all or part of the **var** structure defined in the **sys/var.h** header file. The fields in the **var_hdr** part of the **var** structure are used for parameter update control.

The *parmlen* parameter on the system call should be set to the length of the **var** structure or to the number of bytes of the structure that is desired. The complete definition of the system parameters structure can be found in the **sys/var.h** header file.

Return Values

The SYS_GETPARMS **sysconfig** operation returns a value of –1 if an error occurs and the **errno** global variable is set to the following:

EACCES The calling process does not have the required privilege.

EFAULT The calling process does not have sufficient authority to access the data area described by the *parmp* and *parmlen* parameters provided on the subroutine. This error is also returned if an I/O error occurred when accessing data in this area.

Related Information

The **sysconfig** subroutine.

The SYS_SETPARMS **sysconfig** operation.

Programming in the Kernel Environment in *Kernel Extensions and Device Support Programming Concepts.*

SYS_KLOAD sysconfig Operation

Purpose

Loads a kernel extension into the kernel.

Description

The SYS_KLOAD **sysconfig** function is used to load a kernel extension object file specified by a pathname into the kernel. A kernel module ID for that instance of the module is returned. The SYS_KLOAD **sysconfig** operation loads a new copy of the object file into the kernel even though one or more copies of the specified object file may have already been loaded into the kernel. The returned module ID can then be used for any of these three functions:

- Subsequent invocation of the module's entry point (using the **sysconfig** SYS_CFGKMOD operation)
- Invocation of a device driver's **ddconfig** routine (using the **sysconfig** SYS_CFGDD operation)
- Unloading the kernel module (using the **sysconfig** SYS_KULOAD operation).

The *parmp* parameter on the **sysconfig** subroutine must point to a **cfg_load** structure, (defined in the **sys/sysconfig.h** header file), with the **path** field specifying the path name for a valid kernel object file. The *parmlen* parameter should be set to the size of the **cfg_load** structure.

Note: A separate **sysconfig** operation exists, the SYS_SINGLELOAD operation, which also loads kernel extensions. This operation, however, only loads the requested object file if it has not already been loaded.

Loader Symbol Binding Support

The following information describes the symbol binding support provided when loading kernel object files.

Importing Symbols

Symbols imported from the kernel name space are resolved with symbols that exist in the kernel name space at the time of the load. (Symbols are imported from the kernel name space by specifying the **#!/unix** character string as the first field in an import list at link–edit time.)

Kernel modules can also import symbols from other kernel object files. These other kernel object files are loaded along with the specified object file if they are required to resolve the imported symbols.

Loader Symbol Binding Support, described with the **sysconfig** SYS_KLOAD operation, explains the symbol binding support provided when loading kernel object files.

Finding Directory Locations For Unqualified File Names

If the module header contains an unqualified base filename for the symbol (no / (slash) characters in the name), a libpath search string is used to find the location of the shared object file required to resolve imported symbols. This libpath search string can be taken from one of two places. If the **libpath** field in the **cfg_load** structure is not NULL, then it points to a character string specifying the libpath to be used. However, if the **libpath** field is NULL, then the libpath is taken from the module header of the object file specified by the **path** field in the same (**cfg_load**) structure.

The libpath specification found in object files loaded in order to resolve imported symbols is not used.

The kernel loader service does not support deferred symbol resolution. The load of the kernel object file is terminated with an error if any imported symbols cannot be resolved.

Exporting Symbols

Any symbols exported by the specified kernel object file are added to the kernel name space. This makes these symbols available to other subsequently loaded kernel object files. Any symbols specified with the **SYSCALL** keyword in the export list at linkedit time are added to the system call table at load time. These symbols are then available to application programs as a system call.

Kernel object files loaded on behalf of the specified kernel object file, in order to resolve imported symbols, do not have their exported symbols added to the kernel name space.

These object files are considered private since they do not export symbols to the global kernel name space. For these types of object files, a new copy of the object file is loaded on each SYS_KLOAD operation of a kernel extension that imports symbols from the private object file. In order for a kernel extension to add its exported symbols to the kernel name space, it must be explicitly loaded with the SYS_KLOAD **sysconfig** operation before any other object files using the symbols are loaded. For kernel extensions of this type (those exporting symbols to the kernel name space), typically only one copy of the object file should ever be loaded.

Return Values

If the object file is loaded without error, the module ID is returned in the **kmid** variable within the **cfg_load** structure and the subroutine returns a 0.

On error, the subroutine returns a –1 and the **errno** global variable is set to one of the following values:

EACESS One of the following reasons applies:

- The calling process does not have the required privilege.
- An object module to be loaded is not an ordinary file.
- The mode of the object module file denies read–only permission.

EFAULT The calling process does not have sufficient authority to access the data area described by the *parmp* and *parmlen* parameters provided on the system call. This error is also returned if an I/O error occurred when accessing data in this area.

ENOEXEC The program file has the appropriate access permission, but has an invalid XCOFF object file indication in its header. The **sysconfig** SYS_KLOAD operation only supports loading of XCOFF object files. This error is also returned if the loader is unable to resolve an imported symbol.

EINVAL The program file has a valid XCOFF indicator in its header, but the header is damaged or is incorrect for the machine on which the file is to be run.

ENOMEM The load requires more kernel memory than is allowed by the system–imposed maximum.

ETXTBSY The object file is currently open for writing by some process.

Related Information

The **sysconfig** subroutine.

The **SYS_SINGLELOAD sysconfig** operation, **SYS_KULOAD sysconfig** operation, **SYS_CFGDD sysconfig** operation, **SYS_CFGKMOD sysconfig** operation.

The **ddconfig** device driver entry point.

System Call Kernel Extension Overview in *Kernel Extensions and Device Support Programming Concepts.*

Device Driver Kernel Extension Overview in *Kernel Extensions and Device Support Programming Concepts.*

Virtual File System Introduction in *Kernel Extensions and Device Support Programming Concepts.*

Device Configuration Subsystem: Programming Introduction in *Kernel Extensions and Device Support Programming Concepts.*

Programming in the Kernel Environment in *Kernel Extensions and Device Support Programming Concepts.*

Understanding Kernel Extension Binding in *Kernel Extensions and Device Support Programming Concepts.*

SYS_KULOAD sysconfig Operation

Purpose

Unloads a loaded kernel object file and any imported kernel object files that were loaded with it.

Description

The SYS_KULOAD **sysconfig** operation unloads a previously loaded kernel file and any imported kernel object files that were automatically loaded with it. It does this by decrementing the load and use counts of the specified object file and any object file having symbols imported by the specified object file.

The *parmp* parameter on the **sysconfig** subroutine should point to a **cfg_load** structure, as described for the SYS_KLOAD operation. The **kmid** field should specify the kernel module ID that was returned when the object file was loaded by the **sysconfig** SYS_KLOAD or SYS_SINGLELOAD operation. The **path** and **libpath** fields are not used for this command and can be set to NULL. The *parmlen* parameter should be set to the size of the **cfg_load** structure.

Upon successful completion, the specified object file (and any other object files containing symbols that the specified object file imports) will have their load and use counts decremented. If there are no users of any of the module's exports and its load count is 0, then the object file is immediately unloaded.

However, if there are users of this module, (that is, there are modules bound to this module's exported symbols), the specified module is not unloaded. Instead, it is unloaded on some subsequent unload request, when its use and load counts have gone to zero. The specified module is not in fact unloaded until all current users have been unloaded.

Note: Care must be taken to ensure that a routine has freed all of its system resources before being unloaded. For example, a device driver is typically prepared for unloading by using the **sysconfig** subroutine's SYS_CFGDD operation and specifying termination.

Loader Symbol Binding Support, described with the **sysconfig** SYS_KLOAD operation, explains the symbol binding support provided when loading kernel object files.

Return Values

If the unload operation is successful or the specified object file's load count is successfully decremented, a value of 0 is returned.

On error, the specified file and any imported files are not unloaded, nor are their load and use counts decremented. A value of –1 is returned and the **errno** global variable is set to one of the following:

EACESS The calling process does not have the required privilege.

EINVAL Invalid module ID or the specified module is no longer loaded or already has a load count of 0.

EFAULT The calling process does not have sufficient authority to access the data area described by the *parmp* and *parmlen* parameters provided to the subroutine. This error is also returned if an I/O error occurred when accessing data in this area.

Related Information

The **sysconfig** subroutine.

The SYS_KLOAD **sysconfig** operation, SYS_SINGLELOAD **sysconfig** operation, SYS_CFGDD **sysconfig** operation.

SYS_QDVSW sysconfig Operation

Purpose

Checks the status of a device switch entry in the device switch table.

Description

The SYS_QDVSW **sysconfig** operation checks the status of a device switch entry in the device switch table.

The *parmp* parameter on the **sysconfig** subroutine points to a **qry_devsw** structure defined in the **sys/sysconfig.h** header file. The *parmlen* parameter on the subroutine should be set to the length of the **qry_devsw** structure.

The **qry_devsw** field in the **qry_devsw** structure is modified to reflect the status of the device switch entry specified by the **qry_devsw** field. (The value in the **devno** field corresponds to the major portion of the device number.) The following flags can be returned in the **status** field:

DSW_UNDEFINED
The device switch entry is not defined if this flag has a value of 0 on return.

DSW_DEFINED
The device switch entry is defined.

DSW_CREAD The device driver in this device switch entry provides a routine for character reads or raw input. This flag is set when the device driver provides a **ddread** entry point.

DSW_CWRITE
The device driver in this device switch entry provides a routine for character writes or raw output. This flag is set when the device driver provides a **ddwrite** entry point.

DSW_BLOCK The device switch entry is defined by a block device driver. This flag is set when the device driver provides a **ddstrategy** entry point.

DSW_MPX The device switch entry is defined by a multiplexed device driver. This flag is set when the device driver provides a **ddmpx** entry point.

DSW_SELECT
The device driver in this device switch entry provides a routine for handling the **select** or **poll** subroutines. This flag is set when the device driver provides a **ddselect** entry point.

DSW_DUMP The device driver defined by this device switch entry provides the capability to support one or more of its devices as targets for a kernel dump. This flag is set when the device driver has provided a **dddump** entry point.

DSW_CONSOLE
The device switch entry is defined by the console device driver.

DSW_TCPATH
The device driver in this device switch entry supports devices that are considered to be in the Trusted Computing Path and provides support for the **revoke** subroutine and **frevoke** subroutine. This flag is set when the device driver provides a **ddrevoke** entry point.

DSW_OPENED
The device switch entry is defined and the device has outstanding opens. This flag is set when the device driver has at least one outstanding open.

The DSW_UNDEFINED condition is indicated when the device switch entry has not been defined or has been defined and subsequently deleted. Multiple status flags may be set for other conditions of the device switch entry.

Return Values

If no error is detected, this operation returns with a value of 0. If an error is detected, the return value is set to a value of –1. The **errno** global variable is also set to one of these three values:

EACESS The calling process does not have the required privilege.

EINVAL Device number exceeds the maximum allowed by the kernel.

EFAULT The calling process does not have sufficient authority to access the data area described by the *parmp* and *parmlen* parameters provided on the system call. This error is also returned if an I/O error occurred when accessing data in this area.

Related Information

The **sysconfig** subroutine.

The **ddread** device driver entry point, **ddwrite** device driver entry point, **ddstrategy** device driver entry point, **ddmpx** device driver entry point, **ddselect** device driver entry point, **dddump** device driver entry point, **ddrevoke** device driver entry point.

console special file.

Understanding the Device Switch Table in *Kernel Extensions and Device Support Programming Concepts.*

Trusted Computing Path Support In a Character Device Driver in *Kernel Extensions and Device Support Programming Concepts.*

Understanding Block I/O Device Drivers in *Kernel Extensions and Device Support Programming Concepts.*

Providing Raw I/O Support In a Block I/O Device Driver in *Kernel Extensions and Device Support Programming Concepts.*

Understanding Character I/O Device Drivers, Multiplexed Support In a Character Device Driver in *Kernel Extensions and Device Support Programming Concepts.*

System Call Kernel Extension Overview in *Kernel Extensions and Device Support Programming Concepts.*

Device Driver Kernel Extension Overview in *Kernel Extensions and Device Support Programming Concepts.*

Virtual File System Introduction in *Kernel Extensions and Device Support Programming Concepts*.

Device Configuration Subsystem: Programming Introduction in *Kernel Extensions and Device Support Programming Concepts*.

Programming in the Kernel Environment in *Kernel Extensions and Device Support Programming Concepts*.

Understanding Kernel Extension Binding in *Kernel Extensions and Device Support Programming Concepts*.

SYS_QUERYLOAD sysconfig Operation

Purpose

Determines if a kernel object file has already been loaded.

Description

The SYS_QUERYLOAD **sysconfig** operation performs a query operation to determine if a given object file has been loaded. This object file is specified by the **path** field in the **cfg_load** structure passed in with the *parmp* parameter. This operation utilizes the same **cfg_load** structure that is specified for the SYS_KLOAD operation.

If the specified object file is not loaded, the **kmid** field in the **cfg_load** structure is set to a value of 0 on return. Otherwise, the kernel module ID of the module is returned in the **kmid** field. If multiple instances of the module have been loaded into the kernel, the module ID of the one most recently loaded is returned.

The **libpath** field in the **cfg_load** structure is not used for this option.

Note: Note that a path name comparison is done to determine if the specified object file has been loaded. This operation will erroneously return a *not loaded* condition if the path name to the object file is expressed differently than it was on a previous load request.

Loader Symbol Binding Support, described with the **sysconfig** SYS_KLOAD operation, explains the symbol binding support provided when loading kernel object files.

Return Values

If the specified object file is found, the module ID is returned in the *kmid* variable within the **cfg_load** structure and the subroutine returns a 0. If the specified file is not found, a kmid variable of 0 is returned with a return code of 0. On error, the subroutine returns a –1 and the **errno** global variable is set to one of the following values:

EACCES The calling process does not have the required privilege.

EFAULT The calling process does not have sufficient authority to access the data area described by the *parmp* and *parmlen* parameters provided on the subroutine. This error is also returned if an I/O error occurred when accessing data in this area.

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

EIO An I/O error occurred during the operation.

Related Information

The **sysconfig** subroutine.

The SYS_SINGLELOAD **sysconfig** operation, SYS_KLOAD **sysconfig** operation.

Loader Symbol Binding Support in the SYS_KLOAD **sysconfig** operation.

Programming in the Kernel Environment in *Kernel Extensions and Device Support Programming Concepts.*

Understanding Kernel Extension Binding in *Kernel Extensions and Device Support Programming Concepts.*

SYS_SETPARMS sysconfig Operation

Purpose

Sets the kernel runtime tunable parameters.

Description

The SYS_SETPARMS **sysconfig** operation sets the current system parameters from a copy of the system parameter **var** structure provided by the caller. Only the runtime tunable parameters in the **var** structure can be set by this subroutine.

If the **var_vers** and **var_gen** values in the caller-provided structure do not match the **var_vers** and **var_gen** values in the current system **var** structure, no parameters are modified and an error is returned. The **var_vers**, **var_gen** and **var_size** fields in the structure should not be altered. The **var_vers** value is assigned by the kernel and is used to insure that the correct version of the structure is being used. The **var_gen** value is a generation number having a new value for each read of the structure. This provides consistency between the data read by the SYS_GETPARMS operation and the data written by the SYS_SETPARMS operation.

The *parmp* parameter on the **sysconfig** subroutine points to a buffer that contains all or part of the **var** structure as defined in the **<sys/var.h>** header file.

The *parmlen* parameter on the subroutine should be set either to the length of the **var** structure or to the size of the structure containing the parameters to be modified. The number of system parameters modified by this operation is determined either by the *parmlen* parameter value or by the **var_size** field in the caller-provided **var** structure. (The smaller of the two values is used.)

The structure provided by the caller must contain at least the header fields of the **var** structure. Otherwise, an error will be returned. Partial modification of a parameter in the **var** structure can occur if the caller's data area does not contain enough data to end on a field boundary. It is up to the caller to ensure that this does not happen.

Return Values

The SYS_SETPARMS **sysconfig** operation returns a value of –1 if an error occurred, and the **errno** global variable is set to one of the following:

EACESS The calling process does not have the required privilege.

EINVAL One of the following error situations exists:

- The **var_vers** version number of the provided structure does not match the version number of the current **var** structure.
- The structure provided by the caller does not contain enough data to specify the header fields within the **var** structure.
- One of the specified variable values is invalid or not allowed. On the return from the subroutine, the **var_vers** field in the caller-provided buffer contains the byte offset of the first variable in the structure that was detected in error.

EAGAIN The **var_gen** generation number in the structure provided does not match the current generation number in the kernel. This occurs if consistency is lost between reads and writes of this structure. The caller should repeat the read, modify, and write operations on the structure.

EFAULT The calling process does not have sufficient authority to access the data area described by the *parmp* and *parmlen* parameters provided to the subroutine. This error is also returned if an I/O error occurred when accessing data in this area.

Related Information

The **sysconfig** subroutine.

The SYS_GETPARMS **sysconfig** operation.

Understanding Kernel Extension Binding in *Kernel Extensions and Device Support Programming Concepts.*

SYS_SINGLELOAD sysconfig Operation

Purpose

Loads a kernel extension module if it is not already loaded.

Description

The SYS_SINGLELOAD **sysconfig** operation is identical to the SYS_KLOAD operation, except that the SYS_SINGLELOAD operation loads the object file only if an object file with the same path name has not already been loaded into the kernel.

If an object file with the same path name has already been loaded, the module ID for that object file is returned in the **kmid** field and its load count incremented. If the object file is not loaded, this operation performs the load request exactly as defined for the SYS_KLOAD function.

This option is useful in supporting global kernel routines where only one copy of the routine and its data can be present. Typically routines that export symbols to be added to the kernel name space are of this type.

Note: Note that a path name comparison is done to determine if the same object file has already been loaded. This function will erroneously load a new copy of the object file into the kernel if the path name to the object file is expressed differently than it was on a previous load request.

Loader Symbol Binding Support, described with the **sysconfig** SYS_KLOAD operation, explains the symbol binding support provided when loading kernel object files.

Return Values

The SYS_SINGLELOAD operation returns the same set of error codes that the SYS_KLOAD operation returns.

Related Information

The **sysconfig** subroutine.

The SYS_KLOAD **sysconfig** operation.

Programming in the Kernel Environment in *Kernel Extensions and Device Support Programming Concepts.*

Understanding Kernel Extension Binding in *Kernel Extensions and Device Support Programming Concepts.*

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

D

E

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