AIX Version 7.2

Technical Reference: Communications, Volume 1

AIX Version 7.2

Technical Reference: Communications, Volume 1

Note
Before using this information and the product it supports, read the information in "Notices" on page 409.

This edition applies to AIX Version 7.2 and to all subsequent releases and modifications until otherwise indicated in new editions.

© Copyright IBM Corporation 2015, 2016.
US Government Users Restricted Rights – Use, duplication or disclosure restricted by GSA ADP Schedule Contract with IBM Corp.

Contents

© Copyright IBM Corp. 2015, 2016

About this document

This topic collection provides experienced C programmers with complete detailed information about data link controls, the Data Link Provider Interface, eXternal Data Representation, the AIX® 3270 Host Connection Program, the Network Computing System, Network Information Services and Network Information Services+, the New Database Manager, and remote procedure calls for the AIX operating system. To use the topic collection effectively, you should be familiar with commands, system calls, subroutines, file formats, and special files. This publication is also available on the documentation CD that is shipped with the operating system.

Highlighting

The following highlighting conventions are used in this document:

Item	Description
Bold	Identifies commands, subroutines, keywords, files, structures, directories, and other items whose names are predefined by the system. Also identifies graphical objects such as buttons, labels, and icons that the user selects.
Italics	Identifies parameters whose actual names or values are to be supplied by the user.
`Monospace`	Identifies examples of specific data values, examples of text similar to what you might see displayed, examples of portions of program code similar to what you might write as a programmer, messages from the system, or information you should actually type.

Case-sensitivity in AIX

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

ISO 9000

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

© Copyright IBM Corp. 2015, 2016

Technical Reference: Communications, Volume 1

The subroutines, their structure, parameters, and error codes that are used in AIX are discussed in this topic collection.

The AIX operating system is designed to support The Open Group's Single UNIX Specification Version 3 (UNIX 03) for portability of operating systems based on the UNIX operating system. Many new interfaces, and some current ones, have been added or enhanced to meet this specification. To determine the correct way to develop a UNIX 03 portable application, see The Open Group's UNIX 03 specification on The UNIX System website (http://www.unix.org).

Data Link Controls

This topic collection includes the subroutines that is used to perform various functions for the GDLC device manager.

close Subroutine Interface for Data Link Control (DLC) Devices

Purpose

Closes the generic data link control (GDLC) device manager using a file descriptor.

Syntax

```
int close ( fildes)
```

Description

The **close** subroutine disables a 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.

Each GDLC supports the **close** subroutine interface by way of its **dlcclose** and **dlcmpx** entry points. This subroutine can be called from the process environment only.

Parameters

Item	Description
fildes	Specifies the file descriptor of the GDLC being closed.

Return Values

Item	Description
0	Indicates a successful operation.
EBADF	Indicates a bad file number. This value is defined in the **/usr/include/sys/errno.h** file.

If an error occurs, a value of -1 is also returned.

Related reference:

“open Subroutine Interface for Data Link Control (DLC) Devices” on page 18

Related information:

close subroutine

Generic Data Link Control (GDLC) Environment Overview

© Copyright IBM Corp. 2015, 2016

dlcclose Entry Point of the GDLC Device Manager

Purpose

Closes a generic data link control (GDLC) channel.

Syntax

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

int dlcclose ( devno,  chan)
```

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

Description

Each GDLC supports the **dlcclose** entry point as its switch table entry for the **close** subroutine. The file system calls this entry point from the process environment only.The **dlcclose** entry point 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 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

Item	Description
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.

Return Values

Item	Description
0	Indicates a successful operation.
EBADF	Indicates a bad file number. This value is defined in the **/usr/include/sys/errno.h** file.

Related reference:

"dlcopen Entry Point of the GDLC Device Manager" on page 5

Related information:

fp_close kernel service

Generic Data Link Control (GDLC) Environment Overview

dlcconfig Entry Point of the GDLC Device Manager

Purpose

Configures the generic data link control (GDLC) device manager.

Syntax

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

int dlcconfig ( devno,  op,  uiop)
```

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

Description

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

Each GDLC supports the **dlcconfig** entry point as its switch table entry for the **sysconfig** subroutine. The file system calls this entry point from the process environment only.

Parameters

Item	Description
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. One **dev_t** device number exists for each type of GDLC, such as Ethernet, Token-Ring, or SDLC.
op	Specifies the operation code that indicates the function to be performed: **CFG_INIT** Initializes the GDLC device manager. **CFG_TERM** Terminates the GDLC device manager. **CFG_QVPD** Queries GDLC vital product data. This operation code is optional.
uiop	Points to the **uio** structure specifying the location and length of the caller's data area for the **CFG_INIT** and **CFG_QVPD** operation codes. No data areas are specifically defined for GDLC, but DLCs can define the data areas for a particular network.

Return Values

The following return values are defined in the **/usr/include/sys/errno.h** file:

Item	Description
0	Indicates a successful operation.
EINVAL	Indicates an invalid value.
ENODEV	Indicates that no such device handler is present.
EFAULT	Indicates that a kernel service, such as the **uiomove** or **devswadd** kernel service, has failed.

Related information:

ddconfig subroutine

uiomove subroutine

Generic Data Link Control (GDLC) Environment Overview

dlcioctl Entry Point of the GDLC Device Manager

Purpose

Issues specific commands to generic data link control (GDLC).

Syntax

```
#include <sys/device.h>
#include <sys/gdlextcb.h>
int dlcioctl (devno, op, arg, devflag, chan, ext)
```

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

Description

The **dlcioctl** entry point is called when an 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 GDLC.

Each GDLC supports the **dlcioctl** entry point as its switch table entry for the **ioctl** subroutine. The file system calls this entry point from the process environment only.

Parameters

Item	Description
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. One **dev_t** device number exists 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.
arg	Indicates the parameter from the subroutine that specifies the address of a parameter block.
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

The following return values are defined in the **/usr/include/sys/errno.h** file.

Value	Description
0	Indicates a successful operation.
EBADF	Indicates a bad file number.
EINVAL	Indicates an invalid value.
ENOMEM	Indicates insufficient resources to satisfy the **ioctl** subroutine.

Related reference:

"ioctl Operations (op) for DLC" on page 14

Related information:

ioctl subroutine

fp_ioctl subroutine

dlcmpx Entry Point of the GDLC Device Manager

Purpose

Decodes the device handler's special file name appended to the open call.

Syntax

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

int dlcmpx ( devno, chanp,  channame)
```

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

Description

The operating system calls the **dlcmpx** entry point when a generic data link control (GDLC) channel is allocated. This routine decodes the name of the device handler appended to the end of the GDLC 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 to identify the channel being deallocated. Since GDLC allocates a new channel for each **open** subroutine, a **dlcmpx** routine follows each call to the **dlcclose** routine.

Each GDLC supports the **dlcmpx** entry point as its switch table entry for the **open** and **close** subroutines. The file system calls this entry point from the process environment only.

Parameters

Item	Description
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 parameter is set to a value of -1 by GDLC.
channame	Points 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 deallocated.

Return Values

The following return values are defined in the **/usr/include/sys/errno.h** file:

Value	Description
0	Indicates a successful operation.
EBADF	Indicates a bad file number.
EINVAL	Indicates an invalid value.

Related reference:

"dlcopen Entry Point of the GDLC Device Manager"

"open Subroutine Interface for Data Link Control (DLC) Devices" on page 18

Related information:

open subroutine

ddmpx subroutine

dlcopen Entry Point of the GDLC Device Manager

Purpose

Opens a generic data link control (GDLC) channel.

Syntax

```
#include <sys/device.h>
#include <sys/gdlextcb.h>

int dlcopen ( devno, devflag, chan, ext)
```

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

Description

The **dlcopen** entry point 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 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.

Each GDLC supports the **dlcopen** entry point as its switch table entry for the **open** and **openx** subroutines. The file system calls this entry point from the process environment only.

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

Item	Description
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. One **dev_t** device number exists 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 are 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 nonblocking 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 the **open** subroutine.

Return Values

The following return values are defined in the **/usr/include/sys/errno.h** file.

Value	Description
0	Indicates a successful operation.
ECHILD	Indicates that the device manager cannot create a kernel process.
EINVAL	Indicates an invalid value.
ENODEV	Indicates that no such device handler is present.
ENOMEM	Indicates insufficient resources to satisfy the **open** subroutine.
EFAULT	Indicates that a kernel service, such as the **copyin** or **initp** kernel service was unsuccessful.

Related reference:

"dlcclose Entry Point of the GDLC Device Manager" on page 2

Related information:

ddopen subroutine

initp subroutine

dlcread Entry Point of the GDLC Device Manager

Purpose

Reads receive data from generic data link control (GDLC).

Syntax

```
#include <sys/device.h>
#include <sys/gdlextcb.h>
int dlcread (devno, uiop, chan, ext)
```

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

Description

The **dlcread** entry point is called when a user 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:

Field	Description
`uio_iov`	Points to an **iovec** structure.
`uio_iovcnt`	Indicates the number of elements in the **iovec** structure. This must be set to a value of 1. Vectored read operations are not supported.
`uio_offset`	Indicates 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 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`	Specifies initially the total byte count of the receive data area. GDLC decrements this count for each packet byte received using the **uiomove** kernel service.
`iovec structure`	Contains the starting address and length of the received data.
`iov_base`	Specifies where GDLC writes the address of the received data. This field is a variable in the **iovec** structure.
`iov_len`	Contains the byte length of the data. This field is a variable in the **iovec** structure.

Each GDLC supports the **dlcread** entry point as its switch table entry for the **readx** subroutine. The file system calls this entry point from the process environment only.

Parameters

Item	Description
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. One **dev_t** device number exists 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.

Return Values

Successful read operations and those truncated due to limited user data space each return a value of 0 (zero). 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.

The following return values are defined in the **/usr/include/sys/errno.h** file:

Value	Description
EBADF	Indicates a bad file number.
EINTR	Indicates that a signal interrupted the routine before it received data.
EINVAL	Indicates an invalid value.
ENOMEM	Indicates insufficient resources to satisfy the read operation.

Related reference:

"dlcwrite Entry Point of the GDLC Device Manager" on page 10

Related information:

readx subroutine

fp_lseek subroutine

uiomove subroutine

dlcselect Entry Point of the GDLC Device Manager

Purpose

Selects for asynchronous criteria from generic data link control (GDLC), such as receive data completion and exception conditions.

Syntax

```
#include <sys/device.h>
#include <sys/poll.h>
#include <sys/gdlextcb.h>
int dlcselect (devno, events, reventp, chan)
```

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

Description

The **dlcselect** entry point is called when a user application program invokes a **select** or **poll** subroutine. This allows the user to select receive data or exception conditions. The **POLLOUT** 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 *reventp* (returned events) parameter (passed by reference) 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 **POLLSYNC** flag in the *events* parameter. If this flag is true, the routine returns because the event request was a synchronous request. If the **POLLSYNC** 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, 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 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.

Each GDLC supports the **dlcselect** entry point as its switch table entry for the **select** or **poll** subroutines. The file system calls this entry point from the process environment only.

Parameters

Item	Description
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. One **dev_t** device number exists for each type of GDLC, such as Ethernet, Token-Ring, or SDLC.
events	Identifies the events to check. The following events are: **POLLIN** Read selection. **POLLOUT** Write selection. This is not supported by GDLC. **POLLPRI** Exception selection. **POLLSYNC** 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

The following return values are defined in the **/usr/include/sys/errno.h** file:

Value	Description
0	Indicates a successful operation.
EBADF	Indicates a bad file number.
EINTR	Indicates that a signal interrupted the subroutine before it found any of the selected events.
EINVAL	Indicates that the specified **POLLOUT** write selection is not supported.

Related information:

select subroutine

poll subroutine

fp_select subroutine

dlcwrite Entry Point of the GDLC Device Manager

Purpose

Writes transmit data to generic data link control (GDLC).

Syntax

```
#include <sys/uio.h>
#include <sys/device.h>
#include <sys/gdlextcb.h>
int dlcwrite (devno, uiop, chan, ext)
```

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

Description

The **dlcwrite** entry point is called when a user 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:

Field	Description
`uio_iov`	Points to an **iovec** structure.
`uio_iovcnt`	Indicates the 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`	Specifies 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 **fp_open** kernel service to GDLC.
`uio_resid`	Contains the total byte count of the transmit data area for application users. For kernel users, GDLC ignores this field since the communications memory buffer (**mbuf**) also carries this information.
`iovec structure`	Contains the starting address and length of the transmit.
`iov_base`	Specifies 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`	Specifies 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.

Each GDLC supports the **dlcwrite** entry point as its switch table entry for the **writex** subroutine. The file system calls this entry point from the process environment only.

Parameters

Item	Description
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. One **dev_t** device number exists 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 `uio_fmode` field indicates an application subroutine or in the kernel space if the `uio_fmode` field indicates a kernel subroutine.

Return Values

The following return values are defined in the **/usr/include/sys/errno.h** file:

Value	Description
0	Indicates a successful operation.
EAGAIN	Indicates that transmit is temporarily blocked and a sleep cannot be issued.
EBADF	Indicates a bad file number (application).
EINTR	Indicates that a signal interrupted the routine before it could complete successfully.
EINVAL	Indicates an invalid value, such as too much data for a single packet.
ENOMEM	Indicates insufficient resources to satisfy the **write** subroutine, such as a lack of communications memory buffers (**mbufs**).
ENXIO	Indicates an invalid file pointer (kernel).

Related reference:

"dlcmpx Entry Point of the GDLC Device Manager" on page 4

"dlcread Entry Point of the GDLC Device Manager" on page 7

Related information:

writex subroutine

fp_lseek subroutine

Datagram Data Received Routine for DLC

Purpose

Receives a datagram packet each time it is coded by the kernel user and called by generic data link control (GDLC).

Syntax

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

int (*dlc_open_ext.rcvd_fa)( m,  ext)
struct mbuf *m;
struct dlc_io_ext *ext;
```

Description

The DLC Datagram Data Received routine receives a datagram packet each time it is coded by the kernel user and called by GDLC.

Each GDLC supports a subset of the data-received routines. It is critical to performance that the Datagram Data Received routine be coded to minimize the amount of time spent prior to returning to the GDLC that called it.

Parameters

Item	Description
m	Points 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 read operations.

Return Values

Item	Description
DLC_FUNC_OK	Indicates that the received datagram **mbuf** data has been accepted.
DLC_FUNC_RETRY	Indicates that 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.

Related reference:

"readx Subroutine Interface for Data Link Control (DLC) Devices" on page 22

Related information:

Generic Data Link Control (GDLC) Environment Overview

Exception Condition Routine for DLC

Purpose

Notifies the kernel user each time an asynchronous event occurs in generic data link control (GDLC).

Syntax

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

int (*dlc_open_ext.excp_fa)( ext)
struct dlc_getx_arg *ext;
```

Description

The DLC Exception Condition routine notifies the kernel user each time an asynchronous event occurs, such as **DLC_SAPD_RES** (SAP-disabled) or **DLC_CONT_RES** (contacted), in GDLC.

Each GDLC supports a subset of the data-received routines. It is critical to performance that the Exception Condition routine for DLC be coded to minimize the amount of time spent prior to returning to the GDLC that called it.

Parameters

Item	Description
ext	Specifies the same structure for a **dlc_getx_arg** (get exception) ioctl subroutine.

Return Values

Item	Description
DLC_FUNC_OK	Indicates that 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.

Related reference:

"Parameter Blocks by ioctl Operation for DLC" on page 28
Each command operation has a specific parameter block associated with the command pointed to by the *arg* pointer. Some parameters are sent to the generic data link control (GDLC) and others are returned. The ioctl command operations for DLC are:

Related information:

ioctl subroutine

Generic Data Link Control (GDLC) Environment Overview

ioctl Subroutine Interface for Data Link Control (DLC) Devices

Purpose

Transfers special commands to generic data link control (GDLC) using a file descriptor.

Syntax

```
#include <sys/ioctl.h>
#include <sys/devinfo.h>
#include <sys/gdlextcb.h>

int ioctl ( fildes, op, arg);
```

Description

The **ioctl** subroutine initiates various 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 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.

Each GDLC supports the **ioctl** subroutine interface via its **dlcioctl** entry point. This subroutine may be called from the process environment only.

Parameters

Item	Description
fildes	Specifies the file descriptor of the target GDLC.
op	Specifies the operation to be performed by GDLC.
arg	Specifies the address of the parameter block.

Return Values

Item	Description
0	Indicates a successful operation.

If an error occurs, a value of -1 is returned with one of the following error values available using the **errno** global variable, as defined in the **/usr/include/sys/errno.h** file:

Value	Description
EBADF	Indicates a bad file number.
EINVAL	Indicates an invalid argument.
ENOMEM	Indicates insufficient resources to satisfy the **ioctl** subroutine.

Related reference:

"ioctl Operations (op) for DLC"

"Parameter Blocks by ioctl Operation for DLC" on page 28
Each command operation has a specific parameter block associated with the command pointed to by the *arg* pointer. Some parameters are sent to the generic data link control (GDLC) and others are returned. The ioctl command operations for DLC are:

Related information:

ioctl subroutine

ioctl Operations (op) for DLC

Syntax

```
#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  DLC_ADD_FUNC_ADDR    17
#define  DLC_DEL_FUNC_ADDR    18
#define  DLC_DEL_GRP         19
#define  IOCINFO          /* see  /usr/include/sys/ioctl.h */
```

Description

Note: If the operation's notification is returned asynchronously to the user by way of exception, application users should refer to **DLC_GET_EXCEP ioctl** operation for DLC and kernel users should refer to Exception Condition Routine for DLC for more information.

Each GDLC supports a subset of ioctl subroutine operations. These ioctl operations are selectable through the **fp_ioctl** kernel service or the **ioctl** subroutine. They may be called from the process environment only.

The following ioctl command operations are supported for generic data link control (GDLC):

Operation	Description
DLC_ADD_FUNC_ADDR	Adds a group or multicast receive functional address to a port. This command allows additional functional address bits to be added to the current receive functional address mask, as supported by the individual device handlers. See device handler specifications to determine which address values are supported. **Note:** Currently, token ring is the only local area network (LAN) protocol supporting functional addresses.
DLC_ADD_GRP	Adds 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 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 by way of exception.
DLC_DEL_GRP	Removes a group or multicast address that was previously added to a port with a **DLC_ENABLE_SAP** or **DLC_ADD_GRP** ioctl operation.
DLC_DEL_FUNC_ADDR	Removes a group or multicast receive functional address from a port. This command removes functional address bits from the current receive functional address mask, as supported by the individual device handlers. See device handler specifications to determine which address values are supported. **Note:** Currently, token ring is the only local area network protocol supporting functional addresses.
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 later by way of exception.
DLC_ENABLE_SAP	Enables an 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 later by way of 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 by their exception 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 by way of exception.
DLC_QUERY_LS	Queries an LS.
DLC_QUERY_SAP	Queries an 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 by way of 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 by way of exception.
DLC_TRACE	Traces LS activity.
IOCINFO	Returns a structure that describes the device. Refer to the description of the **/usr/include/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.

Related reference:

"ioctl Subroutine Interface for Data Link Control (DLC) Devices" on page 13

"Parameter Blocks by ioctl Operation for DLC" on page 28

Each command operation has a specific parameter block associated with the command pointed to by the *arg* pointer. Some parameters are sent to the generic data link control (GDLC) and others are returned. The ioctl command operations for DLC are:

Related information:

Generic Data Link Control (GDLC) Environment Overview

I-Frame Data Received Routine for DLC
Purpose

Receives a normal sequenced data packet each time it is coded by the kernel user and called by generic data link control (GDLC).

Syntax

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

int (*dlc_open_ext.rcvi_fa)( m,  ext)
struct mbuf *m;
struct dlc_io_ext *ext;
```

Description

The DLC I-Frame Data Received routine receives a normal sequenced data packet each time it is coded by the kernel user and called by GDLC.

Each GDLC supports a subset of the data-received routines. It is critical to performance that the I-Frame Data Received routine be coded to minimize the amount of time spent prior to returning to the GDLC that called it.

Parameters

Item	Description
m	Points 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.

Return Values

Item	Description
DLC_FUNC_OK	Indicates that the received **I-frame** function call is accepted.
DLC_FUNC_BUSY	Indicates that 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	Indicates that 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.

Related reference:

"Parameter Blocks by ioctl Operation for DLC" on page 28
Each command operation has a specific parameter block associated with the command pointed to by the *arg* pointer. Some parameters are sent to the generic data link control (GDLC) and others are returned. The ioctl command operations for DLC are:

Related information:

ioctl subroutine

Generic Data Link Control (GDLC) Environment Overview

Network Data Received Routine for DLC
Purpose

Receives network-specific data each time it is coded by the kernel user and called by generic data link control (GDLC).

Syntax

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

int (*dlc_open_ext.rcvn_fa)( m,  ext)
struct mbuf *m;
struct dlc_io_ext *ext;
```

Description

The DLC Network Data Received routine receives network-specific data each time the routine is coded by the kernel user and called by GDLC.

Each GDLC supports a subset of the data-received routines. It is critical to performance that the Network Data Received routine be coded to minimize the amount of time spent prior to returning to the GDLC that called it.

Parameters

Item	Description
m	Points 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 read operations.

Return Values

Item	Description
DLC_FUNC_OK	Indicates that the received network **mbuf** data has been accepted.
DLC_FUNC_RETRY	Indicates that the received network **mbuf** data 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 cause a disabling of the service access point.

Related reference:

"readx Subroutine Interface for Data Link Control (DLC) Devices" on page 22

Related information:

Generic Data Link Control (GDLC) Environment Overview

open Subroutine Extended Parameters for DLC

Purpose

Alters certain defaulted parameters for an extended **open** (**openx**) subroutine.

Syntax

```
struct dlc_open_ext
{
   __ulong32_t  maxsaps;
   int (* rcvi_fa)();
   int (* rcvx_fa)();
   int (* rcvd_fa)();
   int (* rcvn_fa)();
   int (* excp_fa)();
};
```

Description

An extended **open** or **openx** subroutine can be issued to alter certain 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 generic data link control (GDLC) calls to notify about asynchronous events, such as receive data available.

The *maxsaps* parameter is optional for both the application and the kernel user. The other 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 which frees the communication's memory buffer (**mbuf**) passed on the call.

These DLC extended parameters for the **open** subroutine are part of the data link control in BOS Extensions 2 for the device manager you are using.

See the **/usr/include/sys/gdlextcb.h** file for more details on GDLC structures.

Parameters

Item	Description
maxsaps	Specifies the maximum number of SAPs the user channel uses to start and run concurrently. Any value from 1 to 127 can be specified. If the default value of 1 is desired, the user must set the field to 0 (zero) before issuing the **open** subroutine.
rcvi_fa	Points to the address of a user I-Frame Data Received routine that handles the sequenced receive data completions. This field is valid for kernel users only and must be set to 0 (zero) by application users.
rcvx_fa	Points to the address of a user XID Data Received routine that handles the exchange ID receive data completions.
rcvd_fa	Points to the address of a user Datagram Data Received routine that handles the datagram receive data completions.
rcvn_fa	Points to the address of a user Network Data Received routine that handles the network receive data completions.
excp_fa	Points to the address of a user Exception Condition routine that handles the exception conditions, such as **DLC_SAPE_RES** (SAP-enabled) or **DLC_CONT_RES** (LS-contacted).

Related reference:

"Parameter Blocks by ioctl Operation for DLC" on page 28
Each command operation has a specific parameter block associated with the command pointed to by the *arg* pointer. Some parameters are sent to the generic data link control (GDLC) and others are returned. The ioctl command operations for DLC are:

Related information:

open subroutine

List of Kernel Routines for DLC

open Subroutine Interface for Data Link Control (DLC) Devices

Purpose

Opens the generic data link control (GDLC) device manager by special file name.

Syntax

```
#include <fcntl.h>
#include <sys/gdlextcb.h>

int open ( path,  oflag,  mode)
```

or

```
int openx (path, oflag, mode,  ext)
```

Description

The **open** subroutine allows the application user to open a GDLC device manager by specifying the DLC special file name and the target device handler 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.

Each GDLC supports the **open** subroutine interface by way of its **dlcopen** and **dlcmpx** entry points. This subroutine may be called from the process environment only.

Parameters

Item	Description
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.

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 values available using the **errno** global variable, as defined in the **/usr/include/sys/errno.h** file:

Value	Description
ECHILD	Indicates that the device manager cannot create a kernel process.
EINVAL	Indicates an invalid value.
ENODEV	Indicates that no such device handler is present.
ENOMEM	Indicates insufficient resources to satisfy the **open** subroutine.
EFAULT	Indicates that a kernel service, such as the **copyin** or **initp** kernel service, has failed.

Related reference:

"close Subroutine Interface for Data Link Control (DLC) Devices" on page 1

"dlcmpx Entry Point of the GDLC Device Manager" on page 4

Related information:

copyin subroutine

read Subroutine Extended Parameters for DLC

Purpose

Provide generic data link control (GDLC) with a structure to return data types and service access point (SAP) and link station (LS) correlator.

Syntax

```
#define  DLC_INFO   0x80000000
#define  DLC_XIDD   0x40000000
#define  DLC_DGRM   0x20000000
#define  DLC_NETD   0x10000000
#define  DLC_OFLO   0x00000002
#define  DLC_RSPP   0x00000001

struct dlc_io_ext
{
   __ulong32_t  sap_corr;
   __ulong32_t  ls_corr;
   __ulong32_t  flags;
   __ulong32_t  dlh_len;
};
```

Description

An extended read or readx subroutine must be issued by an application user to provide GDLC with a structure to return the type of data and the SAP and LS correlator.

Parameters

sap_corr
Specifies the user's SAP identifier of the received data.

ls_corr
Specifies the user's LS identifier of the received data.

flags
Specifies flags for the readx subroutine. The following flags are supported:

DLC_INFO
Indicates that normal sequenced data has been received for a link station using an I-Frame Data Received routine. 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
Indicates that exchange identification (XID) data has been received for a link station using an XID Data Received routine. 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
Indicates that a datagram has been received for an LS using a Datagram Data Received routine. 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
Indicates that data has been received from the network for a service access point using a Network Data Received routine. This may be link-establishment data such as X.21 call-progress signals or Smartmodem 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
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
Indicates that the XID received requires an XID response to be sent back to the remote link station.

dlh_len
Specifies data link header length. This field has a 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, this field 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 is included in the read operation.

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) .

These DLC extended parameters for the read subroutine are part of the data link control in BOS Extensions 2 for the device manager you are using.

Related reference:

"writex Subroutine Interface for Data Link Control (DLC) Devices" on page 26

"Parameter Blocks by ioctl Operation for DLC" on page 28
Each command operation has a specific parameter block associated with the command pointed to by the *arg* pointer. Some parameters are sent to the generic data link control (GDLC) and others are returned. The ioctl command operations for DLC are:

Related information:

List of Kernel Routines for DLC

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)
```

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. Generic data link control (GDLC) does not break up received packets into multiple user data areas.

Each GDLC supports the **readx** subroutine interface via its **dlcread** entry point. This subroutine can be called from the process environment only.

Parameters

Item	Description
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. **Note:** It is the user's responsibility to set the *ext* parameter area to 0 (zero) before 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 0 (zero) is returned.

If an error occurs, a value of -1 is returned with one of the following error numbers available using the **errno** global variable, as defined in the **/usr/include/sys/errno.h** file:

Value	Description
EBADF	Indicates a bad file number.
EINTR	Indicates that a signal interrupted the subroutine before it received data.
EINVAL	Indicates an invalid value.
ENOMEM	Indicates insufficient resources to satisfy the read operation.

Related reference:

"writex Subroutine Interface for Data Link Control (DLC) Devices" on page 26

"Parameter Blocks by ioctl Operation for DLC" on page 28

Each command operation has a specific parameter block associated with the command pointed to by the *arg* pointer. Some parameters are sent to the generic data link control (GDLC) and others are returned. The ioctl command operations for DLC are:

"write Subroutine Extended Parameters for DLC" on page 24

Related information:

open subroutine

uiomove subroutine

List of Kernel Routines for DLC

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)
```

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: Generic data link control (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 be done reasonably only 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.

Each GDLC supports the **select** subroutine interface via its **dlcselect** entry point. This subroutine can be called from the process environment only.

Parameters

Item	Description
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 to 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. Also, 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 (zero).

If an error occurs, a value of -1 is returned with one of the following error values available using the **errno** global variable, as defined in the **/usr/include/sys/errno.h** file:

Item	Description
EBADF	Indicates a bad file number.
EINTR	Indicates that a signal interrupted the subroutine before it found any of the selected events.
EINVAL	Indicates that one of the parameters contained an invalid value.

Related information:

select subroutine

Select/Poll Logic for ddwrite and ddread Routines

Generic Data Link Control (GDLC) Environment Overview

write Subroutine Extended Parameters for DLC

Purpose

Provide generic data link control (GDLC) with data types, service access points (SAPs), and link station (LS) correlators.

Syntax

#define DLC_INFO 0x80000000 #define DLC_XIDD 0x40000000 #define DLC_DGRM 0x20000000 #define DLC_NETD 0x10000000

__ulong32_t *sap_corr*; **__ulong32_t** *ls_corr*; **__ulong32_t** *flags*; **__ulong32_t** *dlh_len*; **};**

Description

An extended **write** or **writex** subroutine must be issued by an application or kernel user to provide GDLC with data types, SAPs, and LS correlators.

These DLC extended parameters for the **write** subroutine are part of the data link control in BOS Extensions 2 for the device manager you are using.

Parameters

Item	Description
sap_corr	Specifies the GDLC SAP correlator of the write data. This field must contain the same correlator value passed back form GDLC in the `gdlc_sap_corr` field when the SAP was enabled.
dlh_len	Not used for writes.
ls_corr	Specifies the GDLC LS correlator of the write data. This field must contain the same correlator value passed back from GDLC in the `gdlc_ls_corr` field when the LS was started.
flags	Specifies flags for the **writex** subroutine. The following flags are supported: **DLC_INFO** 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** Requests an exchange identification (XID) non-sequenced command or response packet 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** Requests a datagram packet to be sent. A datagram is an unnumbered information (UI) response. This request is valid any time the target link station has been started. **DLC_NETD** 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.

Related reference:

"readx Subroutine Interface for Data Link Control (DLC) Devices" on page 22

Related information:

write subroutine

List of Kernel Routines for DLC

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)
```

Description

Four types of data can be sent to generic data link control (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** kernel service 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 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, 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 system blocked or nonblocked file status flags. These are set for each channel using the **O_NDELAY** and **O_NONBLOCK** values passed on **open** or **fcntl** subroutines with the **F_SETFD** parameter.

GDLC only looks at the `uio_fmode` field 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.

Each GDLC supports the **writex** subroutine interface via its **dlcwrite** entry point. This subroutine may be called from the process environment only.

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 user's discretion.

Parameters

Item	Description
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.

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 values available using the **errno** global variable, as defined in the **/usr/include/sys/errno.h** file.

Value	Description
EAGAIN	Indicates insufficient resources to satisfy the write. For example, the routine was unable to obtain a necessary lock. The user can try again later.
EBADF	Indicates a bad file number.
EINTR	Indicates that a signal interrupted the subroutine before it completed successfully.
EINVAL	Indicates an invalid value, such as too much data for a single packet.
EIO	Indicates that an I/O error has occurred, such as loss of the port.
ENOMEM	Indicates insufficient resources to satisfy the write operation. For example, a lack of communications memory buffers (**mbufs**). The user can try again later.

Related reference:

"read Subroutine Extended Parameters for DLC" on page 20

"readx Subroutine Interface for Data Link Control (DLC) Devices" on page 22

"Parameter Blocks by ioctl Operation for DLC" on page 28

Each command operation has a specific parameter block associated with the command pointed to by the *arg* pointer. Some parameters are sent to the generic data link control (GDLC) and others are returned. The ioctl command operations for DLC are:

Related information:

fcntl subroutine

uiomove subroutine

XID Data Received Routine for DLC

Purpose

Receives an exchange identification (XID) packet each time it is coded by the kernel user and called by generic data link control (GDLC).

Syntax

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

int (*dlc_open_ext.rcvx_fa)( m,  ext)
struct mbuf *m;
struct dlc_io_ext *ext;
```

Description

The DLC XID Data Received routine receives an XID packet each time the routine is coded by the kernel user and called by GDLC.

Each GDLC supports a subset of the data-received routines. It is performance critical that the XID Data Received routine be coded to minimize the amount of time spent prior to returning to the GDLC that called it.

Parameters

Item	Description
m	Points 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.

Return Values

Item	Description
DLC_FUNC_OK	Indicates that the received XID **mbuf** data has been accepted.
DLC_FUNC_RETRY	Indicates that the received XID **mbuf** data 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 may close the link station.

Related reference:

"readx Subroutine Interface for Data Link Control (DLC) Devices" on page 22

Related information:

Generic Data Link Control (GDLC) Environment Overview

Parameter Blocks by ioctl Operation for DLC

Each command operation has a specific parameter block associated with the command pointed to by the *arg* pointer. Some parameters are sent to the generic data link control (GDLC) and others are returned. The ioctl command operations for DLC are:

DLC_ADD_FUNC_ADDR ioctl Operation for DLC

The **DLC_ADD_FUNC_ADDR** ioctl operation is selectable through the **fp_ioctl** kernel service or the **ioctl** subroutine. It can be called from the process environment only.

The following parameter block adds a functional address mask any time a service access point (SAP) has been enabled via **DLC_ENA_SAP** ioctl. Multiple functional address bits may be specified.

```
struct dlc_func_addr
   {
   __ulong32_t  gdlc_sap_corr;   /* GDLC SAP correlator */
   __ulong32_t len_func_addr_mask;   /* length of functional */
      /* address mask */
   uchar_t func_addr_mask[DLC_MAX_ADDR]; /* functional address */
      /* mask */
   };
```

The fields of this ioctl operation are:

Field	Description
`gdlc_sap_corr`	Contains the generic data link control (GDLC) service access point (SAP) correlator being requested to delete a functional address from a port.
`len_func_addr_mask`	Contains the byte length of the functional address mask to be added.
`func_addr_mask`	Contains the functional address mask value to be ORed with the functional address on the adapter. See the individual DLC interface documentation to determine the length and format of this field.

DLC_ADD_GRP ioctl Operation for DLC

The **DLC_ADD_GRP** ioctl operation is selectable through the **fp_ioctl** kernel service or the **ioctl** subroutine. It can be called from the process environment only.

The following parameter block adds a group or multicast receive address:

```
struct dlc_add_grp
{
   __ulong32_t gdlc_sap_corr;   /* GDLC SAP correlator */
   __ulong32_t grp_addr_len;   /* group address length */
   uchar_t grp_addr[DLC_MAX_ADDR]; /* grp addr to be added */
};
```

The fields of this ioctl operation are:

Field	Description
gdlc_sap_corr	Contains the generic data link control (GDLC) service access point (SAP) Correlator being requested to add a group or multicast address to a port.
grp_addr_len	Contains the byte length of the group or multicast address to be added.
grp_addr	Contains the group or multicast address value to be added.

DLC_ALTER ioctl Operation for DLC

The **DLC_ALTER** ioctl operation is selectable through the **fp_ioctl** kernel service or the **ioctl** subroutine. It can be called from the process environment only.

The following parameter block alters a link station's (LS) configuration parameters:

```
#define DLC_MAX_ROUT   20   /* Maximum Size of Routing Info */

struct dlc_alter_arg
   {
   __ulong32_t  gdlc_sap_corr;   /* GDLC SAP correlator */
   __ulong32_t  gdlc_ls_corr;   /* GDLC link station correlator */
   __ulong32_t  flags;   /* Alter Flags */
   __ulong32_t  repoll_time;   /* New Repoll Timeout */
   __ulong32_t  ack_time;   /* New Acknowledge Timeout */
   __ulong32_t  inact_time;   /* New Inactivity Timeout */
   __ulong32_t  force_time;   /* New Force Timeout */
   __ulong32_t  maxif;   /* New Maximum I-Frame Size */
   __ulong32_t  xmit_wind;   /* New Transmit Value */
   __ulong32_t  max_repoll;   /* New Max Repoll Value */
   __ulong32_t  routing_len;   /* Routing Length */
   u_char_t  routing[DLC_MAX_ROUT];  /* New Routing Data */
   __ulong32_t  result_flags;   /* Returned flags */
   };
```

The fields of this ioctl operation are:

Field	Description
gdlc_sap_corr	Indicates the generic data link control (GDLC) service access point (SAP) correlator of the target LS.
gdlc_ls_corr	Indicates the GDLC LS correlator to be altered.

Field	Description
flags	Specifies alter flags. The following flags are supported:

DLC_ALT_RTO
Alter repoll timeout:

0 = Do not alter repoll timeout.

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 timeout value specified in the LS configuration is overridden by the value supplied in the repoll timeout 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 timeout:

0 = Do not alter the acknowledgment timeout.

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 timeout value specified in the LS configuration is overridden by the value supplied in the acknowledgment timeout 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 timeout:

0 = Do not alter inactivity timeout.

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 timeout value specified in the LS configuration is overridden by the value supplied in the inactivity timeout 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 timeout:

0 = Do not alter force halt timeout.

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 timeout value specified in the LS configuration is overridden by the value supplied in the force halt timeout field of the **Alter** command. This new value remains in effect until another value is specified or the LS is halted.

Field	Description
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 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 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 timeout, 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 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 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.

Field	Description
	DLC_ALT_SM1 Set primary SDLC Control mode: 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 secondary SDLC Control mode: 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 notification for Inactivity Time-Out mode: 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 automatic halt for Inactivity Time-Out mode: 0 = Do not alter Inactivity Time-Out mode. 1 = Set Inactivity Time-Out mode to automatic halt.
`repoll_time`	Provides a new value to replace the LS repoll time-out value whenever the **DLC_ALT_RTO** flag is set.
`ack_time`	Provides a new value to replace the LS acknowledgment time-out value whenever the **DLC_ALT_AKT** flag is set.
`inact_time`	Provides a new value to replace the LS inactivity time-out value whenever the alter **DLC_ALT_ITO** flag is set.
`force_time`	Provides a new value to replace the LS force halt time-out value whenever the **DLC_ALT_FHT** flag is set.
`maxif`	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 the receive buffer and only increases the internal value to the allowed maximum.
`xmit_wind`	Provides a new value to replace the LS transmit window count value whenever the **DLC_ALT_XWIN** flag is set.
`max_repoll`	Provides the new value that is to replace the LS maximum repoll count value whenever the **DLC_ALT_MXR** flag is set.
`routing_len`	Provides a new value to replace the LS routing field length whenever the **DLC_ALT_RTE** flag is set.
`routing`	Provides a new value to replace the LS routing data whenever the **DLC_ALT_RTE** flag is set.

Field	Description
result_flags	Returns the following result indicators at the completion of the alter operation, depending on the command: **DLC_MSS_RES** Indicates mode set secondary. Set to 1, this bit indicates that the station mode has been set to secondary as a result of the user issuing an **Alter** (set mode secondary) command. **DLC_MSSF_RES** Indicates mode set secondary was unsuccessful. Set to 1, this bit indicates that the station mode has been not set to secondary as a result of the user 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** Indicates mode set primary. Set to 1, this bit indicates that the station mode has been set to primary as a result of the user issuing an **Alter** (set mode primary) command. **DLC_MSPF_RES** Indicates mode set primary was unsuccessful. Set to 1, this bit indicates that the station mode has not been set to primary as a result of the user issuing an **Alter** (set mode primary) command. This occurs whenever an SDLC LS is already in data phase.

The protocol-dependent area allows additional fields to be provided by a specific protocol type. Corresponding flags may be necessary to support additional fields. This optional data area must directly follow (or append to) the end of the **dlc_alter_arg** structure.

DLC_CONTACT ioctl Operation for DLC

The **DLC_CONTACT** ioctl operation is selectable through the **fp_ioctl** kernel service or the **ioctl** subroutine. It can be called from the process environment only.

The following parameter block contacts a remote station for a particular local link station (LS):

```
struct dlc_corr_arg
   {
   __ulong32_t  gdlc_sap_corr;   /* GDLC SAP correlator */
   __ulong32_t  gdlc_ls_corr;   /* GDLC link station correlator */
   };
```

The fields of this ioctl operation are:

Field	Description
gdlc_sap_corr	Contains the GDLC SAP correlator of the target LS.
gdlc_ls_corr	Contains the GDLC LS correlator to be contacted.

DLC_DEL_FUNC_ADDR ioctl Operation for DLC

The **DLC_DEL_FUNC_ADDR** ioctl operation is selectable through the **fp_ioctl** kernel service or the **ioctl** subroutine. It can be called from the process environment only.

The following parameter block deletes a previously defined functional address mask any time a service access point (SAP) has been enabled with a **DLC_ENA_SAP** ioctl. Multiple functional address bits can be specified.

```
struct dlc_func_addr
   {
   __ulong32_t gdlc_sap_corr;   /* GDLC SAP correlator */
   __ulong32_t len_func_addr_mask;   /* length of functional */
      /* address mask */
   uchar_t func_addr_mask[DLC_MAX_ADDR]; /*functional add. mask */
   };
```

The fields of this ioctl operation are:

Field	Description
gdlc_sap_corr	Indicates the generic data link control (GDLC) service access point (SAP) identifier being requested to delete a functional address from a port.
len_func_addr_mask	Contains the byte length of the functional address mask to be deleted.
func_addr_mask	Contains the functional address mask value to be deleted from with the functional address on the adapter. See the individual DLC interface documentation to determine the length and format of this field.

DLC_DEL_GRP ioctl Operation for DLC

The **DLC_DEL_GRP** ioctl operation is selectable through the **fp_ioctl** kernel service or the **ioctl** subroutine. It can be called from the process environment only.

The following parameter removes a previously defined group or multicast address:

```
struct dlc_add_grp
   {
    __ulong32_t  gdlc_sap_corr;      /*GDLC SAP correlator  */
    __ulong32_t grpaddr_len;      /*group address length */
    uchar_t grp_addr[DLC_MAX_ADDR]; /*group address to be
                  removed        */
   };
```

The fields of this ioctl operation are:

Field	Description
gdlc_sap_corr	Indicates the generic data link control (GDLC) service access point (SAP) identifier being requested to remove a group or multicast address from a port. This field is known as the GDLC SAP Correlator field.
grp_addr_len	Contains the byte length of the group or multicast address to be removed.
grp_addr	Contains the group or multicast address to be removed.

DLC_DISABLE_SAP ioctl Operation for DLC

The **DLC_DISABLE_SAP** ioctl operation is selectable through the **fp_ioctl** kernel service or the **ioctl** subroutine. It can be called from the process environment only.

The following parameter block disables a service access point (SAP):

```
struct dlc_corr_arg
   {
   __ulong32_t  gdlc_sap_corr;   /* GDLC SAP correlator */
   __ulong32_t gdlc_ls_corr;   /* <not used for disabling a SAP> */
   };
```

The fields of this ioctl operation are:

Field	Description
gdlc_sap_corr	Contains GDLC SAP correlator. The field indicates the GDLC SAP identifier to be disabled.
gdlc_ls_corr	Contains GDLC LS correlator. The GDLC LS identifier is returned to the user as soon as resources are determined to be available. This correlator must accompany all commands associated with this LS.

DLC_ENABLE_SAP ioctl Operation for DLC

The **DLC_ENABLE_SAP** ioctl operation is selectable through the **fp_ioctl** kernel service or the **ioctl** subroutine. It can be called from the process environment only.

The following parameter block enables a service access point (SAP):

```
#define DLC_MAX_NAME   20
#define DLC_MAX_GSAPS   7
#define DLC_MAX_ADDR   8

#define  DLC_ESAP_NTWK   0x40000000
#define  DLC_ESAP_LINK   0x20000000
#define  DLC_ESAP_PHYC   0x10000000
#define  DLC_ESAP_ANSW   0x08000000
#define  DLC_ESAP_ADDR   0x04000000
struct dlc_esap_arg
   {

   __ulong32_t  gdlc_sap_corr;
   __ulong32_t  user_sap_corr;
   __ulong32_t  len_func_addr_mask;
   uchar_t  func_addr_mask [DLC_MAX_ADDR];

   __ulong32_t  len_grp_addr;
   uchar_t  grp_addr [DLC_MAX_ADDR];
   __ulong32_t  max_ls;
   __ulong32_t  flags;
   __ulong32_t  len_laddr_name;

   u_char_t  laddr_name [DLC_MAX_NAME];
   u_char_t  num_grp_saps;
   u_char_t  grp_sap [DLC_MAX_GSAPS];
   u_char_t res1[3];
   u_char_t  local_sap;
   };
```

The fields of this ioctl operation are:

Field	Description
gdlc_sap_corr	Specifies the generic data link control's (GDLC) SAP identifier that is returned to the user. This correlator must accompany all subsequent commands associated with this SAP.
user_sap_corr	Specifies an identifier or correlator the user wishes to have returned on all SAP results from GDLC. It allows the user of multiple SAPs to choose a correlator to route the SAP-specific results.
len_func_addr_mask	Specifies the byte length of the following functional address mask. This field must be set to 0 if no functional address is required. Length values of 0 through 8 are supported.

Field	Description
func_addr_mask	Specifies 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 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	Specifies the byte length of the following group address. This field must be set to 0 (zero) if no group address is required. Length values of 0 through 8 are supported.
grp_addr	Specifies 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	Specifies the maximum number of link stations (LSs) allowed to operate concurrently on a particular SAP. The protocol used determines the values for this field.
flags	Supports the following flags of the **DLC_ENABLE_SAP** ioctl operation: **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. Specifies whether the local address or name field contains an address or a name: 0 = Local name specified (default) 1 = Local address specified **DLC_ESAP_ANSW** Teleprocessing autocall or autoanswer: 0 = Manual call and answer (default) 1 = Automatic call and answer
len_laddr_name	Specifies the byte length of the following local address or name. Length values of 1 through 20 are supported.

Field	Description
laddr_name	Contains the unique network name or address of the user local SAP as indicated by the **DLC_ESAP_ADDR** flag. Some protocols allow the local SAP to be identified by name (for example, Name-Discovery Services) and others by address (for example, Address Resolve Procedures). Other protocols such as Synchronous Data Link Control (SDLC) do not identify the local SAP. Check the individual DLC's usage of this field for the protocol you are operating.
num_grp_saps	Specifies the number of group SAPs to which the user's local SAP responds. If no group SAPs are needed, this field must contain a 0. Up to seven group SAPs can be specified.
grp_sap	Contains the specific group SAP values to which the user local SAP responds (seven maximum).
local_sap	Specifies the local SAP address opened. Receive packets with this LSAP value indicated in the destination SAP field are routed to the LSs opened under this particular SAP.

The protocol-specific data area allows parameters to be defined by the specific GDLC device manager, such as X.21 call-progress signals or Smartmodem call-establishment data. This optional data area must directly follow (or append to) the end of the **dlc_esap_arg** structure.

DLC_ENTER_LBUSY ioctl Operation for DLC

The **DLC_ENTER_LBUSY** ioctl operation is selectable through the **fp_ioctl** kernel service or the **ioctl** subroutine. It can be called from the process environment only.

The following parameter block enters local busy mode on a particular link station (LS):

```
struct dlc_corr_arg
   {
   __ulong32_t  gdlc_sap_corr;   /* GDLC SAP correlator */
   __ulong32_t  gdlc_ls_corr;   /* GDLC link station correlator */
   };
```

The fields of this ioctl operation are:

Field	Description
gdlc_sap_corr	Contains the GDLC SAP correlator of the target LS.
gdlc_ls_corr	Contains the GDLC LS correlator to enter local busy mode.

DLC_ENTER_SHOLD ioctl Operation for DLC

The **DLC_ENTER_SHOLD** ioctl operation is selectable through the **fp_ioctl** kernel service or the **ioctl** subroutine. It can be called from the process environment only.

The following parameter block enters short hold mode on a particular link station (LS):

```
struct dlc_corr_arg
   {
   __ulong32_t  gdlc_sap_corr;   /* GDLC SAP correlator */
   __ulong32_t  gdlc_ls_corr;   /* GDLC link station correlator */
   };
```

The fields of this ioctl operation are:

Field	Description
gdlc_sap_corr	Contains the generic data link control (GDLC) service access point (SAP) correlator of the target LS.
gdlc_ls_corr	Contains the GDLC LS correlator to enter short hold mode.

DLC_EXIT_LBUSY ioctl Operation for DLC

The **DLC_EXIT_LBUSY** ioctl operation is selectable through the **fp_ioctl** kernel service or the **ioctl** subroutine. It can be called from the process environment only.

The following parameter block exits local busy mode on a particular link station (LS):

```
struct dlc_corr_arg
   {
   __ulong32_t  gdlc_sap_corr;   /* GDLC SAP correlator */
   __ulong32_t  gdlc_ls_corr;   /* GDLC link station correlator */
   };
```

The fields of this ioctl operation are:

Field	Description
gdlc_sap_corr	Contains the GDLC SAP correlator of the target LS.
gdlc_ls_corr	Contains the GDLC LS correlator to exit local busy mode.

DLC_EXIT_SHOLD ioctl Operation for DLC

The **DLC_EXIT_SHOLD** ioctl operation is selectable through the **fp_ioctl** kernel service or the **ioctl** subroutine. It can be called from the process environment only.

The following parameter block exits short hold mode on a particular link station (LS):

```
struct dlc_corr_arg
   {
   __ulong32_t  gdlc_sap_corr;   /* GDLC SAP correlator */
   __ulong32_t  gdlc_ls_corr;   /* GDLC link station correlator */
   };
```

The fields of this ioctl operation are:

Field	Description
gdlc_sap_corr	Contains the generic data link control (GDLC) service access point (SAP) correlator of the target LS.
gdlc_ls_corr	Contains the GDLC LS correlator to exit short hold mode.

DLC_GET_EXCEP ioctl Operation for DLC

The **DLC_GET_EXCEP** ioctl operation is selectable through the **fp_ioctl** kernel service or the **ioctl** subroutine. It can be called from the process environment only.

The following parameter block returns asynchronous exception notifications to the application user:

```
struct dlc_getx_arg
   {
   __ulong32_t  user_sap_corr;   /* user SAP corr - RETURNED */
   __ulong32_t  user_ls_corr;   /* user ls corr - RETURNED */
   __ulong32_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 */
};
```

The fields of this ioctl operation are:

Field	Description
user_sap_corr	Indicates the user service access point (SAP) correlator for this exception.
user_ls_corr	Indicates the user link station (LS) correlator for this exception.
result_ind	Result indicators:

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 enables: an extended result. Set to 1, this bit indicates that the SAP is active and ready for LSs to be started.

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.

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 can 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 timeout). 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 synchronous data link control (SDLC) links when negotiating a point-to-point connection, for example.

Field	Description
result_code	Indicates the 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. **DLC_SUCCESS** The result indicated was successful. **DLC_PROT_ERR** Protocol error. **DLC_BAD_DATA** A bad data compare on a TEST. **DLC_NO_RBUF** No remote buffering on test. **DLC_RDISC** Remote initiated discontact. **DLC_DISC_TO** Discontact abort timeout. **DLC_INACT_TO** Inactivity timeout. **DLC_MSESS_RE** Mid session reset. **DLC_NO_FIND** Cannot find the remote name. **DLC_INV_RNAME** Invalid remote name. **DLC_SESS_LIM** Session limit exceeded. **DLC_LST_IN_PRGS** Listen already in progress. **DLC_LS_NT_COND** LS unusual network condition. **DLC_LS_ROUT** Link station resource outage. **DLC_REMOTE_BUSY** Remote station found, but busy. **DLC_REMOTE_CONN** Specified remote is already connected. **DLC_NAME_IN_USE** Local name already in use. **DLC_INV_LNAME** Invalid local name. **DLC_SAP_NT_COND** SAP network unusual network condition. **DLC_SAP_ROUT** SAP resource outage. **DLC_USR_INTRF** User interface error. **DLC_ERR_CODE** Error in the code has been detected. **DLC_SYS_ERR** System error.

Field	Description
result_ext	Indicates 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 block enables a service access point (SAP) result extension:

```
struct dlc_sape_res
   {
   __ulong32_t  max_net_send;   /* maximum write network data length */
   __ulong32_t  lport_addr_len;   /* local port network address length */
   u_char_t  lport_addr[DLC_MAX_ADDR];/* the local port address */
   };
```

The fields of this extension are:

Field	Description
max_net_send	Indicates 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**/**mbufs** page cluster size, but is not necessarily limited to a single **mbuf** structure since **mbuf** clusters can be linked.
lport_addr_len	Indicates the byte length of the local port network address.
lport_addr	Indicates the hexadecimal value of the local port network address.

DLC_STAS_RES Link Station Started Result Extension: The following parameter block starts a link station (LS) result extension:

```
struct dlc_stas_res
   {
   ulong32_t  maxif;   /* max size of the data sent */
      /* on a write */
   ulong32_t  rport_addr_len;   /* remote port network address */
      /* length */
   u_char_t  rport_addr[DLC_MAX_ADDR]; /* remote port address */
   ulong32_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   */
   ulong32_t  max_data_off;   /* the maximum data offsets for sends*/
   };
```

The fields of this extension are:

Field	Description
maxif	Contains the maximum byte size allowable for user data. This value is derived from the value supplied by the user at the 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 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 exceed the length of a single **mbuf** cluster. The returned value never exceeds the value supplied by the user, but may be smaller if buffering is not large enough to hold the specified value.

Field	Description
rport_addr_len	Contains the byte length of the remote port network address.
rport_addr	Contains the hexadecimal value of the remote port network address.
rname_len	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 0 (zero). Network names can be 1 to 20 characters in length.
rname	Contains the name used by the remote SAP. This field is valid only if name-discovery procedures were used to locate the remote station.
rsap	Contains the hexadecimal value of the remote SAP address.
max_data_off	Contains the write data offset in bytes of a communications **mbuf** cluster where transmit data must minimally begin. This allows ample room for the DLC and MAC headers to be inserted if needed. Some DLCs may be able to prepend additional **mbuf** clusters for their headers, and in this case will set this field to 0 (zero). This field is only valid for kernel users that pass in a communications **mbuf** structure on write operations. **Note:** 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.

Related reference:
"DLC_START_LS ioctl Operation for DLC" on page 46

DLC_STAH_RES Link Station Halted Result Extension: The following parameter block halts the link station (LS) result extension:

```
struct dlc_stah_res
   {
   __ulong32_t  conf_ls_corr;   /* conflicting link station corr */
   };
```

The field of this extension is:

Field	Description
conf_ls_corr	Indicates conflicting link station correlator. Contains the user's link station identifier that already has the specified remote station attached.

This extension is valid only if the result code value indicates -936 (specified remote is already connected).

DLC_RADD_RES Remote Address/Name Change Result Extension: The following parameter block changes the remote address or name of the result extension:

```
struct dlc_radd_res
   {
   __ulong32_t  rname_len;   /* remote network name/addr length */
   u_char  rname[DLC_MAX_NAME];/* remote network name/addr */
   };
```

The fields of this extension are:

Field	Description
rname_len	Indicates the remote network address or name length. Contains the byte length of the updated remote SAP's network address or name.
rname	Contains the updated address or name being used by the remote SAP.

DLC_HALT_LS ioctl Operation for DLC

The **DLC_HALT_LS** ioctl operation is selectable through the **fp_ioctl** kernel service or the **ioctl** subroutine. It can be called from the process environment only.

The following parameter block halts a link station (LS):

```
struct dlc_corr_arg
   {
   __ulong32_t  gdlc_sap_corr;   /* GDLC SAP correlator */
   __ulong32_t  gdlc_ls_corr;   /* GDLC link station correlator */
   };
```

The fields of this ioctl operation are:

Field	Description
gdlc_sap_corr	Contains the GDLC SAP correlator: The GDLC SAP identifier of the target LS.
gdlc_ls_corr	Contains the GDLC LS correlator: The GDLC LS identifier to be halted.

DLC_QUERY_LS ioctl Operation for DLC

The **DLC_QUERY_LS** ioctl operation is selectable through the **fp_ioctl** kernel service or the **ioctl** subroutine. It can be called from the process environment only.

The following parameter block queries statistics of a particular link station (LS):

```
struct dlc_qls_arg
   {
   __ulong32_t  gdlc_sap_corr;   /* GDLC SAP correlator */
   __ulong32_t  gdlc_ls_corr;   /* GDLC ls correlator */
   __ulong32_t  user_sap_corr;   /* user's SAP correlator - RETURNED */
   __ulong32_t  user_ls_corr;   /* user's link station corr-RETURNED */
   u_char_t  ls_diag[DLC_MAX_DIAG]; /* the char name of the ls */
   __ulong32_t  ls_state;   /* current ls state */
   __ulong32_t  ls_sub_state;   /* further clarification of state */
   struct dlc_ls_counters  counters;
   __ulong32_t  protodd_len;   /*protocol dependent data byte length*/
   };
```

The fields of this ioctl operation are:

Field	Description
gdlc_sap_corr	Specifies the generic data link control (GDLC) service access point (SAP) correlator of the target LS.
gdlc_ls_corr	Specifies the GDLC LS correlator to be queried.
user_sap_corr	Specifies the user SAP correlator returned for routing purposes.
user_ls_corr	Specifies the user LS correlator, that is the user LS identifier returned for routing purposes.
ls_diag	Contains the link station (LS) diagnostic tag. Indicates 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.

Field	Description
ls_state	Contains the current state of this LS: **DLC_OPENING** Indicates the SAP or link station is in the process of opening. **DLC_OPENED** Indicates the SAP or link station has been opened. **DLC_CLOSING** Indicates the SAP or link station is the process of closing. **DLC_INACTIVE** Indicates the link station is currently inactive.
ls_sub_state	Contains the current substate of this LS. Several indicators may be active concurrently. **DLC_CALLING** Indicates the link station is calling. **DLC_LISTENING** Indicates the link station is listening. **DLC_CONTACTED** Indicates the link station is contacted into sequenced data mode. **DLC_LOCAL_BUSY** Indicates the local link station is currently busy. **DLC_REMOTE_BUSY** Indicates the remote link station is currently busy.

Field	Description
counters	Contains 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. **test_cmds_sent** Specifies the number of test commands sent. **test_cmds_fail** Specifies the number of test commands failed. **test_cmds_rec** Specifies the number of test commands received. **data_pkt_sent** Specifies the number of sequenced data packets sent. **data_pkt_resent** Specifies the number of sequenced data packets resent. **max_cont_resent** Specifies the maximum number of contiguous resendings. **data_pkt_rec** Indicates data packets received. **inv_pkt_rec** Specifies the number of invalid packets received. **adp_rec_err** Specifies the number of data-detected receive errors. **adp_send_err** Specifies the number of data-detected transmit errors. **rec_inact_to** Specifies the number of received inactivity timeouts. **cmd_polls_sent** Specifies the number of command polls sent. **cmd_repolls_sent** Specifies the number of command repolls sent. **cmd_cont_repolls** Specifies the maximum number of continuous repolls sent.
protodd_len	Indicates length of protocol-dependent data. This field contains the byte length of the following area.

The protocol-dependent data 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 optional data area must directly follow (or append to) the end of the **dlc_qls_arg** structure.

DLC_QUERY_SAP ioctl Operation for DLC

The **DLC_QUERY_SAP** ioctl operation is selectable through the **fp_ioctl** kernel service or the **ioctl** subroutine. It can be called from the process environment only.

The following parameter block 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
   {
   __ulong32_t  gdlc_sap_corr;   /* GDLC SAP correlator */
```

```
   __ulong32_t  user_sap_corr;   /* user SAP correlator (returned) */
   __ulong32_t  sap_state;      /* state of the SAP,returned by kernel*
   uchar_t  dev[DLC_MAX_DIAG]; /* the returned device handler's */
         /* device name */
   __ulong32_t  devdd_len;      /* device driver dependent data */
         /* byte length */
   };
```

The fields of this ioctl operation are:

Field	Description
gdlc_sap_corr	Contains the generic data link control (GDLC) SAP correlator to be queried.
user_sap_corr	Contains the user SAP correlator returned for routing purposes.
sap_state	Contains the current SAP state: **DLC_OPENING** Indicates the SAP or link station is in the process of opening. **DLC_OPENED** Indicates the SAP or link station has been opened. **DLC_CLOSING** Indicates the SAP or link station is the process of closing.
dev	Contains the **/dev** directory name of the communications I/O device handler being used by this SAP.
devdd_len	Contains the byte length of the expected device driver statistics that will be appended to the **dlc_qsap_arg** structure.

The device driver- dependent data contains the device statistics of the attached network device handler. This is generally the query device statistics (reliability/availability/serviceability log area) returned from an ioctl operation issued to the device handler by the Data Link Control (DLC). See the individual GDLC device manager specifications, discussed in the Generic Data Link Control (GDLC) Environment Overview, for information on the particular fields returned.

The optional data area must directly follow or append to the end of the **dlc_qsap_arg** structure.

DLC_START_LS ioctl Operation for DLC

The **DLC_START_LS** ioctl operation is selectable through the **fp_ioctl** kernel service or the **ioctl** subroutine. It can be called from the process environment only.

The following parameter block starts a link station (LS) on a particular SAP as a caller or listener:

```
#define DLC_MAX_DIAG   16   /* the maximum string of chars */
      /* in the diag name */
struct dlc_sls_arg
   {

   __ulong32_t  gdlc_ls_corr;   /* GDLC User link station correlator */
   u_char_t  ls_diag[DLC_MAX_DIAG]; /* the char name of the ls */
   __ulong32_t  gdlc_sap_corr;   /* GDLC SAP correlator */
   __ulong32_t  user_ls_corr;   /* User's SAP correlator */
   __ulong32_t  flags;   /* Start Link Station flags */
   __ulong32_t  trace_chan;   /* Trace Channel (rc of trcstart)*/
   __ulong32_t  len_raddr_name;   /* Length of the remote name/addr*/
   u_char_t  raddr_name[DLC_MAX_NAME]; /* The Remote addr/name */
```

```
__ulong32_t  maxif;   /* Maximum number of bytes in an */
   /* I-field */
__ulong32_t  rcv_wind;   /* Maximum size of receive window */
__ulong32_t  xmit_wind;   /* Maximum size of 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 */

__ulong32_t  max_repoll;   /* Maximum Repoll count */
__ulong32_t  repoll_time;   /* Repoll timeout value */
__ulong32_t  ack_time;   /* Time to delay trans of an ack */
__ulong32_t  inact_time;   /* Time before inactivity times out */
__ulong32_t  force_time;   /* Time before a forced disconnect */
};
```

The fields of this ioctl operation are:

Field	Description
gdlc_ls_corr	Contains 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	Contains LS diagnostic tag. Any ASCII 1 to 16-character name 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	Contains GDLC LS correlator. Specifies the SAP with which to associate this link station. This field must contain the same correlator value passed to the user in the gdlc_sap_corr field by GDLC when the SAP was enabled.
user_ls_corr	Contains user LS correlator. Specifies an identifier or correlator that the user wishes to have returned on all LS results and data from GDLC. It allows the user of multiple link stations to route the station-specific results based on a correlator.

Field	Description
flags	Contains common LS flags. The following flags are supported: **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).

Field	Description
trace_chan	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	Specifies the byte length of the remote address or name. This field must be set to 0 if no remote address or name is required to start the LS. Length values of 0 through 20 are supported.
raddr_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	Specifies the maximum number of I-field bytes that can be in one packet. This value is reduced by GDLC if the device handler 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	The receive window specifies the maximum number of sequentially numbered receive I-frames the local station can accept before sending an acknowledgment.
xmit_wind	Specifies the transmit window and the maximum number of sequentially numbered transmitted I-frames that can be outstanding at any time.
rsap	Specifies the remote SAP address being called. This field is valid only if the **DLC_SLS_LSVC** indicator or the **DLC_SLS_ADDR** indicator is set active.

Field	Description
rsap_low	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	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	Specifies the maximum number of retries for an unacknowledged command frame, or in the case of an I-frame timeout, the number of times the nonresponding remote link station is polled with a supervisory command frame.
repoll_time	Contains the timeout 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	Contains the timeout 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	Contains the timeout value (in increments of 1 second) used to specify the maximum amount of time allowed before receive inactivity returns an error.
force_time	Contains the timeout value (in increments of 1 second) specifying the period to wait for a normal disconnection. Once the timeout occurs, the disconnection is forced and the link station is halted.

The protocol-specific data area allows parameters to be defined by a specific GDLC device manager, such as Token-Ring dynamic window increment or SDLC primary slow poll. This optional data area must directly follow (or append to) the end of the **dlc_sls_arg** structure.

DLC_TEST ioctl Operation for DLC

The **DLC_TEST** ioctl operation is selectable through the **fp_ioctl** kernel service or the **ioctl** subroutine. It can be called from the process environment only.

The following parameter block tests the link to a remote for a particular local link station (LS):

```
struct dlc_corr_arg
   {
   __ulong32_t  gdlc_sap_corr;   /* GDLC SAP correlator */
   __ulong32_t  gdlc_ls_corr;   /* GDLC link station correlator */
   };
```

The fields of this ioctl operation are:

Field	Description
gdlc_sap_corr	Indicates the GDLC SAP correlator of the target LS.
gdlc_ls_corr	Indicates the GDLC LS correlator to be tested.

DLC_TRACE ioctl Operation for DLC

The **DLC_TRACE** ioctl operation is selectable through the **fp_ioctl** kernel service or the **ioctl** subroutine. It can be called from the process environment only.

The following parameter block traces link station (LS) activity for short or long activities:

```
struct dlc_trace_arg
   {
   __ulong32_t  gdlc_sap_corr;   /* GDLC SAP correlator */
   __ulong32_t  gdlc_ls_corr;   /* GDLC link station correlator */
   __ulong32_t  trace_chan;   /* Trace Channel (rc of trcstart) */
   __ulong32_t  flags;   /* Trace Flags */
   };
```

The fields of this ioctl operation are:

Field	Description
gdlc_sap_corr	Contains the GDLC SAP correlator. The correlator returned by GDLC when the SAP was enabled by the user. This correlator identifies the user SAP to the GDLC protocol process.
gdlc_ls_corr	Contains the GDLC LS correlator. The correlator returned by GDLC when the LS was started by the user. This correlator identifies the user LS to the GDLC protocol process.
trace_chan	Specifies the trace channel number obtained from the **trcstart** subroutine. This field is only valid if the **DLC_TRCO** indicator is set active.
flags	Specifies trace flags. The following flags are supported: **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).

IOCINFO ioctl Operation for DLC

This operation returns a structure that describes the device. 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.

The **IOCINFO** ioctl operation is selectable through the **fp_ioctl** kernel service or the **ioctl** subroutine. It can be called from the process environment only.

Related information:

Generic Data Link Control (GDLC) Environment Overview

Data Link Provider Interface (DLPI)

This topic collection includes the subroutines that perform different data link service.

DL_ATTACH_REQ Primitive

Purpose

Requests that the data link service (DLS) provider associate a physical point of attachment (PPA) with a stream.

Structure

The message consists of one **M_PROTO** message block, which contains the following structure:

```
typedef struct
 {
   ulong    dl_primitive;
   ulong    dl_ppa;
} dl_attach_req_t;
```

This structure is defined in **/usr/include/sys/dlpi.h**.

Description

The **DL_ATTACH_REQ** primitive requests that the DLS provider associate a PPA with a stream. The **DL_ATTACH_REQ** primitive is needed for *style 2* DLS providers to identify the physical medium over which communication is to transpire.

Parameters

Item	Description
dl_primitive	Specifies the **DL_ATTACH_REQ** message.
dl_ppa	Specifies the identifier of the PPA to be associated with the stream. The dlpi driver is implemented a *style 2* provider The value of the *dl_ppa* parameter must include identification of the communication medium. For media that multiplex multiple channels over a single physical medium, this identifier should also specify a specific communication channel (where each channel on a physical medium is associated with a separate PPA). **Note:** Because of the provider-specific nature of this value, DLS user software that is to be protocol independent should avoid hard-coding the PPA identifier. The DLS user should retrieve the necessary PPA identifier from some other entity (such as a management entity) and insert it without inspection into the **DL_ATTACH_REQ** primitive.

States

Item	Description
Valid	The primitive is valid in the **DL_UNATTACHED** state.
New	The resulting state is **DL_ATTACH_PENDING**.

Acknowledgments

Item	Description
Successful	The **DL_OK_ACK** primitive is sent to the DLS user resulting in the **DL_UNBOUND** state.
Unsuccessful	The **DL_ERROR_ACK** primitive is returned and the resulting state is unchanged.

Error Codes

Item	Description
DL_ACCESS	Indicates the DLS user does not have proper permission to use the requested PPA.
DL_BADPPA	Indicates the specified PPA is invalid.
DL_OUTSTATE	Indicates the primitive was issued from an invalid state.
DL_SYSERR	Indicates a system error occurred. The system error is indicated in the **DL_ERROR_ACK** primitive.

Related reference:

"DL_BIND_REQ Primitive" on page 53
"DL_OK_ACK Primitive" on page 76
"DL_ERROR_ACK Primitive" on page 70
"DL_INFO_ACK Primitive" on page 73

DL_BIND_ACK Primitive

Purpose

Reports the successful bind of a data link service access point (DLSAP) to a stream.

Structure

The message consists of one **M_PCPROTO** message block, which contains the following structure:

```
typedef struct
{
   ulong  dl_primitive;
   ulong  dl_sap;
   ulong  dl_addr_length;
   ulong  dl_addr_offset;
   ulong  dl_max_conind;
   ulong  dl_xidtest_flg;
} dl_bind_ack_t;
```

This structure is defined in **/usr/include/sys/dlpi.h**.

Description

The **DL_BIND_ACK** primitive reports the successful bind of a DLSAP to a stream and returns the bound DLSAP address to the data link service (DLS) user. This primitive is generated in response to a **DL_BIND_REQ** primitive.

Parameters

Item	Description
dl_primitive	Specifies the **DL_BIND_ACK** primitive.
dl_sap	Specifies the DLSAP address information associated with the bound DLSAP. It corresponds to the *dl_sap* parameter of the associated **DL_BIND_REQ** primitive, which contains part or all of the DLSAP address. For the portion of the DLSAP address conveyed in the **DL_BIND_REQ** primitive, this parameter contains the corresponding portion of the address for the DLSAP that was actually bound.
dl_addr_length	Specifies the length of the complete DLSAP address that was bound to the Data Link Provider Interface (DLPI) stream. The bound DLSAP is chosen according to the guidelines presented under the description of the **DL_BIND_REQ** primitive.
dl_addr_offset	Specifies where the DLSAP address begins. The value of this parameter is the offset from the beginning of the **M_PCPROTO** block.
dl_max_conind	Specifies whether a **DL_CODLS** stream will allow incoming connection idications (**DL_CONNECT_IND**). If the value is zero, the stream cannot accept any **DL_CONNECT_IND** messages; the stream will only accept **DL_CONNECT_REQ**. If the value is greater than zero, then this stream is a listening stream, and indicates how many **DL_CONNECT_IND**'s can be pending at one time.
dl_xidtest_flg	Specifies the XID and test responses supported by the provider. Valid values are: **0** The DLS user will be handling all XID and TEST traffic. **DL_AUTO_XID** Automatically handles XID responses. **DL_AUTO_TEST** Automatically handles test responses. **DL_AUTO_XID | DL_AUTO_TEST** Automatically handles both XID and TEST responses.

States

Item	Description
Valid	The primitive is valid in the **DL_BIND_PENDING** state.
New	The resulting state is **DL_IDLE**.

Related reference:
"DL_BIND_REQ Primitive"

DL_BIND_REQ Primitive

Purpose

Requests that the data link service (DLS) provider bind a data link service access point (DLSAP) to a stream.

Structure

The message consists of one **M_PROTO** message block, which contains the following structure:

```
typedef struct
{
   ulong  dl_primitive;
   ulong  dl_sap;
   ulong  dl_max_conind;
   ushort  dl_service_mode;
   ushort  dl_conn_mgmt;
   ulong  dl_xidtest_flg;
} dl_bind_req_t;
```

This structure is defined in **/usr/include/sys/dlpi.h**.

Description

A stream is active when the DLS provider can transmit and receive protocol data units destined to or originating from the stream. The physical point of attachment (PPA) associated with each stream must be initialized when the **DL_BIND_REQ** primitive has been processed.The PPA is initialized when the **DL_BIND_ACK** primitive is received. If the PPA cannot be initialized, the **DL_BIND_REQ** primitive fails.

Parameters

Item	Description
dl_primitive	Specifies the **DL_BIND_REQ** primitive.

Item	Description
dl_sap	Identifies the DLSAP to be bound to the Data Link Provider Interface (DLPI) stream. This parameter can contain either the full DLSAP address or a portion of the address sufficient to uniquely identify the DLSAP. The **DL_BIND_ACK** primitive returns the full address of the bound DLSAP. The *dl_sap* parameter is a ulong containing and ethertype for DL_ETHER, or a single byte SAP for 802.2 networks. The DLS provider adheres to the following rules when it binds a DLSAP address: • The DLS provider must define and manage its DLSAP address space. • The DLS provider allows the same DLSAP to be bound to multiple streams. The DLS provider may not be able to bind the specified DLSAP address for the following reasons: • The DLS provider statically associated a specific DLSAP with each stream. The value of the *dl_sap* parameter is ignored by the DLS provider and the **DL_BIND_ACK** primitive returns the DLSAP address that is already associated with the stream. **Note:** Because of the provider-specific nature of the DLSAP address, protocol-independent DLS user software should not have this value hard-coded. The DLS user should retrieve the necessary DLSAP address from the appropriate header file for that protocol and insert it without inspection into the **DL_BIND_REQ** primitive.
dl_max_conind	Specifies the maximum number of outstanding **DL_CONNECT_IND** primitives allowed on the DLPI stream. This field controls whether a connection-oriented stream will accept incoming connection indications. This parameter can have one of the following values: **0** The stream cannot accept any **DL_CONNECT_IND** primitives. **>0** The DLS user accepts the specified number of **DL_CONNECT_IND** primitives before having to respond with a **DL_CONNECT_RES** or **DL_DISCONNECT_REQ** primitive. The DLS provider may not be able to support the value supplied in the *dl_max_conind* parameter for the following reasons: • If the provider cannot support the specified number of outstanding connect indications, it should set the value down to a number it can support. • Only one stream that is bound to the indicated DLSAP can have an allowed number of maximum outstanding connect indications greater than 0. If a **DL_BIND_REQ** primitive specifies a value greater than 0, but another stream has already bound itself to the DLSAP with a value greater than 0, the request fails. The DLS provider then sets the *dl_errno* parameter of the **DL_ERROR_ACK** primitive to a value of **DL_BOUND**. • A connection cannot be accepted on a stream bound with a *dl_max_conind* greater than zero. No other streams in which the value of the *dl_max_conind* parameter is greater than 0 can be bound to the same DLSAP. This restriction prevents more than one stream bound to the same DLSAP from receiving connect indications and accepting connections. – A DLS user should always be able to request a *dl_max_conind* parameter value of 0, since this indicates to the DLS provider that the stream will only be used to originate connect requests. – A stream in which the *dl_max_conind* parameter has a negotiated value greater than 0 cannot originate connect requests. **Note:** This field is ignored in connectionless-mode service.

Item	Description
dl_service_mode	Specifies the following modes of service for this stream: **DL_CODLS** Selects the connection-oriented only mode. The connection primitives will be accepted. In addition, an arbitrary number of streams may bind to the same *dl_sap* on the same interface, as long as *dl_max_conind* is zero. No incoming datagram traffic will be sent up this stream. Such frames will either be routed to a **DL_CLDLS** stream, or silently discarded. **DL_CLDLS** Selects the connectionless only mode. The connection primitives will not be accepted. This mode selects exclusive control of connectionless traffic. All datagrams (**DL_UNITDATA_IND**) from any remote station addressed to this *dl_sap* will be received on this stream, even if another stream is currently connected on the same *dl_sap*. Only one stream per interface may bind **DL_CLDS**. **DL_CLDLS | DL_CODLS** Selects the connection-oriented service augmented with conectionless traffic. An arbitrary number of streams may bind to the same *dl_sap* on the same interface. This mode is mutually exclusive with **DL_CLDLS**. If the DLS provider does not support the requested service mode, a **DL_ERROR_ACK** primitive is generated. This primitive conveys a value of **DL_UNSUPPORTED**.
dl_conn_mgmt	This field is ignored.
dl_xidtest_flg	Indicates to the DLS provider that XID or test responses for this stream are to be automatically generated by the DLS provider. The *xidtest_flg* parameter contains a bit mask that can specify either, both, or neither of the following values: **DL_AUTO_XID** Indicates to the DLS provider that automatic responses to XID commands are to be generated. **DL_AUTO_TEST** Indicates to the DLS provider that automatic responses to test commands are to be generated. **DL_AUTO_XID | DL_AUTO_TEST** Indicates to the DLS provider that automatic responses to both XID commands and test commands are to be generated. The DLS provider supports automatic handling of XID and test responses. If an automatic XID or test response has been requested, the DLS provider does not generate **DL_XID_IND** or **DL_TEST_IND** primitives. Therefore, if the provider receives an XID request (**DL_XID_REQ**) or test request (**DL_TEST_REQ**) from the DLS user, the DLS provider returns a **DL_ERROR_ACK** primitive, specifying a **DL_XIDAUTO** or **DL_TESTAUTO** error code, respectively. If no value is specified in the *dl_xidtest_flg* parameter, the DLS provider does not automatically generate XID and test responses. The value informs the DLS provider that the DLS user will be handling all XID and TEST traffic. A nonzero value indicates the DLS provider is responsible for either XID or TEST traffic or both. If the driver handles XID or TEST, the DLS user will not receive any incoming XID or TEST frames, nor be allowed to send them.

States

Item	Description
Valid	The primitive is valid in the **DL_UNBOUND** state.
New	The resulting state is **DL_BIND_PENDING**.

Acknowledgments

Item	Description
Successful	The **DL_BIND_ACK** primitive is sent to the DLS user. The resulting state is **DL_IDLE**.
Unsuccessful	The **DL_ERROR_ACK** primitive is returned. The resulting state is unchanged.

Error Codes

Item	Description
DL_ACCESS	Indicates the DLS user does not have proper permission to use the requested DLSAP address.
DL_BADADDR	Indicates the DLSAP address information is invalid or is in an incorrect format.
DL_BOUND	Indicates the DLS user attempted to bind a second stream to a DLSAP with a *dl_max_conind* parameter value greater than 0, or the DLS user attempted to bind a second connection management stream to the PPA.
DL_INITFAILED	Indicates the automatic initialization of the PPA failed.
DL_NOADDR	Indicates the DLS provider cannot allocate a DLSAP address for this stream.
DL_NOAUTO	Indicates automatic handling of XID and test responses is not supported.
DL_NOTINIT	Indicates the PPA was not initialized prior to this request.
DL_NOTESTAUTO	Indicates automatic handling of test responses is not supported.
DL_NOXIDAUTO	Indicates automatic handling of XID responses is not supported.
DL_OUTSTATE	Indicates the primitive was issued from an invalid state.
DL_SYSERR	Indicates a system error occurred. The system error is indicated in the **DL_ERROR_ACK** primitive.
DL_UNSUPPORTED	Indicates the DLS provider does not support the requested service mode on this stream.

Related reference:

"DL_ATTACH_REQ Primitive" on page 50

"DL_BIND_ACK Primitive" on page 51

"DL_ERROR_ACK Primitive" on page 70

"DL_INFO_ACK Primitive" on page 73

"DL_UNBIND_REQ Primitive" on page 97

DL_CONNECT_CON Primitive

Purpose

Informs the local data link service (DLS) user that the requested data link connection has been established.

Structure

The primitive consists of one **M_PROTO** message block, which contains the following structure:

```
typedef struct
{
   ulong  dl_primitive;
   ulong  dl_resp_addr_length;
   ulong  dl_resp_addr_offset;
   ulong  dl_qos_length;
   ulong  dl_qos_offset;
   ulong  dl_growth;
} dl_connect_con_t;
```

Description

The **DL_CONNECT_CON** primitive informs the local DLS user that the requested data link connection has been established. The primitive contains the data link service access point (DLSAP) address of the responding DLS user.

Note: This primitive applies to connection mode.

Parameters

Item	Description
dl_primitive	Specifies the **DL_CONNECT_CON** primitive.
dl_resp_addr_length	Specifies the length of the address of the responding DLSAP associated with the newly established data link connection.
dl_resp_addr_offset	Specifies where responding DLSAP address begins. The value of this parameter is the offset from the beginning of the **M_PROTO** message block.
dl_qos_length	The DLS provider does not support QOS parameters. This value is set to 0.
dl_qos_offset	The DLS provider does not support QOS parameters. This value is set to 0.
dl_growth	Defines a growth field for future enhancements to this primitive. Its value must be set to zero.

States

Item	Description
Valid	The primitive is valid in the **DL_OUTCON_PENDING** state.
New	The resulting state is **DL_DATAXFER**.

Related reference:
"DL_CONNECT_REQ Primitive" on page 59

DL_CONNECT_IND Primitive
Purpose

Informs the local data link service (DLS) user that a remote (calling) DLS user is attempting to establish a data link connection.

Structure

The primitive consists of one **M_PROTO** message block, which contains the following structure.

```
typedef struct
{
   ulong  dl_primitive;
   ulong  dl_correlation;
   ulong  dl_called_addr_length;
   ulong  dl_called_addr_offset;
   ulong  dl_calling_addr_length;
   ulong  dl_calling_addr_offset;
   ulong  dl_qos_length;
   ulong  dl_qos_offset;
   ulong  dl_growth;
} dl_connect_req_t;
```

Description

The **DL_CONNECT_IND** primitive informs the local DLS user that a remote (calling) DLS user is attempting to establish a data link connection. The primitive contains the data link service access point (DLSAP) addresses of the calling and called DLS user.

The **DL_CONNECT_IND** primitive also contains a number that allows the DLS user to correlate the primitive with a subsequent **DL_CONNECT_RES**, **DL_DISCONNECT_REQ**, or **DL_DISCONNECT_IND** primitive.

The number of outstanding **DL_CONNECT_IND** primitives issued by the DLS provider must not exceed the value of *the dl_max_conind* parameter specified by the **DL_BIND_ACK** primitive. If this limit is reached and an additional connect request arrives, the DLS provider does not pass the corresponding connect indication to the DLS user until a response is received for an outstanding request.

Note: This primitive applies to connection mode.

Parameters

Item	Description
dl_primitive	Specifies the **DL_CONNECT_IND** primitive.
dl_correlation	Specifies the correlation number to be used by the DLS user to associate this message with the **DL_CONNECT_RES**, **DL_DISCONNECT_REQ**, or **DL_DISCONNECT_IND** primitive that is to follow. This value enables the DLS user to multithread connect indications and responses. All outstanding connect indications must have a distinct, nonzero correlation value set by the DLS provider.
dl_called_addr_length	Specifies the length of the address of the DLSAP for which this **DL_CONNECT_IND** primitive is intended. This address is the full DLSAP address specified by the calling DLS user and is typically the value returned on the **DL_BIND_ACK** associated with the given stream.
dl_called_addr_offset	Specifies where the called DLSAP address begins. The value of this parameter is the offset from the beginning of the **M_PROTO** message block.
dl_calling_addr_length	Specifies the length of the address of the DLSAP from which the **DL_CONNECT_REQ** primitive was sent.
dl_calling_addr_offset	Specifies where the calling DLSAP address begins. The value of this parameter is the offset from the beginning of the **M_PROTO** message block.
dl_qos_length	The DLS provider does not support QOS parameters. This length field is set to 0.
dl_qos_offset	The DLS provider does not support QOS parameters. This length field is set to 0.
dl_growth	Defines a growth field for future enhancements to this primitive. Its value must be set to 0.

States

Item	Description
Valid	The primitive is valid in the **DL_IDLE** state. It is also valid in the **DL_INCON_PENDING** state when the maximum number of outstanding **DL_CONNECT_IND** primitives has not been reached on this stream.
New	The resulting state is **DL_INCON_PENDING**, regardless of the current state.

Acknowledgments

The DLS user must send either the **DL_CONNECT_RES** primitive to accept the connect request or the **DL_DISCONNECT_REQ** primitive to reject the connect request. In either case, the responding message must convey the correlation number received from the **DL_CONNECT_IND** primitive. The DLS provider uses the correlation number to identify the connect request to which the DLS user is responding.

Related reference:

"DL_BIND_ACK Primitive" on page 51

"DL_CONNECT_RES Primitive" on page 60
"DL_DISCONNECT_IND Primitive" on page 66
"DL_DISCONNECT_REQ Primitive" on page 67

DL_CONNECT_REQ Primitive

Purpose

Requests that the data link service (DLS) provider establish a data link connection with a remote DLS user.

Structure

The primitive consists of one **M_PROTO** message block, which contains the following structure:

```
typedef struct
{
   ulong  dl_primitive;
   ulong  dl_dest_addr_length;
   ulong  dl_dest_addr_offset;
   ulong  dl_qos_length;
   ulong  dl_qos_offset;
   ulong  dl_growth;
} dl_connect_req_t;
```

Description

The **DL_CONNECT_REQ** primitive requests that the DLS provider establish a data link connection with a remote DLS user. The request contains the data link service access point (DLSAP) address of the remote DLS user.

Note: This primitive applies to connection mode.

Parameters

Item	Description
dl_primitive	Specifies the **DL_CONNECT_REQ** primitive.
dl_dest_addr_length	Specifies the length of the DLSAP address that identifies the DLS user with whom a connection is to be established. If the called user is implemented using DLPI, this address is the full DLSAP address returned on the **DL_BIND_ACK** primitive.
dl_dest_addr_offset	Specifies where the destination DLSAP address begins. The value of this parameter is the offset from the beginning of the **M_PROTO** message block.
dl_qos_length	The DLS provider does not support any QOS parameter values. This value is set to 0.
dl_qos_offset	The DLS provider does not support any QOS parameter values. This value is set to 0.
dl_growth	Defines a growth field for future enhancements to this primitive. Its value must be set to 0.

States

Item	Description
Valid	The primitive is valid in the **DL_IDLE** state.
New	The resulting state is **DL_OUTCON_PENDING**.

Acknowledgments

There is no immediate response to the connect request. However, if the connect request is accepted by the called DLS user, the **DL_CONNECT_CON** primitive is sent to the calling DLS user, resulting in the **DL_DATAXFER** state.

If the connect request is rejected by the called DLS user, the called DLS user cannot be reached, or the DLS provider or called DLS user do not agree on the specified quality of service, a **DL_DISCONNECT_IND** primitive is sent to the calling DLS user, resulting in the **DL_IDLE** state.

If the request is erroneous, the **DL_ERROR_ACK** primitive is returned and the resulting state is unchanged.

Error Codes

Item	Description
DL_ACCESS	Indicates the DLS user does not have proper permission to use the requested DLSAP address.
DL_BADADDR	Indicates the DLSAP address information is invalid or is in an incorrect format.
DL_BADQOSPARAM	Indicates the QOS parameters contain invalid values.
DL_BADQOSTYPE	Indicates the QOS structure type is not supported by the DLS provider.
DL_OUTSTATE	Indicates the primitive was issued from an invalid state.
DL_SYSERR	Indicates a system error occurred. The system error is indicated in the **DL_ERROR_ACK** primitive.
DL_UNSUPPORTED	Indicates the DLS user has indicated QOS parameters, which are unsupported.

Related reference:

"DL_CONNECT_CON Primitive" on page 56

"DL_DISCONNECT_IND Primitive" on page 66

"DL_ERROR_ACK Primitive" on page 70

"DL_BIND_ACK Primitive" on page 51

DL_CONNECT_RES Primitive

Purpose

Directs the data link service (DLS) provider to accept a connect request from a remote DLS user.

Structure

The primitive consists of one **M_PROTO** message block, which contains the following structure:

```
typedef struct
{
   ulong  dl_primitive;
   ulong  dl_correlation;
   ulong  dl_resp_token;
   ulong  dl_qos_length;
   ulong  dl_qos_offset;
   ulong  dl_growth;
} dl_connect_res_t;
```

Description

The **DL_CONNECT_RES** primitive directs the DLS provider to accept a connect request from a remote (calling) DLS user on a designated stream. The DLS user can accept the connection on the same stream where the connect indication arrived, or on a different, previously bound stream. The response contains the correlation number from the corresponding **DL_CONNECT_IND** primitive, selected quality of service (QOS) parameters, and an indication of the stream on which to accept the connection.

After issuing this primitive, the DLS user can immediately begin transferring data using the **DL_DATA_REQ** primitive. However, if the DLS provider receives one or more **DL_DATA_REQ** primitives from the local DLS user before it has established a connection, the provider must queue the data transfer requests internally until the connection is successfully established.

Note: This primitive applies to connection mode.

Parameters

Item	Description
dl_primitive	Specifies the **DL_CONNECT_RES** primitive.
dl_correlation	Specifies the correlation number that was received with the corresponding **DL_CONNECT_IND** primitive. The DLS provider uses the correlation number to identify the connect indication to which the DLS user is responding.
dl_resp_token	Specifies one of the following values: **>0** Specifies the token associated with the responding stream on which the DLS provider is to establish the connection. This stream must be in the **DL_IDLE** state. The token value for a stream can be obtained by issuing a **DL_TOKEN_REQ** primitive on that stream. **0** Indicates the DLS user is accepting the connection on the stream where the connect indication arrived.
dl_qos_length	The DLS provider does not support QOS parameters. This value is set to 0.
dl_qos_offset	The DLS provider does not support QOS parameters. This value is set to 0.
dl_growth	Defines a growth field for future enhancements to this primitive. Its value must be set to 0.

States

Item	Description
Valid	The primitive is valid in the **DL_INCON_PENDING** state.
New	The resulting state is **DL_CONN_RES_PENDING**.

Acknowledgments

Item	Description
Successful	The **DL_OK_ACK** primitive is sent to the DLS user. If no outstanding connect indications remain, the resulting state for the current stream is **DL_IDLE**. Otherwise, it remains **DL_INCON_PENDING**. For the responding stream (designated by the *dl_resp_token* parameter), the resulting state is **DL_DATAXFER**. If the current stream and responding stream are the same, the resulting state of that stream is **DL_DATAXFER**. These streams can only be the same when the response corresponds to the only outstanding connect indication.
Unsuccessful	The **DL_ERROR_ACK** primitive is returned on the stream where the **DL_CONNECT_RES** primitive was received, and the resulting state of that stream and the responding stream is unchanged.

Error Codes

Item	Description
DL_ACCESS	Indicates the DLS user does not have proper permission to use the requested data link service access point (DLSAP) address.
DL_BADCORR	Indicates the correlation number specified in this primitive does not correspond to a pending connect indication.
DL_BADQOSPARAM	Indicates the QOS parameters contain invalid values.
DL_BADQOSTYPE	Indicates the QOS structure type is not supported by the DLS provider.
DL_BADTOKEN	Indicates the token for the responding stream is not associated with a currently open stream.
DL_OUTSTATE	Indicates the primitive was issued from an invalid state, or the responding stream was not in a valid state for establishing a connection.
DL_PENDING	Indicates the current and responding streams are the same, and there is more than one outstanding connect indication.
DL_SYSERR	Indicates a system error occurred. The system error is indicated in the **DL_ERROR_ACK** primitive.

Related reference:
"DL_CONNECT_IND Primitive" on page 57
"DL_DATA_REQ Primitive" on page 63
"DL_ERROR_ACK Primitive" on page 70
"DL_OK_ACK Primitive" on page 76
"DL_TOKEN_REQ Primitive" on page 95

DL_DATA_IND Primitive

Purpose

Conveys a data link service data unit (DLSDU) from the data link service (DLS) provider to the DLS user.

Structure

The primitive consists of one or more **M_DATA** message blocks containing at least one byte of data. (That is, there is no DLPI data structure associated with this primitive.)

Description

The **DL_DATA_IND** primitive conveys a DLSDU from the DLS provider to the DLS user. The DLS provider guarantees to deliver each DLSDU to the local DLS user in the same order as received from the remote DLS user. If the DLS provider detects unrecoverable data loss during data transfer, this may be indicated to the DLS user by a **DL_RESET_IND** primitive, or, if the connection is lost, by a **DL_DISCONNECT_IND** primitive.

Note: This primitive applies to connection mode.

States

Item	Description
Valid	The primitive is valid in the **DL_DATAXFER** state.
New	The resulting state is unchanged.

Related reference:
"DL_DISCONNECT_IND Primitive" on page 66
"DL_RESET_IND Primitive" on page 83

DL_DATA_REQ Primitive

Purpose

Conveys a complete data link service data unit (DLSDU) from the data link service (DLS) user to the DLS provider for transmission over the data link connection.

Structure

This primitive consists of one or more **M_DATA** message blocks containing at least one byte of data. (That is, there is no DLPI data structure associated with this primitive.)

Description

The **DL_DATA_REQ** primitive conveys a complete DLSDU from the DLS user to the DLS provider for transmission over the data link connection. The DLS provider guarantees to deliver each DLSDU to the remote DLS user in the same order as received from the local DLS user. If the DLS provider detects unrecoverable data loss during data transfer, the DLS user can be notified by a **DL_RESET_IND** primitive. If the connection is lost, the user can be notified by a **DL_DISCONNECT_IND** primitive.

To simplify support of a **read**/**write** interface to the data link layer, the DLS provider must recognize and process messages that consist of one or more **M_DATA** message blocks without a preceding **M_PROTO** message block. This message type may originate from the **write** subroutine.

Note:

1. This does not imply that the Data Link Provider Interface (DLPI) directly supports a pure **read**/**write** interface. If such an interface is desired, a streams module could be implemented to be pushed above the DLS provider.
2. (Support of Direct User-Level Access) A streams module would implement more field processing itself to support direct user-level access. This module could collect messages and send them in one larger message to the DLS provider, or break large DLSDUs passed to the DLS user into smaller messages. The module would only be pushed if the DLS user was a user-level process.
3. The **DL_DATA_REQ** primitive applies to connection mode.

States

Item	Description
Valid	The primitive is valid in the **DL_DATAXFER** state. If it is received in the **DL_IDLE** or **DL_PROV_RESET_PENDING** state, the primitive is discarded without generating an error.
New	The resulting state is unchanged.

Acknowledgments

Item	Description
Successful	No response is generated.
Unsuccessful	A streams **M_ERROR** message is issued to the DLS user specifying an **errno** global value of **EPROTO**. This action should be interpreted as a fatal, unrecoverable, protocol error. A request will fail under the following conditions: • The primitive was issued from an invalid state. If the request is issued in the **DL_IDLE** or **DL_PROV_RESET_PENDING** state. However, the request is discarded without generating an error. • The amount of data in the current DLSDU is not within the DLS provider's acceptable bounds as specified by the *dl_min_sdu* and *dl_max_sdu* parameters of the **DL_INFO_ACK** primitive.

Related reference:

"DL_CONNECT_RES Primitive" on page 60

"DL_DISCONNECT_IND Primitive" on page 66

"DL_INFO_ACK Primitive" on page 73
"DL_RESET_IND Primitive" on page 83

DL_DETACH_REQ Primitive

Purpose

Requests that the data link service (DLS) *style 2* provider detach a physical point of attachment (PPA) from a stream.

Structure

The message consists of one **M_PROTO** message block, which contains the following structure:

```
typedef struct
{
   ulong  dl_primitive;
} dl_detach_req_t;
```

This structure is defined in **/usr/include/sys/dlpi.h**.

Description

For *style 2* DLS providers, the **DL_DETACH_REQ** primitive requests the DLS provider detach a PPA from a stream.

Parameters

Item	Description
dl_primitive	Specifies the **DL_DETACH_REQ** primitive.

States

Item	Description
Valid	The primitive is valid in the **DL_UNBOUND** state.
New	The resulting state is **DL_DETACH_PENDING**.

Acknowledgments

Item	Description
Successful	The **DL_OK_ACK** primitive is sent to the DLS user. The resulting state is **DL_UNATTACHED**.
Unsuccessful	The **DL_ERROR_ACK** primitive is returned, and the resulting state is unchanged.

Error Codes

Item	Description
DL_OUTSTATE	Indicates the primitive was issued from an invalid state.
DL_SYSERR	Indicates a system error occurred. The system error is indicated in the **DL_ERROR_ACK** primitive.

Related reference:

"DL_ERROR_ACK Primitive" on page 70

"DL_OK_ACK Primitive" on page 76

"DL_PROMISCON_REQ Primitive" on page 81

DL_DISABMULTI_REQ Primitive

Purpose

Requests that the data link service (DLS) provider disable specific multicast addresses on a per stream basis.

Structure

The message consists of one **M_PROTO** message block, which contains the following structure:

```
typedef struct
{
   ulong  dl_primitive;
   ulong  dl_addr_length;
   ulong  dl_addr_offset;
} dl_disabmulti_req_t;
```

This structure is defined in **/usr/include/sys/dlpi.h**.

Description

The **DL_DISABMULTI_REQ** primitive requests that the DLS provider disable specific multicast addresses on a per stream basis.

The DLS provider must not run in the interrupt environment. If the DLS provider runs in the interrupt environment, the system returns a **DL_ERROR_ACK** primitive with an error code of **DL_SYSERR** and an operating system error code of 0.

Parameters

Item	Description
dl_primitive	Specifies the **DL_DISABMULTI_REQ** primitive.
dl_addr_length	Specifies the length of the physical address.
dl_addr_offset	Indicates where the multicast address begins. The value of this parameter is the offset from the beginning of the **M_PROTO** message block.

States

Item	Description
Valid	The primitive is valid in any state in which a local acknowledgement is not pending, with the exception of the **DL_UNATTACH** state.
New	The resulting state is unchanged.

Acknowledgments

Item	Description
Successful	The **DL_OK_ACK** primitive is sent to the DLS user.
Unsuccessful	The **DL_ERROR_ACK** primitive is returned, and the resulting state is unchanged.

Error Codes

Item	Description
DL_BADADDR	Indicates the data link service access point (DLSAP) address information is invalid or is in an incorrect format.
DL_NOTENAB	Indicates the address specified is not enabled.
DL_NOTSUPPORTED	Indicates the primitive is known but not supported by the DLS provider.
DL_OUTSTATE	Indicates the primitive was issued from an invalid state.
DL_SYSERR	Indicates a system error occurred. The **DL_ERROR_ACK** primitive indicates the system error.

Related reference:

"DL_OK_ACK Primitive" on page 76

"DL_ERROR_ACK Primitive" on page 70

"DL_ENABMULTI_REQ Primitive" on page 69

DL_DISCONNECT_IND Primitive

Purpose

Informs the data link service (DLS) user that the data link connection on the current stream has been disconnected, or that a pending connection has been cancelled.

Structure

The primitive consists of one **M_PROTO** message block, which contains the following structure:

```
typedef struct
{
   ulong  dl_primitive;
   ulong  dl_originator;
   ulong  dl_reason;
   ulong  dl_correlation;
} dl_disconnect_ind_t;
```

Description

The **DL_DISCONNECT_IND** primitive informs the DLS user of one of the following conditions:

- The data link connection on the current stream has been disconnected.
- A pending connection from either the **DL_CONNECT_REQ** or **DL_CONNECT_IND** primitive has been cancelled.

The primitive indicates the origin and the cause of the disconnect.

Note: This primitive applies to connection mode.

Parameters

Item	Description
dl_primitive	Specifies the **DL_DISCONNECT_IND** primitive.
dl_originator	Indicates whether the disconnect originated from a DLS user or provider. Valid values are **DL_USER** and **DL_PROVIDER**.
dl_reason	Specifies the reason for the disconnect. Reasons for disconnect are: **DL_DISC_PERMANENT_CONDITION** Indicates the connection was released because of a permanent condition. **DL_DISC_TRANSIENT_CONDITION** Indicates the connection was released because of a temporary condition. **DL_CONREJ_DEST_UNKNOWN** Indicates the connect request has an unknown destination. **DL_CONREJ_DEST_UNREACH_PERMANENT** Indicates the connection was released because the destination for connect request could not be reached. This is a permanent condition. **DL_CONREJ_DEST_UNREACH_TRANSIENT** Indicates the connection was released because the destination for connect request could not be reached. This is a temporary condition. **DL_CONREJ_QOS_UNAVAIL_PERMANENT** Indicates the requested quality of service (QOS) parameters became permanently unavailable while establishing a connection. **DL_CONREJ_QOS_UNAVAIL_TRANSIENT** Indicates the requested QOS parameters became temporarily unavailable while establishing a connection. **DL_DISC_UNSPECIFIED** Indicates the connection was closed because of an unspecified reason.
dl_correlation	If the value is nonzero, specifies the correlation number contained in the **DL_CONNECT_IND** primitive being cancelled.This value permits the DLS user to associate the message with the proper **DL_CONNECT_IND** primitive. If the disconnect request indicates the release of a connection that is already established, or is indicating the rejection of a previously sent **DL_CONNECT_REQ** primitive, the value of the *dl_correlation* parameter is zero.

States

Item	Description
Valid	The primitive is valid in any of the following states: • **DL_DATAXFER** • **DL_INCON_PENDING** • **DL_OUTCON_PENDING** • **DL_PROV_RESET_PENDING** • **DL_USER_RESET_PENDING**
New	The resulting state is **DL_IDLE**.

Related reference:

"DL_CONNECT_IND Primitive" on page 57
"DL_CONNECT_REQ Primitive" on page 59
"DL_DATA_IND Primitive" on page 62
"DL_DATA_REQ Primitive" on page 63

DL_DISCONNECT_REQ Primitive

Purpose

Requests that an active data link be disconnected.

Structure

The primitive consists of one **M_PROTO** message block, which contains the following structure:

```
typedef struct
{
   ulong  dl_primitive;
   ulong  dl_reason;
   ulong  dl_correlation;
} dl_disconnect_req_t;
```

Description

The **DL_DISCONNECT_REQ** primitive requests the data link service (DLS) provider to disconnect an active data link connection or one that was in the process of activation. The **DL_DISCONNECT_REQ** primitive can be sent in response to a previously issued **DL_CONNECT_IND** or **DL_CONNECT_REQ** primitive. If an incoming **DL_CONNECT_IND** primitive is being refused, the correlation number associated with that connect indication must be supplied. The message indicates the reason for the disconnect.

Note: This primitive applies to connection mode.

Parameters

Item	Description
dl_primitive	Specifies the **DL_DISCONNECT_REQ** primitive.
dl_reason	Indicates one of the following reasons for the disconnect: **DL_DISC_NORMAL_CONDITION** Indicates normal release of a data link connection. **DL_DISC_ABNORMAL_CONDITION** Indicates abnormal release of a data link connection. **DL_CONREJ_PERMANENT_COND** Indicates a permanent condition caused the rejection of a connect request. **DL_CONREJ_TRANSIENT_COND** Indicates a transient condition caused the rejection of a connect request. **DL_DISC_UNSPECIFIED** Indicates the connection was closed for an unspecified reason.
dl_correlation	Specifies one of the following values: **0** Indicates either the disconnect request is releasing an established connection or is cancelling a previously sent **DL_CONNECT_REQ** primitive. **>0** Specifies the correlation number that was contained in the **DL_CONNECT_IND** primitive being rejected. This value permits the DLS provider to associate the primitive with the proper **DL_CONNECT_IND** primitive when rejecting an incoming connection.

States

Item	Description
Valid	The primitive is valid in any of the following states: • **DL_DATAXFER** • **DL_INCON_PENDING** • **DL_OUTCON_PENDING** • **DL_PROV_RESET_PENDING** • **DL_USER_RESET_PENDING**
New	**DL_DISCON11_PENDING**

Acknowledgments

Item	Description
Successful	The **DL_OK_ACK** primitive is sent to the DLS user resulting in the **DL_IDLE** state.
Unsuccessful	The **DL_ERROR_ACK** primitive is returned, and the resulting state is unchanged.

Error Codes

Item	Description
DL_BADCORR	Indicates the correlation number specified in this primitive does not correspond to a pending connect indication.
DL_OUTSTATE	Indicates the primitive was issued from an invalid state.
DL_SYSERR	Indicates a system error occurred. The system error is indicated in the **DL_ERROR_ACK** primitive.

Related reference:

"DL_CONNECT_IND Primitive" on page 57

"DL_OK_ACK Primitive" on page 76

"DL_ERROR_ACK Primitive" on page 70

DL_ENABMULTI_REQ Primitive

Purpose

Requests that the data link service (DLS) provider enable specific multicast addresses on a per stream basis.

Structure

The primitive consists of one **M_PROTO** message block, which contains the following structure:

```
typedef struct
{
   ulong  dl_primitive;
   ulong  dl_addr_length;
   ulong  dl_addr_offset;
} dl_enabmulti_req_t;
```

This structure is defined in **/usr/include/sys/dlpi.h**.

Description

The **DL_ENABMULTI** primitive requests that the DLS provider enable specific multicast addresses on a per stream basis. It is invalid for a DLS provider to pass upstream messages that are destined for any address other than those explicitly enabled on that stream by the DLS user.

If a duplicate address is requested, the system returns a **DL_OK_ACK** primitive, with no operation performed. If the stream is closed, all multicast addresses associated with the stream will be unregistered.

The DLS provider must not run in the interrupt environment. If the DLS provider runs in the interrupt environment, the system returns a **DL_ERROR_ACK** primitive with a **DL_SYSERR** error code and an operating system error code of 0.

Parameters

Item	Description
dl_primitive	Specifies the **DL_ENABMULTI** primitive.
dl_addr_length	Specifies the length of the multicast address.
dl_addr_offset	Indicates where the multicast address begins. The value of this parameter is the offset from the beginning of the **M_PROTO** message block.

States

Item	Description
Valid	The primitive is valid in any state in which a local acknowledgement is not pending, with the exception of the **DL_UNATTACH** state.
New	The resulting state is unchanged.

Acknowledgments

Item	Description
Successful	The **DL_OK_ACK** primitive is sent to the DLS user.
Unsuccessful	The **DL_ERROR_ACK** primitive is returned, and the resulting state is unchanged.

Error Codes

Item	Description
DL_BADADDR	Indicates the data link service access point (DLSAP) address information is invalid or is in an incorrect format.
DL_NOTSUPPORTED	Indicates the primitive is known but not supported by the DLS provider.
DL_OUTSTATE	Indicates the primitive was issued from an invalid state, or the responding stream was not in a valid state for establishing a connection.
DL_TOOMANY	Indicates the limit has been exceeded for the maximum number of DLSAPs per stream.
DL_SYSERR	Indicates a system error. The **DL_ERROR_ACK** primitive indicates the error.

Related reference:

"DL_DISABMULTI_REQ Primitive" on page 65

"DL_OK_ACK Primitive" on page 76

"DL_ERROR_ACK Primitive"

DL_ERROR_ACK Primitive

Purpose

Informs the data link service (DLS) user that a request or response was invalid.

Structure

The message consists of one **M_PCPROTO** message block, which contains the following structure:

```
typedef struct
{
```

```
   ulong  dl_primitive;
   ulong  dl_error_primitive;
   ulong  dl_errno;
   ulong  dl_unix_errno;
} dl_ok_ack_t;
```

This structure is defined in **/usr/include/sys/dlpi.h**.

Description

The **DL_ERROR_ACK** primitive informs the DLS user that the previously issued request or response was invalid. This primitive identifies the primitive in error, specifies a Data Link Provider Interface (DLPI) error code, and if appropriate, indicates an operating system error code.

Parameters

Item	Description
dl_primitive	Specifies the **DL_ERROR_ACK** primitive.
dl_error_primitive	Identifies the primitive that caused the error.
dl_errno	Specifies the DLPI error code associated with the failure. See the individual request or response for the error codes that are applicable. In addition to those errors: **DL_BADPRIM** Indicates an unrecognized primitive was issued by the DLS user. **DL_NOTSUPPORTED** Indicates an unsupported primitive was issued by the DLS user.
dl_unix_errno	Specifies the operating system error code associated with the failure. This value should be nonzero only when the *dl_errno* parameter is set to **DL_SYSERR**. It is used to report operating system failures that prevent the processing of a given request or response.

States

Item	Description
Valid	The primitive is valid in all states that have a pending acknowledgment or confirmation.
New	The resulting state is the same as the one from which the acknowledged request or response was generated.

DL_GET_STATISTICS_ACK Primitive

Purpose

Returns statistics in response to the **DL_GET_STATISTICS_REQ** primitive.

Structure

The message consists of one **M_PCPROTO** message block, which contains the following structure:

```
typedef struct
{
   ulong  dl_primitive;
   ulong  dl_stat_length;
   ulong  dl_stat_offset;
} dl_get_statistics_ack_t;
```

This structure is defined in **/usr/include/sys/dlpi.h**.

Description

The **DL_GET_STATISTICS_ACK** primitive returns statistics in response to the **DL_GET_STATISTICS_REQ** primitive.

The **/usr/include/sys/dlpistats.h** file defines the statistics that the **DL_GET_STATISTICS_ACK** and **DL_GET_STATISTICS_REQ** primitives support. The primitives support the statistics both globally (totals for all streams) and per stream. Per stream, or *local*, statistics can be requested only for the stream over which the **DL_GET_STATISTICS_REQ** primitive is requested.

The global and local statistics structures are returned concatenated. The offset in the **M_PCPROTO** message, returned by the **DL_GET_STATISTICS_ACK** primitive, indicates where the two concatenated structures begin. The first statistics structure contains information about the local stream over which the **DL_GET_STATISTICS_REQ** primitive was issued. The second statistics structure contains the global statistics collected and summed for all streams.

The structures for the local statistics are initialized to zero when the stream is opened. The structure for the global statistics is initialized to zero when the **dlpi** kernel extension is loaded. The statistics structures can be reset to zero using the **DL_ZERO_STATS IOCTL** command. See "IOCTL Specifics" in Data Link Provider Interface Information.

The statistics collected by the DLPI provider are considered vague. There are no locks protecting the counters to prevent write collisions.

Parameters

Item	Description
dl_primitive	Specifies the **DL_GET_STATISTICS_ACK** primitive.
dl_stat_length	Specifies the length of the statistics structure.
dl_stat_offset	Indicates where the statistics information begins. The value of this parameter is the offset from the beginning of the **M_PCPROTO** block.

States

Item	Description
Valid	The primitive is valid in any attached state in which a local acknowledgement is not pending.
New	The resulting state is unchanged.

Related reference:

"DL_BIND_ACK Primitive" on page 51

Related information:

Data Link Provider Interface Information

DL_GET_STATISTICS_REQ

Purpose

Directs the data link service (DLS) provider to return statistics to the DLS user.

Structure

The message consists of one **M_PROTO** message block, which contains the following structure:

```
typedef struct
{
   ulong  dl_primitive;
} dl_get_statistics_req_t;
```

The **dl_get_statistics_req_t** structure is defined in **/usr/include/sys/dlpi.h**.

Description

The **DL_GET_STATISTICS_REQ** primitive directs the DLS provider to return statistics.

Parameters

Item	Description
dl_primitive	Specifies the **DL_GET_STATISTICS_REQ** primitive.

States

Item	Description
Valid	The primitive is valid in any attached state in which a local acknowledgment is not pending.
New	The resulting state is unchanged.

Acknowledgments

Item	Description
Successful	The **DL_GET_STATISTICS_ACK** primitive is sent to the DLS user.
Unsuccessful	The **DL_ERROR_ACK** primitive is returned to the DLS user.

Error Codes

Item	Description
DL_NOTSUPPORTED	Indicates the primitive is known but not supported by the DLS provider.
DL_SYSERR	Indicates a system error. The **DL_ERROR_ACK** primitive indicates the error.

Related reference:

"DL_BIND_ACK Primitive" on page 51

"DL_ERROR_ACK Primitive" on page 70

DL_INFO_ACK Primitive

Purpose

Returns information about the Data Link Provider Interface (DLPI) stream in response to the **DL_INFO_REQ** primitive.

Structure

The message consists of one **M_PCPROTO** message block, which contains the following structure:

```
typedef struct
{
   ulong  dl_primitive;
   ulong  dl_max_sdu;
   ulong  dl_min_sdu;
   ulong  dl_addr_length;
```

```
   ulong  dl_mac_type;
   ulong  dl_reserved;
   ulong  dl_current_state;
   long  dl_sap_length;
   ulong  dl_service_mode;
   ulong  dl_qos_length;
   ulong  dl_qos_offset;
   ulong  dl_qos_range_length;
   ulong  dl_qos_range_offset;
   ulong  dl_provider_style;
   ulong  dl_addr_offset;
   ulong  dl_version;
   ulong  dl_brdcst_addr_length;
   ulong  dl_brdcst_addr_offset;
   ulong  dl_growth;
} dl_info_ack_t;
```

This structure is defined in **/usr/include/sys/dlpi.h**.

Description

The **DL_INFO_ACK** primitive returns information about the DLPI stream to the data link service (DLS). The **DL_INFO_ACK** primitive is a response to the **DL_INFO_REQ** primitive.

Parameters

Item	Description
dl_primitive	Specifies the **DL_INFO_ACK** primitive.
dl_max_sdu	Specifies the maximum number of bytes that can be transmitted in a data link service data unit (DLSDU). This value must be a positive integer greater than or equal to the value of the *dl_min_sdu* parameter.
dl_min_sdu	Specifies the minimum number of bytes that can be transmitted in a DLSDU. The minimum value is 1.
dl_addr_length	Specifies the length, in bytes, of the provider's data link service access point (DLSAP) address. For hierarchical subsequent binds, the length returned is the total length. The total length is the sum of the values for the physical address, service access point (SAP), and subsequent address length.
dl_mac_type	Specifies the type of medium supported by this DLPI stream. Possible values include: **DL_CSMACD** Indicates the medium is carrier sense multiple access with collision detection (ISO 8802/3). **DL_TPR** Indicates the medium is token-passing ring (ISO 8802/5). **DL_ETHER** Indicates the medium is Ethernet bus. **DL_FDDI** Indicates the medium is a Fiber Distributed Data Interface. **DL_OTHER** Indicates any other medium.
dl_reserved	Indicates a reserved field, the value of which must be set to 0.
dl_current_state	Specifies the state of the DLPI interface for the stream the DLS provider issues this acknowledgement.

Item	Description
dl_sap_length	Indicates the current length of the SAP component of the DLSAP address. The specified value must be an integer. The absolute value of the *dl_sap_length* parameter provides the length of the SAP component within the DLSAP address. The value can be one of the following: **>0** Indicates the SAP component precedes the physical component within the DLSAP address. **<0** Indicates the physical component precedes the SAP component within the DLSAP address. **0** Indicates that no SAP has been bound.
dl_service_mode	Specifies which service modes that the DLS provider supports if the **DL_INFO_ACK** primitive is returned before the **DL_BIND_REQ** primitive is processed. This parameter contains a bit-mask specifying the following value: **DL_CODLS** Indicates connection-oriented DLS. **DL_CLDLS** Indicates connectionless DLS. Once a specific service mode has been bound to the stream, this field returns that specific service mode.
dl_qos_length	The DLS provider does not support *_qos_* parameters. This value is set to 0.
dl_qos_offset	The DLS provider does not support *_qos_* parameters. This value is set to 0.
dl_qos_range_length	The DLS provider does not support *_qos_* parameters. This value is set to 0.
dl_qos_range_offset	The DLS provider does not support *_qos_* parameters. This value is set to 0.
dl_provider_style	Specifies the style of the DLS provider associated with the DLPI stream. The following provider class is defined: **DL_STYLE2** Indicates the DLS user must explicitly attach a PPA to the DLPI stream using the **DL_ATTACH_REQ** primitive.
dl_addr_offset	Specifies the offset of the address that is bound to the associated stream. If the DLS user issues a **DL_INFO_REQ** primitive before binding a DLSAP, the value of the *dl_addr_length* parameter is set to 0.
dl_version	Indicates the version of the supported DLPI.
dl_brdcst_addr_length	Indicates the length of the physical broadcast address.
dl_brdcst_addr_offset	Indicates where the physical broadcast address begins. The value of this parameter is the offset from the beginning of the **PCPROTO** block.
dl_growth	Specifies a growth field for future use. The value of this parameter is 0.

States

Item	Description
Valid	The primitive is valid in any state in response to a **DL_INFO_REQ** primitive.
New	The resulting state is unchanged.

Related reference:

"DL_DATA_REQ Primitive" on page 63
"DL_INFO_REQ Primitive"
"DL_BIND_REQ Primitive" on page 53
"DL_ATTACH_REQ Primitive" on page 50
"DL_UNITDATA_IND Primitive" on page 98
"DL_UNITDATA_REQ Primitive" on page 99

DL_INFO_REQ Primitive

Purpose

Requests information about the Data Link Provider Interface (DLPI) stream.

Structure

The message consists of one **M_PCPROTO** message block, which contains the following structure:

```
typedef struct
{
   ulong dl_primitive;
} dl_info_req_t;
```

This structure is defined in **/usr/include/sys/dlpi.h**.

Description

The **DL_INFO_REQ** primitive requests information from the data link service (DLS) provider about the DLPI stream. This information includes a set of provider-specific parameters, as well as the current state of the interface.

Parameters

Item	Description
dl_primitive	Conveys the **DL_INFO_REQ** primitive.

States

Item	Description
Valid	The primitive is valid in any state in which a local acknowledgment is not pending.
New	The resulting state is unchanged.

Acknowledgments

The DLS provider responds to the information request with a **DL_INFO_ACK** primitive.

Related reference:

"DL_INFO_ACK Primitive" on page 73

DL_OK_ACK Primitive

Purpose

Acknowledges that a previously issued primitive was received successfully.

Structure

The message consists of one **M_PCPROTO** message block, which contains the following structure:

```
typedef struct
{
   ulong dl_primitive;
   ulong dl_correct_primitive;
} dl_ok_ack_t;
```

This structure is defined in **/usr/include/sys/dlpi.h**.

Description

The **DL_OK_ACK** primitive acknowledges to the data link service (DLS) user that a previously issued primitive was received successfully. It is only initiated for the primitives listed in the "States" section.

Parameters

Item	Description
dl_primitive	Specifies the **DL_OK_ACK** primitive.
dl_correct_primitive	Identifies the received primitive that is being acknowledged.

States

Item	Description
Valid	The primitive is valid in response to the following primitives: • **DL_ATTACH_REQ** • **DL_DETACH_REQ** • **DL_UNBIND_REQ** • **DL_SUBS_UNBIND_REQ** • **DL_PROMISCON_REQ** • **DL_ENABMULTI_REQ** • **DL_DISABMULTI_REQ** • **DL_PROMISCOFF_REQ**
New	The resulting state depends on the current state and is fully defined in "Allowable Sequence of DLPI Primitives" in your copy of the AT&T DLPI Specifications.

Related reference:

"DL_ATTACH_REQ Primitive" on page 50

"DL_UNBIND_REQ Primitive" on page 97

"DL_PROMISCON_REQ Primitive" on page 81

"DL_ENABMULTI_REQ Primitive" on page 69

DL_PHYS_ADDR_ACK Primitive

Purpose

Returns the value for the physical address to the data link service (DLS) user in response to a **DL_PHYS_ADDR_REQ** primitive.

Structure

The message consists of one **M_PCPROTO** message block, which contains the following structure:

```
typedef struct
{
   ulong  dl_primitive;
   ulong  dl_addr_length;
   ulong  dl_addr_offset;
} dl_phys_addr_req_t;
```

This structure is defined in **/usr/include/sys/dlpi.h**.

Description

The **DL_PHYS_ADDR_ACK** primitive returns the value for the physical address to the DLS user in response to a **DL_PHYS_ADDR_REQ** primitive.

Parameters

Item	Description
dl_primitive	Specifies the **DL_ PHYS_ADDR_ACK** primitive.
dl_addr_length	Specifies the length of the physical address.
dl_addr_offset	Indicates where the physical address begins. The value of this parameter is the offset from the beginning of the **M_PCPROTO** block.

States

Item	Description
Valid	The primitive is valid in any state in response to a **DL_PHYS_ADDR_REQ** primitive.
New	The resulting state is unchanged.

Related reference:
"DL_PHYS_ADDR_REQ Primitive"

DL_PHYS_ADDR_REQ Primitive

Purpose

Requests that the data link service (DLS) provider return the current value of the physical address associated with the stream.

Structure

The message consists of one **M_PROTO** message block, which contains the following structure:

```
typedef struct
{
   ulong  dl_primitive;
   ulong  dl_addr_type;
} dl_phys_addr_req_t;
```

This structure is defined in **/usr/include/sys/dlpi.h**.

Description

The **DL_PHYS_ADDR_REQ** primitive requests that the DLS provider return the current value of the physical address associated with the stream.

Parameters

Item	Description
dl_primitive	Specifies the **DL_PHYS_ADDR_REQ** primitive.
dl_addr_type	Specifies the requested address. The value is: **DL_CURR_PHYS_ADDR** Current physical address.

States

Item	Description
Valid	The primitive is valid in any attached state in which a local acknowledgment is not pending. For a *style 2* DLS provider, this is after a PPA is attached using the **DL_ATTACH_REQ** provider.
New	The resulting state is unchanged.

Acknowledgments

Item	Description
Successful	The **DL_PHYS_ADDR_ACK** primitive is sent to the DLS user.
Unsuccessful	The **DL_ERROR_ACK** primitive is returned to the DLS user.

Error Codes

Item	Description
DL_NOTSUPPORTED	Indicates the primitive is known but not supported by the DLS provider.
DL_OUTSTATE	Indicates the primitive was issued from an invalid state.
DL_UNSUPPORTED	Indicates the requested address type is not supplied by the DLS provider.
DL_SYSERR	Indicates a system error occurred and the provider did not have access to the physical address.

Related reference:

"DL_PHYS_ADDR_ACK Primitive" on page 77

"DL_ERROR_ACK Primitive" on page 70

DL_PROMISCOFF_REQ Primitive

Purpose

Requests that the data link service (DLS) provider disable promiscuous mode on a per-stream basis, at either the physical level or the service access point (SAP) level.

Structure

The message consists of one **M_PROTO** message block, which contains the following structure:

```
typedef struct
{
   ulong  dl_primitive;
   ulong  dl_level;
} dl_promiscoff_req_t;
```

This structure is defined in **/usr/include/sys/dlpi.h**.

Description

A device in promiscuous mode lets a user view *all* packets, not just those destined for the user.

The **DL_PROMISCOFF_REQ** primitive requests that the DLS provider disable promiscuous mode on a per-stream basis, at either the physical level or the SAP level.

If the DLS user disables the promiscuous mode at the physical level, the DLS user no longer receives a copy of every packet on the wire for all SAPs.

If the DLS user disables the promiscuous mode at the SAP level, the DLS user no longer receives a copy of every packet on the wire directed to that user for all SAPs.

If the DLS user disables the promiscuous mode for all multicast addresses, the DLS user no longer receives all packets on the wire that have either a multicast or group destination address. This includes broadcast.

An application issuing the **DL_PROMISCOFF_REQ** primitive must have root authority. Otherwise, the DLS provider returns the **DL_ERROR_ACK** primitive with an error code of **DL_ACCESS**.

The DLS provider must not run in the interrupt environment. If it does, the system returns a **DL_ERROR_ACK** primitive with an error code of **DL_SYSERR** and an operating system error code of 0.

Parameters

Item	Description
dl_primitive	Specifies the **DL_PROMISCOFF_REQ** primitive.
dl_level	Indicates promiscuous mode at the physical or SAP level. Possible values include: **DL_PROMISC_PHYS** Indicates promiscuous mode at the physical level. **DL_PROMISC_SAP** Indicates promiscuous mode at the SAP level. **DL_PROMISC_MULTI** Indicates promiscuous mode for all multicast addresses.

States

Item	Description
Valid	The primitive is valid in any state in which an acknowledgement is not pending, with the exception of **DL_UNATTACH**.
New	The resulting state is unchanged.

Acknowledgments

Item	Description
Successful	The **DL_OK_ACK** primitive is sent to the DLS user.
Unsuccessful	The **DL_ERROR_ACK** primitive is returned, and the resulting state is unchanged.

Error Codes

Item	Description
DL_ACCESS	Indicates the DLS user does not have permission to issue the primitive.
DL_NOTENAB	Indicates the mode is not enabled.
DL_NOTSUPPORTED	Indicates the primitive is known but not supported by the DLS provider.
DL_OUTSTATE	Indicates the primitive was issued from an invalid state.
DL_SYSERR	Indicates a system error occurred. The system error is indicated in the **DL_ERROR_ACK** primitive.
DL_UNSUPPORTED	Indicates the DLS provider does not supply the requested level.

Related reference:

"DL_OK_ACK Primitive" on page 76

"DL_ERROR_ACK Primitive" on page 70

"DL_PROMISCON_REQ Primitive" on page 81

DL_PROMISCON_REQ Primitive

Purpose

Requests that the data link service (DLS) provider enable promiscuous mode on a per stream basis, at either the physical level or the service access point (SAP) level.

Structure

The message consists of one **M_PROTO** message block, which contains the following structure:

```
typedef struct
{
   ulong  dl_primitive;
   ulong  dl_level;
} dl_promiscon_req_t;
```

This structure is defined in **/usr/include/sys/dlpi.h**.

Description

A device in promiscuous mode lets a user view *all* packets, not just those destined for the user.

The **DL_PROMISCON_REQ** primitive requests that the DLS provider enable promiscuous mode on a per-stream basis, either at the physical level or at the SAP level.

The DLS provider routes all received messages on the media to the DLS user until either a **DL_DETACH_REQ** or a **DL_PROMISCOFF_REQ** primitive is received or the stream is closed.

If the DLS user enables the promiscuous mode at the physical level, the DLS user receives a copy of every packet on the wire for all SAPs.

If the DLS user enables the promiscuous mode at the SAP level, the DLS user receives a copy of every packet on the wire directed to that user for all SAPs.

If the DLS user enables the promiscuous mode for all multicast addresses, the DLS user receives all packets on the wire that have either a multicast or group destination address. This includes broadcast.

If the DLS user issues duplicate requests, the system returns a **DL_OK_ACK** primitive and does not perform the operation.

An application issuing the **DL_PROMISCON_REQ** primitive must have root authority. Otherwise, the DLS provider returns the **DL_ERROR_ACK** primitive with an error code of **DL_ACCESS**.

The DLS provider must not run in the interrupt environment. If it does, the system returns a **DL_ERROR_ACK** primitive with an error code of **DL_SYSERR** and an operating system error code of 0.

The above code fragment .

The following sample code fragment discards the **DL_UNITDATA_IND** header, and will work with **dlpi**:

```
if (raw_mode) {
if (mp->b_datap->db_type == M_PROTO) {
union DL_primitives *p;
p = (union DL_primitives *)mp->b_rptr;
if (p->dl_primitive == DL_UNITDATA_IND) {
```

```
mblk_t *mpl = mp->b_cont;
freeb(mp);
mp = mpl;
}
}
}
```

For compatibility with future releases, it is recommended that you parse the frame yourself. The MAC and LLC headers are presented in the M_DATA message for promiscuous mode.

Parameters

Item	Description
dl_primitive	Specifies the **DL_PROMISCON_REQ** primitive.
dl_level	Indicates promiscuous mode at the physical or SAP level. Possible values include: **DL_PROMISC_PHYS** Indicates promiscuous mode at the physical level. **DL_PROMISC_SAP** Indicates promiscuous mode at the SAP level. **DL_PROMISC_MULTI** Indicates promiscuous mode for all multicast addresses.

States

Item	Description
Valid	The primitive is valid in any state in which an acknowledgement is not pending, with the exception of **DL_UNATTACH**.
New	The resulting state is unchanged.

Acknowledgments

Item	Description
Successful	The **DL_OK_ACK** primitive is sent to the DLS user.
Unsuccessful	The **DL_ERROR_ACK** primitive is returned, and the resulting state is unchanged.

Error Codes

Item	Description
DL_ACCESS	Indicates the DLS user does not have permission to issue the primitive.
DL_NOTSUPPORTED	Indicates the primitive is known but not supported by the DLS provider.
DL_OUTSTATE	Indicates the primitive was issued from an invalid state.
DL_SYSERR	Indicates a system error occurred. The system error is indicated in the **DL_ERROR_ACK** primitive.
DL_UNSUPPORTED	Indicates the DLS provider does not support the requested service on this stream.

Related reference:

"DL_OK_ACK Primitive" on page 76

"DL_ERROR_ACK Primitive" on page 70

"DL_DETACH_REQ Primitive" on page 64

"DL_PROMISCOFF_REQ Primitive" on page 79

DL_RESET_CON Primitive

Purpose

Informs the data link service (DLS) user that the reset has been completed.

Structure

The primitive consists of one **M_PROTO** message block, which contains the following structure:

```
typedef struct
{
   ulong  dl_primitive;
} dl_reset_con_t;
```

Description

The **DL_RESET_CON** primitive informs the DLS user initiating the reset that the reset has been completed.

Note: This primitive applies to connection mode.

Parameters

Item	Description
dl_primitive	Specifies the **DL_RESET_CON** primitive.

States

Item	Description
Valid	The primitive is valid in the **DL_USER_RESET_PENDING** state.
New	The resulting state is **DL_DATAXFER**.

Related reference:

"DL_RESET_IND Primitive"

"DL_RESET_REQ Primitive" on page 84

DL_RESET_IND Primitive

Purpose

Indicates a data link service (DLS) connection has been reset.

Structure

The primitive consists of one **M_PROTO** message block, which contains the following structure:

```
typedef struct
{
   ulong  dl_primitive;
   ulong  dl_originator;
   ulong  dl_reason;
} dl_disconnect_ind_t;
```

Description

The **DL_RESET_IND** primitive informs the DLS user that either the remote DLS user is resynchronizing the data link connection, or the DLS provider is reporting loss of data from which it can not recover. The primitive indicates the reason for the reset.

Note: This primitive applies to connection mode.

Parameters

Item	Description
dl_primitive	Specifies the **DL_RESET_IND** primitive.
dl_originator	Specifies whether the reset was originated by the DLS user or DLS provider. The values are **DL_USER** or **DL_PROVIDER**, respectively.
dl_reason	Indicates one of the following reasons for the reset: **DL_RESET_FLOW_CONTROL** Indicates flow control congestion. **DL_RESET_LINK_ERROR** Indicates the occurrence of a data link error. **DL_RESET_RESYNCH** Indicates a request for resynchronization of a data link connection.

States

Item	Description
Valid	The primitive is valid in the **DL_DATAXFER** state.
New	The resulting state is **DL_PROV_RESET_PENDING**.

Acknowledgments

The DLS user should issue a **DL_RESET_RES** primitive to continue the resynchronization procedure.

Related reference:

"DL_DATA_IND Primitive" on page 62

"DL_DATA_REQ Primitive" on page 63

"DL_RESET_CON Primitive" on page 83

"DL_RESET_RES Primitive" on page 85

DL_RESET_REQ Primitive

Purpose

Requests that the data link service (DLS) provider begin resynchronizing a data link connection.

Structure

The primitive consists of one **M_PROTO** message block, which contains the following structure:

```
typedef struct
{
   ulong  dl_primitive;
} dl_reset_req_t;
```

Description

The **DL_RESET_REQ** primitive requests that the DLS provider begin resynchronizing a data link connection.

Note:

1. No guarantee exists that data in transit when the **DL_RESET_REQ** primitive is initiated will be delivered.
2. This primitive applies to connection mode.

Parameters

Item	Description
dl_primitive	Specifies the **DL_RESET_REQ** primitive.

States

Item	Description
Valid	The primitive is valid in state **DL_DATAXFER**.
New	The resulting state is **DL_USER_RESET_PENDING**.

Acknowledgments

Item	Description
Successful	There is no immediate response to the reset request. However, as resynchronization completes, the **DL_RESET_CON** primitive is sent to the initiating DLS user, resulting in the **DL_DATAXFER** state.
Unsuccessful	The **DL_ERROR_ACK** primitive is returned and the resulting state is unchanged.

Error Codes

Item	Description
DL_OUTSTATE	Indicates the primitive was issued from an invalid state.
DL_SYSERR	Indicates a system error occurred. The system error is indicated in the **DL_ERROR_ACK** primitive.

Related reference:

"DL_RESET_CON Primitive" on page 83

"DL_ERROR_ACK Primitive" on page 70

DL_RESET_RES Primitive

Purpose

Directs the data link service (DLS) provider to complete resynchronizing the data link connection.

Structure

The primitive consists of one **M_PROTO** message block, which contains the following structure:

```
typedef struct
{
   ulong  dl_primitive;
} dl_reset_res_t;
```

Description

The **DL_RESET_RES** primitive directs the DLS provider to complete resynchronizing the data link connection.

Note: This primitive applies to connection mode.

Parameters

Item	Description
dl_primitive	Specifies the **DL_RESET_RES** primitive.

States

Item	Description
Valid	The primitive is valid in the **DL_PROV_RESET_PENDING** state.
New	The resulting state is **DL_RESET_RES_PENDING**.

Acknowledgments

Item	Description
Successful	The **DL_OK_ACK** primitive is sent to the DLS user, and the resulting state is **DL_DATAXFER**.
Unsuccessful	The **DL_ERROR_ACK** primitive is returned, and the resulting state is unchanged.

Error Codes

Item	Description
DL_OUTSTATE	Indicates the primitive was issued from an invalid state.
DL_SYSERR	Indicates a system error occurred. The system error is indicated in the **DL_ERROR_ACK** primitive.

Related reference:
"DL_RESET_IND Primitive" on page 83

DL_SUBS_BIND_ACK Primitive

Purpose

Reports the successful bind of a subsequent data link service access point (DLSAP) to a stream and returns the bound DLSAP address to the data link service (DLS) user.

Structure

The message consists of one **M_PCPROTO** message block, which contains the following structure:

```
typedef struct
{
   ulong  dl_primitive;
   ulong  dl_subs_sap_length;
   ulong  dl_subs_sap_offset;
} dl_subs_bind_ack_t;
```

This structure is defined in **/usr/include/sys/dlpi.h**.

Description

The **DL_SUBS_BIND_ACK** primitive reports the successful bind of a subsequent DLSAP to a stream and returns the bound DLSAP address to the DLS user. This primitive is generated in response to a **DL_BIND_REQ** primitive.

Parameters

Item	Description
dl_primitive	Specifies the **DL_SUBS_BIND_ACK** primitive.
dl_subs_sap_length	Specifies the length of the specified DLSAP.
dl_subs_sap_offset	Indicates where the DLSAP begins. The value of this parameter is the offset from the beginning of the **M_PROTO** message block.

States

Item	Description
Valid	The primitive is valid in the **DL_SUBS_BIND_PND** state.
New	The resulting state is **DL_IDLE**.

Related reference:
"DL_SUBS_BIND_REQ Primitive"

DL_SUBS_BIND_REQ Primitive

Purpose

Requests that the data link service (DLS) provider bind a subsequent data link service access point (DLSAP) to the stream.

Structure

The message consists of one **M_PROTO** message block, which contains the following structure:

```
typedef struct
{
   ulong  dl_primitive;
   ulong  dl_subs_sap_offset;
   ulong  dl_subs_sap_length;
   ulong  dl_subs_bind_class;
} dl_subs_bind_req_t;
```

This structure is defined in **/usr/include/sys/dlpi.h**.

Description

The **DL_SUBS_BIND_REQ** primitive requests that the DLS provider bind a subsequent DLSAP to the stream. The DLS user must identify the address of the subsequent DLSAP to be bound to the stream.

The 802.2 networks accept either **DL_HIERARCHICAL_BIND** or **DL_PEER_BIND**. The *dl_subs_sap_length* parameter must be 5 (sizeof snap) for hierarchical binds, and *dl_subs_sap_offset* must point to a complete SNAP. For peer binds, *dl_subs_sap_length* may be either 1 or 5, and *dl_subs_sap_offset* must point to either a single byte SAP or a complete SNAP (as in hierarchical binds).

In the case of SNAP binds, **DL_PEER_BIND** and **DL_HIERARCHICAL_BIND** are synonymous, and fully interchangeable.

Several distinct SAPs/SNAPs may be bound on any single stream. Since a DSAP address field is limited to 8 bits, a maximum of 256 SAPS/SNAPS can be bound to a single stream. Closing the stream or issuing **DL_UNBIND_REQ** causes all SAPs and SNAPs to be unbound automatically, or each subs sap can be individually unbound.

DL_ETHER supports only **DL_PEER_BIND**, and *dl_subs_sap_offset* must point to an ethertype (dl_subs_sap_length == sizeof(ushort)).

Examples:

Preferred Request	Sap
DL_BIND_REQ	0xaa
DL_SUBS_BIND_REQ/DL_HIERARCHICAL_BIND	08.00.07.80.9b
DL_SUBS_BIND_REQ/DL_HIERARCHICAL_BIND	08.00.07.80.f3

or

Equivalent Effect	Sap
DL_BIND_REQ	0xaa
DL_SUBS_BIND_REQ/DL_PEER_BIND	08.00.07.80.9b
DL_SUBS_BIND_REQ/DL_PEER_BIND	08.00.07.80.f3

or

Equivalent Effect	Sap
DL_BIND_REQ	0xaa
DL_SUBS_BIND_REQ/DL_HIERARCHICAL_BIND	08.00.07.80.9b
DL_SUBS_BIND_REQ/DL_PEER_BIND	08.00.07.80.f3

Parameters

Item	Description
dl_primitive	Specifies the **DL_SUBS_BIND_REQ** primitive.
dl_subs_sap_length	Specifies the length of the specified DLSAP.
dl_subs_sap_offset	Indicates where the DLSAP begins. The value of this parameter is the offset from the beginning of the **M_PROTO** message block.
dl_subs_bind_class	Specifies either peer or hierarchical addressing. Possible values include: **DL_PEER_BIND** Specifies peer addressing. The DLSAP specified is used instead of the DLSAP bound in the bind request. **DL_HIERARCHICAL_BIND** Specifies hierarchical addressing. The DLSAP specified is used in addition to the DLSAP specified using the bind request.

States

Item	Description
Valid	The primitive is valid in the **DL_IDLE** state.
New	The resulting state is **DL_SUBS_BIND_PND**.

Acknowledgments

Item	Description
Successful	The **DL_SUBS_BIND_ACK** primitive is sent to the DLS user, and the resulting state is **DL_IDLE**.
Unsuccessful	The **DL_ERROR_ACK** primitive is returned, and the resulting state is unchanged.

Error Codes

Item	Description
DL_ACCESS	Indicates the DLS user does not have proper permission to use the requested DLSAP address.
DL_BADADDR	Indicates the DLSAP address information is invalid or is in an incorrect format.
DL_OUTSTATE	Indicates the primitive was issued from an invalid state.
DL_SYSERR	Indicates a system error occurred. The system error is indicated in the **DL_ERROR_ACK** primitive.
DL_TOOMANY	Indicates the limit has been exceeded for the maximum number of DLSAPs per stream.
DL_UNSUPPORTED	Indicates the DLS provider does not support the requested addressing class.

Related reference:

"DL_SUBS_BIND_ACK Primitive" on page 86

"DL_ERROR_ACK Primitive" on page 70

"DL_SUBS_UNBIND_REQ Primitive"

"DL_UNBIND_REQ Primitive" on page 97

DL_SUBS_UNBIND_REQ Primitive

Purpose

Requests that the data link service (DLS) provider unbind the data link service access point (DLSAP) that was bound by a previous **DL_SUBS_BIND_REQ** primitive from this stream.

Structure

The message consists of one **M_PROTO** message block, which contains the following structure:

```
typedef struct
{
   ulong  dl_primitive;
   ulong  dl_subs_sap_length;
   ulong  dl_subs_sap_offset;
} dl_subs_unbind_req_t;
```

This structure is defined in **/usr/include/sys/dlpi.h**.

Description

The **DL_SUBS_UNBIND_REQ** primitive requests that the DLS provider unbind the DLSAP that was bound by a previous **DL_SUBS_BIND_REQ** primitive from this stream.

Parameters

Item	Description
dl_primitive	Specifies the **DL_SUBS_UNBIND_REQ** primitive.
dl_subs_sap_length	Specifies the length of the specified DLSAP.
dl_subs_sap_offset	Indicates where the DLSAP begins. The value of this parameter is the offset from the beginning of the **M_PROTO** message block.

States

Item	Description
Valid	The primitive is valid in the **DL_IDLE** state.
New	The resulting state is **DL_SUBS_UNBIND_PND**.

Acknowledgments

Item	Description
Successful	The **DL_OK_ACK** primitive is sent to the DLS user. The resulting state is **DL_IDLE**.
Unsuccessful	The **DL_ERROR_ACK** primitive is returned, and the resulting state is unchanged.

Error Codes

Item	Description
DL_BADADDR	Indicates the DLSAP address information is invalid or is in an incorrect format.
DL_OUTSTATE	Indicates the primitive was issued from an invalid state.
DL_SYSERR	Indicates a system error occurred. The system error is indicated in the **DL_ERROR_ACK** primitive.

Related reference:

"DL_OK_ACK Primitive" on page 76

"DL_ERROR_ACK Primitive" on page 70

"DL_SUBS_BIND_REQ Primitive" on page 87

DL_TEST_CON Primitive

Purpose

Conveys the test-response data link service data unit (DLSDU) from the data link service (DLS) provider to the DLS user in response to a **DL_TEST_REQ** primitive.

Structure

The primitive consists of one **M_PROTO** message block, which contains the following structure, followed by zero or more **M_DATA** blocks containing zero or more bytes of data:

```
typedef struct
{
   ulong  dl_primitive;
   ulong  dl_flag;
   ulong  dl_dest_addr_length;
   ulong  dl_dest_addr_offset;
   ulong  dl_src_addr_length;
   ulong  dl_src_addr_offset;
} dl_test_con_t;
```

Description

The **DL_TEST_CON** primitive conveys the test-response DLSDU from the DLS provider to the DLS user in response to a **DL_TEST_REQ** primitive.

Note: This primitive applies to XID and test operations.

Parameters

Item	Description
dl_primitive	Specifies the **DL_TEST_CON** primitive.
dl_flag	Indicates flag values for the request as follows: **DL_POLL_FINAL** Indicates whether the poll/final bit is set.
dl_dest_addr_length	Specifies the length of the data link service access point (DLSAP) address of the destination DLS user. If the destination user is implemented using Data Link Provider Interface (DLPI), this address is the full DLSAP address returned on the **DL_BIND_ACK** primitive.
dl_dest_addr_offset	Indicates where the destination DLSAP address begins. The value of this parameter is the offset from the beginning of the **M_PROTO** message block.
dl_src_addr_length	Specifies the length of the DLSAP address of the source DLS user.
dl_src_addr_offset	Indicates where the source DLSAP address begins. The value of this parameter is the offset from the beginning of the **M_PROTO** message block.

States

Item	Description
Valid	The primitive is valid in the **DL_IDLE** or **DL_DATAXFER** state.
New	The resulting state is unchanged.

Related reference:
"DL_BIND_ACK Primitive" on page 51

DL_TEST_IND Primitive

Purpose

Conveys the test-response indication data link service data unit (DLSDU) from the data link service (DLS) provider to the DLS user.

Structure

The primitive consists of one **M_PROTO** message block, which contains the following structure, followed by zero or more **M_DATA** blocks containing zero or more bytes of data:

```
typedef struct
{
   ulong dl_primitive;
   ulong dl_flag;
   ulong dl_dest_addr_length;
   ulong dl_dest_addr_offset;
   ulong dl_src_adr_length;
   ulong dl_src_addr_offset;
} dl_test_ind_t;
```

Description

The **DL_TEST_IND** primitive conveys the test-response indication DLSDU from the DLS provider to the DLS user.

Note: This primitive applies to XID and test operations.

Parameters

Item	Description
dl_primitive	Specifies the **DL_TEST_IND** primitive.
dl_flag	Indicates flag values for the request as follows: **DL_POLL_FINAL** Indicates whether the poll/final bit is set.
dl_dest_addr_length	Specifies the length of the data link service access point (DLSAP) address of the destination DLS user. If the destination user is implemented using the Data Link Provider Interface (DLPI), this address is the full DLSAP address returned on the **DL_BIND_ACK** primitive.
dl_dest_addr_offset	Indicates where the destination DLSAP address begins. The value of this parameter is the offset from the beginning of the **M_PROTO** message block.
dl_src_addr_length	Specifies the length of the DLSAP address of the source DLS user.
dl_src_addr_offset	Indicates where the source DLSAP address begins. The value of this parameter is the offset from the beginning of the **M_PROTO** message block.

States

Item	Description
Valid	The primitive is valid in the **DL_IDLE** or **DL_DATAXFER** state.
New	The resulting state is unchanged.

Related reference:
"DL_BIND_ACK Primitive" on page 51

DL_TEST_REQ Primitive

Purpose

Conveys one test-command data link service data unit (DLSDU) from the data link service (DLS) user to the DLS provider for transmission to a peer DLS provider.

Structure

The message consists of one **M_PROTO** message block, which contains the following structure, followed by zero or more **M_DATA** blocks containing zero or more bytes of data:

```
typedef struct
{
   ulong  dl_primitive;
   ulong  dl_flag;
   ulong  dl_dest_addr_length;
   ulong  dl_dest_addr_offset;
} dl_test_req_t;
```

Description

The **DL_TEST_REQ** primitive conveys one test-command DLSDU from the DLS user to the DLS provider for transmission to a peer DLS provider.

A **DL_ERROR_ACK** primitive is always returned.

Note: This primitive applies to XID and test operations.

Parameters

Item	Description
dl_primitive	Specifies the **DL_TEST_REQ** primitive.
dl_flag	Indicates flag values for the request as follows: **DL_POLL_FINAL** Indicates whether the poll/final bit is set.
dl_dest_addr_length	Specifies the length of the data link service access point (DLSAP) address of the destination DLS user. If the destination user is implemented using the Data Link Provider Interface (DLPI), this address is the full DLSAP address returned on the **DL_BIND_ACK** primitive.
dl_dest_addr_offset	Indicates where the destination DLSAP address begins. The value of this parameter is the offset from the beginning of the **M_PROTO** message block.

States

Item	Description
Valid	The primitive is valid in the **DL_IDLE** or **DL_DATAXFER** state.
New	The resulting state is unchanged.

Acknowledgments

Item	Description
Unsuccessful	The **DL_ERROR_ACK** primitive is returned for an invalid test-command request.

Note: It is recommended that the DLS user use a timeout procedure to recover from a situation when the peer DLS user does not respond.

Error Code

Item	Description
DL_OUTSTATE	The primitive was issued from an invalid state.
DL_BADADDR	The DLSAP address information was invalid or was in an incorrect format.
DL_BADDATA	The amount of data in the current DLSDU exceeded the DLS provider's DLSDU limit.
DL_SYSERR	A system error has occurred. The system error is indicated in the **DL_ERROR_ACK** primitive.
DL_TESTAUTO	Indicates the previous bind request specified automatic handling of test responses.

Related reference:

"DL_BIND_ACK Primitive" on page 51

"DL_ERROR_ACK Primitive" on page 70

DL_TEST_RES Primitive

Purpose

Conveys the test-response data link service data unit (DLSDU) from the data link service (DLS) user to the DLS provider in response to a **DL_TEST_IND** primitive.

Structure

The primitive consists of one **M_PROTO** message block, which contains the following structure, followed by zero or more **M_DATA** blocks containing zero or more bytes of data:

```
typedef struct
{
   ulong  dl_primitive;
```

```
   ulong  dl_flag;
   ulong  dl_dest_addr_length;
   ulong  dl_dest_addr_offset;
} dl_test_res_t;
```

Description

The **DL_TEST_RES** primitive conveys the test-response DLSDU from the DLS user to the DLS provider in response to a **DL_TEST_IND** primitive.

Note: This primitive applies to XID and test operations.

Parameters

Item	Description
dl_primitive	Specifies the **DL_TEST_RES** primitive.
dl_flag	Indicates flag values for the request as follows: **DL_POLL_FINAL** Indicates whether the poll/final bit is set.
dl_dest_addr_length	Specifies the length of the data link service access point (DLSAP) address of the destination DLS user. If the destination user is implemented using the Data Link Provider Interface (DLPI), this address is the full DLSAP address returned on the **DL_BIND_ACK** primitive.
dl_dest_addr_offset	Indicates where the destination DLSAP address begins. The value of this parameter is the offset from the beginning of the **M_PROTO** message block.

States

Item	Description
Valid	The primitive is valid in the **DL_IDLE** or **DL_DATAXFER** state.
New	The resulting state is unchanged.

Related reference:
"DL_BIND_ACK Primitive" on page 51

DL_TOKEN_ACK Primitive

Purpose

Specifies the connection-response token assigned to a stream.

Structure

The primitive consists of one **M_PCPROTO** message block, which contains the following structure:

```
typedef struct
{
   ulong  dl_primitive;
   ulong  dl_token;
} dl_token_req_t;
```

Description

The **DL_TOKEN_ACK** primitive is sent in response to the **DL_TOKEN_REQ** primitive. The **DL_TOKEN_ACK** primitive specifies the connection-response token assigned to the stream.

Note: This primitive applies to connection mode.

Parameters

Item	Description
dl_primitive	Specifies the **DL_TOKEN_ACK** primitive.
dl_token	Specifies the connection-response token associated with a stream. This value must be a nonzero value. After an initial **DL_TOKEN_REQ** primitive is issued on a stream, the data link service (DLS) provider generates the same token value for each subsequent **DL_TOKEN_REQ** primitive issued on the stream. The DLS provider generates a token value for each stream upon receipt of the first **DL_TOKEN_REQ** primitive issued on that stream. The same token value is returned in response to all subsequent **DL_TOKEN_REQ** primitives issued on a stream.

States

Item	Description
Valid	The primitive is valid in any state in response to a **DL_TOKEN_REQ** primitive.
New	The resulting state is unchanged.

Related reference:
"DL_TOKEN_REQ Primitive"

DL_TOKEN_REQ Primitive

Purpose

Requests that a connection-response token be assigned to the stream and returned to the data link service (DLS) user.

Structure

The primitive consists of one **M_PCPROTO** message block, which contains the following structure:

```
typedef struct
{
   ulong  dl_primitive;
} dl_token_req_t;
```

Description

The **DL_TOKEN_REQ** primitive requests that a connection-response token be assigned to the stream and returned to the DLS user. This token can be supplied in the **DL_CONNECT_RES** primitive to indicate the stream on which a connection is to be established.

Note: This primitive applies to connection mode.

Parameters

Item	Description
dl_primitive	Specifies the **DL_TOKEN_REQ** primitive.

States

Item	Description
Valid	The primitive is valid in any state in which a local acknowledgement is not pending.
New	The resulting state is unchanged.

Acknowledgments

The DLS provider responds to the information request with a **DL_TOKEN_ACK** primitive.

Related reference:

"DL_TOKEN_ACK Primitive" on page 94

"DL_CONNECT_RES Primitive" on page 60

DL_UDERROR_IND Primitive

Purpose

Informs the data link service (DLS) user that a previously sent **DL_UNITDATA_REQ** primitive produced an error or could not be delivered.

Structure

The message consists of either one **M_PROTO** message block or one **M_PCPROTO** message block, which contains the following structure:

```
typedef struct
{
   ulong  dl_primitive;
   ulong  dl_dest_addr_length;
   ulong  dl_dest_addr_offset;
   ulong  dl_unix_errno;
   ulong  dl_errno;
} dl_uderror_ind_t;
```

This structure is defined in **/usr/include/sys/dlpi.h**.

Description

The **DL_UDERROR_IND** primitive informs the DLS user that a previously sent **DL_UNITDATA_REQ** primitive produced an error or could not be delivered. The primitive indicates the destination DLSAP address associated with the failed request, and returns an error value that specifies the reason for failure.

There is, however, no guarantee that such an error report will be generated for all undeliverable data units, because connectionless-mode data transfer is not a confirmed service.

Parameters

Item	Description
dl_primitive	Specifies the **DL_UDERROR_IND** primitive.
dl_dest_addr_length	Specifies the length of the DLSAP address of the destination DLS user.
dl_dest_addr_offset	Indicates where the destination DLSAP address begins. The value of this parameter is the offset from the beginning of the **M_PROTO** message block.
dl_unix_errno	Specifies the operating system code associated with the failure. This value should be nonzero only when the *dl_errno* parameter is set to **DL_SYSERR**. It is used to report operating system failures that prevent the processing of a given request or response.
dl_errno	Indicates the Data Link Provider Interface (DLPI) error code associated with the failure. Possible values include: **DL_BADADDR** Indicates the DLSAP address information is invalid or is in an incorrect format. **DL_OUTSTATE** Indicates the primitive was issued from an invalid state. **DL_UNSUPPORTED** Indicates the DLS provider does not support the requested priority. **DL_UNDELIVERABLE** Indicates the request was valid but for some reason the DLS provider could not deliver the data unit (for example, due to lack of sufficient local buffering to store the data unit).

States

Item	Description
Valid	The primitive is valid in the **DL_IDLE** state.
New	The resulting state is unchanged.

Related reference:

"DL_UNITDATA_REQ Primitive" on page 99

"DL_UNITDATA_IND Primitive" on page 98

DL_UNBIND_REQ Primitive

Purpose

Requests the data link service (DLS) provider to unbind a data link service access point (DLSAP).

Structure

The message consists of one **M_PROTO** message block, which contains the following structure:

```
typedef struct
{
   ulong  dl_primitive;
} dl_unbind_req_t;
```

This structure is defined in **/usr/include/sys/dlpi.h**.

Description

The **DL_UNBIND_REQ** primitive requests that the DLS provider unbind the DLSAP that had been bound by a previous **DL_BIND_REQ** primitive. If one or more DLSAPs were bound to the stream with a **DL_SUBS_BIND_REQ** primitive and have not been unbound with a **DL_SUBS_UNBIND_REQ** primitive, the **DL_UNBIND_REQ** primitive unbinds all the subsequent DLSAPs for that stream along with the DLSAP bound with the previous **DL_BIND_REQ** primitive.

At the successful completion of the request, the DLS user can issue a new **DL_BIND_REQ** primitive for a potentially new DLSAP.

Parameters

Item	Description
dl_primitive	Specifies the **DL_UNBIND_REQ** primitive.

States

Item	Description
Valid	The primitive is valid in the **DL_IDLE** state.
New	The resulting state is **DL_UNBIND_PENDING**.

Acknowledgments

Item	Description
Successful	The **DL_OK_ACK** primitive is sent to the DLS user, and the resulting state is **DL_UNBOUND**.
Unsuccessful	The **DL_ERROR_ACK** primitive is returned, and the resulting state is unchanged.

Error Codes

Item	Description
DL_OUTSTATE	Indicates the primitive was issued from an invalid state.
DL_SYSERR	Indicates a system error occurred. The system error is indicated in the **DL_ERROR_ACK** primitive.

Related reference:

"DL_OK_ACK Primitive" on page 76

"DL_BIND_REQ Primitive" on page 53

"DL_ERROR_ACK Primitive" on page 70

"DL_SUBS_BIND_REQ Primitive" on page 87

DL_UNITDATA_IND Primitive

Purpose

Conveys one data link service data unit (DLSDU) from the data link service (DLS) provider to the DLS user.

Structure

The message consists of one **M_PROTO** message block, which contains the following structure, followed by one or more **M_DATA** blocks containing at least one byte of data:

```
typedef struct
{
   ulong  dl_primitive;
   ulong  dl_dest_addr_length;
   ulong  dl_dest_addr_offset;
   ulong  dl_src_addr_length;
   ulong  dl_src_addr_offset;
   ulong  dl_group_address;
} dl_unitdata_ind_t;
```

This structure is defined in **/usr/include/sys/dlpi.h**.

Description

The **DL_UNITDATA_IND** primitive conveys one DLSDU from the DLS provider to the DLS user.

Note: The amount of user data that may be transferred in a single DLSDU is limited. This limit is conveyed by the *dl_max_sdu* parameter of the **DL_INFO_ACK** primitive.

Parameters

Item	Description
dl_primitive	Specifies the **DL_UNITDATA_IND** primitive.
dl_dest_addr_length	Specifies the length of the data link service access point (DLSAP) address of the destination DLS user. If the destination user is implemented using the Data Link Provider Interface (DLPI), the full DLSAP address is returned on the **DL_BIND_ACK** primitive.
dl_dest_addr_offset	Indicates where the destination DLSAP address begins. The value of this parameter is the offset from the beginning of the **M_PROTO** message block.
dl_src_addr_length	Specifies the length of the DLSAP address of the source DLS user.
dl_src_addr_offset	Indicates where the source DLSAP address begins. The value of this parameter is the offset from the beginning of the **M_PROTO** message block.
dl_group_address	Indicates the address set by the DLS provider upon receiving and passing upstream a data message when the destination address of the data message is a multicast or broadcast address.

States

Item	Description
Valid	The primitive is valid in the **DL_IDLE** state.
New	The resulting state is unchanged.

Related reference:

"DL_INFO_ACK Primitive" on page 73

"DL_BIND_ACK Primitive" on page 51

"DL_UDERROR_IND Primitive" on page 96

DL_UNITDATA_REQ Primitive

Purpose

Conveys one data link service data unit (DLSDU) from the data link service (DLS) user to the DLS provider for transmission to a peer DLS user.

Structure

The message consists of one **M_PROTO** message block, which contains the following structure, followed by one or more **M_DATA** blocks containing at least one byte of data:

```
typedef struct
{
   ulong  dl_primitive;
   ulong  dl_dest_addr_length;
   ulong  dl_dest_addr_offset;
   dl_priority_t  dl_priority;
} dl_unitdata_req_t;
```

This structure is defined in **/usr/include/sys/dlpi.h**.

Description

The **DL_UNITDATA_REQ** primitive conveys one DLSDU from the DLS user to the DLS provider for transmission to a peer DLS user.

The amount of user data that may be transferred in a single DLSDU is limited. This limit is conveyed by the *dl_max_sdu* parameter of the **DL_INFO_ACK** primitive.

Because connectionless-mode data transfer is an unacknowledged service, the DLS provider makes no guarantees of delivery of connectionless DLSDUs. It is the responsibility of the DLS user to do any necessary sequencing or retransmissions of DLSDUs in the event of a presumed loss.

Parameters

Item	Description
dl_primitive	Specifies the **DL_UNITDATA_REQ** primitive.
dl_dest_addr_length	Specifies the length of the data link service access point (DLSAP) address of the destination DLS user. If the destination user is implemented using the Data Link Provider Interface (DLPI), the full DLSAP address is returned on the **DL_BIND_ACK** primitive.
dl_dest_addr_offset	Indicates where the destination DLSAP address begins. The value of this parameter is the offset from the beginning of the **M_PROTO** message block.
dl_priority	Indicates the priority value within the supported range for this particular DLSDU.

States

Item	Description
Valid	The primitive is valid in the **DL_IDLE** state.
New	The resulting state is unchanged.

Acknowledgments

If the DLS provider accepts the data for transmission, there is no response. This does not, however, guarantee that the data will be delivered to the destination DLS user, because the connectionless-mode data transfer is not a confirmed service.

If the request is erroneous, the **DL_UDERROR_IND** primitive is returned, and the resulting state is unchanged.

If for some reason the request cannot be processed, the DLS provider may generate a **DL_UDERROR_IND** primitive to report the problem. There is, however, no guarantee that such an error report will be generated for all undeliverable data units, because connectionless-mode data transfer is not a confirmed service.

Error Codes

Item	Description
DL_BADADDR	Indicates the DLSAP address information is invalid or is in an incorrect format.
DL_BADDATA	Indicates the amount of data in the current DLSDU exceeds the DLS provider's DLSDU limit.
DL_OUTSTATE	Indicates the primitive was issued from an invalid state.
DL_UNSUPPORTED	Indicates the DLS provider does not support the requested priority.

Related reference:

"DL_UDERROR_IND Primitive" on page 96

"DL_INFO_ACK Primitive" on page 73

"DL_BIND_ACK Primitive" on page 51

DL_XID_CON Primitive

Purpose

Conveys an XID data link service data unit (DLSDU) from the data link service (DLS) provider to the DLS user in response to a **DL_XID_REQ** primitive.

Structure

The primitive consists of one **M_PROTO** message block, which contains the following structure, followed by zero or more **M_DATA** blocks containing zero or more bytes of data:

```
typedef struct
{
   ulong  dl_primitive;
   ulong  dl_flag;
   ulong  dl_dest_addr_length;
   ulong  dl_dest_addr_offset;
   ulong  dl_src_addr_length;
   ulong  dl_src_addr_offset;
} dl_xid_con_t;
```

Description

The **DL_XID_CON** conveys an XID DLSDU from the DLS provider to the DLS user in response to a **DL_XID_REQ** primitive.

Note: This primitive applies to XID and test operations.

Parameters

Item	Description
dl_primitive	Specifies the **DL_XID_CON** primitive.
dl_flag	Indicates flag values for the request as follows: **DL_POLL_FINAL** Indicates whether the poll/final bit is set.
dl_dest_addr_length	Specifies the length of the data link service access point (DLSAP) address of the destination DLS user. If the destination user is implemented using the Data Link Provider Interface (DLPI), this address is the full DLSAP address returned on the **DL_BIND_ACK** primitive.
dl_dest_addr_offset	Indicates where the destination DLSAP address begins. The value of this parameter is the offset from the beginning of the **M_PROTO** message block.
dl_src_addr_length	Specifies the length of the DLSAP address of the source DLS user.
dl_src_addr_offset	Indicates where the source DLSAP address begins. The value of this parameter is the offset from the beginning of the **M_PROTO** message block.

States

Item	Description
Valid	The primitive is valid in the **DL_IDLE** or **DL_DATAXFER** state.
New	The resulting state is unchanged.

Related reference:
"DL_BIND_ACK Primitive" on page 51
"DL_XID_REQ Primitive" on page 103

DL_XID_IND Primitive

Purpose

Conveys an XID data link service data unit (DLSDU) from the DLS provider to the data link service (DLS) user.

Structure

The primitive consists of one **M_PROTO** message block, which contains the following structure, followed by zero or more **M_DATA** blocks containing zero or more bytes of data:

```
typedef struct
{
   ulong  dl_primitive;
   ulong  dl_flag;
   ulong  dl_dest_addr_length;
   ulong  dl_dest_addr_offset;
   ulong  dl_src_addr_length;
   ulong  dl_src_addr_offset;
} dl_xid_ind_t;
```

Description

The **DL_XID_IND** primitive conveys an XID DLSDU from the DLS provider to the DLS user.

Note: This primitive applies to XID and test operations.

Parameters

Item	Description
dl_primitive	Specifies the **DL_XID_IND** primitive.
dl_flag	Indicates flag values for the request as follows: **DL_POLL_FINAL** Indicates whether the poll/final bit is set.
dl_dest_addr_length	Specifies the length of the data link service access point (DLSAP) address of the destination DLS user. If the destination user is implemented using the Data Link Provider Interface (DLPI), this address is the full DLSAP address returned on the **DL_BIND_ACK** primitive.
dl_dest_addr_offset	Indicates where the destination DLSAP address begins. The value of this parameter is the offset from the beginning of the **M_PROTO** message block.
dl_src_addr_length	Specifies the length of the DLSAP address of the source DLS user.
dl_src_addr_offset	Indicates where the source DLSAP address begins. The value of this parameter is the offset from the beginning of the **M_PROTO** message block.

States

Item	Description
Valid	The primitive is valid in the **DL_IDLE** or **DL_DATAXFER** state.
New	The resulting state is unchanged.

Related reference:
"DL_BIND_ACK Primitive" on page 51

DL_XID_REQ Primitive
Purpose

Conveys one XID data link service data unit (DLSDU) from the data link service (DLS) user to the DLS provider for transmission to a peer DLS user.

Structure

The message consists of one **M_PROTO** message block, which contains the following structure, followed by zero or more **M_DATA** blocks containing zero or more bytes of data:

```
typedef struct
{
   ulong  dl_primitive;
   ulong  dl_flag;
   ulong  dl_dest_addr_length;
   ulong  dl_dest_addr_offset;
} dl_xid_req_t;
```

This structure is defined in **/usr/include/sys/dlpi.h**.

Description

Conveys one XID DLSDU from the DLS user to the DLS provider for transmission to a peer DLS user.

A **DL_ERROR_ACK** primitive is always returned.

Note: This primitive applies to XID and test operations.

Parameters

Item	Description
dl_primitive	Specifies the **DL_XID_REQ** primitive.
dl_flag	Indicates flag values for the request as follows: **DL_POLL_FINAL** Indicates whether the poll/final bit is set.
dl_dest_addr_length	Specifies the length of the data link service access point (DLSAP) address of the destination DLS user. If the destination user is implemented using the Data Link Provider Interface (DLPI), this address is the full DLSAP address returned on the **DL_BIND_ACK** primitive.
dl_dest_addr_offset	Indicates where the destination DLSAP address begins. The value of this parameter is the offset from the beginning of the **M_PROTO** message block.

States

Item	Description
Valid	The primitive is valid in the **DL_IDLE** or **DL_DATAXFER** state.
New	The resulting state is unchanged.

Acknowledgments

Item	Description
Unsuccessful	The **DL_ERROR_ACK** primitive is returned for an invalid XID request.

Note: It is recommended that the DLS user use a timeout procedure to recover from a situation when there is no response from the peer DLS User.

Error Codes

Item	Description
DL_OUTSTATE	The primitive was issued from an invalid state.
DL_BADADDR	The DLSAP address information was invalid or was in an incorrect format.
DL_BADDATA	The amount of data in the current DLSDU exceeded the DLS provider's DLSDU limit.
DL_SYSERR	A system error has occurred. The system error is indicated in the **DL_ERROR_ACK** primitive.
DL_XIDAUTO	Indicates the previous bind request specified that the provider would handle XID.

Related reference:

"DL_XID_CON Primitive" on page 101

"DL_BIND_ACK Primitive" on page 51

"DL_ERROR_ACK Primitive" on page 70

DL_XID_RES Primitive

Purpose

Conveys an XID data link service data unit (DLSDU) from the data link service (DLS) user to the DLS provider in response to a **DL_XID_IND** primitive.

Structure

The primitive consists of one **M_PROTO** message block, which contains the following structure, followed by zero or more **M_DATA** blocks containing zero or more bytes of data:

```
typedef struct
{
   ulong  dl_primitive;
   ulong  dl_flag;
   ulong  dl_dest_addr_length;
   ulong  dl_dest_addr_offset;
} dl_xid_res_t;
```

Description

The **DL_XID_RES** primitive conveys an XID DLSDU from the DLS user to the DLS provider in response to a **DL_XID_IND** primitive.

Note: This primitive applies to XID and test operations.

Parameters

Item	Description
dl_primitive	Specifies the **DL_XID_RES** primitive.
dl_flag	Indicates flag values for the request as follows: **DL_POLL_FINAL** Indicates whether the poll/final bit is set.
dl_dest_addr_length	Specifies the length of the data link service access point (DLSAP) address of the destination DLS user. If the destination user is implemented using the Data Link Provider Interface (DLPI), this address is the full DLSAP address returned on the **DL_BIND_ACK** primitive.
dl_dest_addr_offset	Indicates where the destination DLSAP address begins. The value of this parameter is the offset from the beginning of the **M_PROTO** message block.

States

Item	Description
Valid	The primitive is valid in the **DL_IDLE** or **DL_DATAXFER** state.
New	The resulting state is unchanged.

Related reference:
"DL_BIND_ACK Primitive" on page 51

---

eXternal Data Representation

This topic collection includes the subroutines that help in external data representation in the required format.

xdr_accepted_reply Subroutine

Purpose

Encodes RPC reply messages.

Library

C Library (**libc.a**)

Syntax

```
#include <rpc/rpc.h>

int 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

Item	Description
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.

Related information:

List of RPC Programming References

eXternal Data Representation (XDR) Overview for Programming

Remote Procedure Call (RPC) Overview for Programming

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

Item	Description
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, the XDR program 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.

Related information:

List of XDR Programming References

eXternal Data Representation (XDR) Overview for Programming

xdr_bool Subroutine

Purpose

Translates between Booleans and their external representations.

Library

C Library (**libc.a**)

Syntax

```
#include <rpc/xdr.h>

xdr_bool ( xdrs,  bp)
XDR *xdrs;
bool_t *bp;
```

Description

The **xdr_bool** subroutine is a filter primitive that translates between Booleans (C integers) and their external representations. When encoding data, this filter produces values of either 1 or 0.

Parameters

Item	Description
xdrs	Points to the eXternal Data Representation (XDR) stream handle.
bp	Specifies the address of the Boolean data.

Return Values

Upon successful completion, this subroutine returns a value of 1. If unsuccessful, it returns a value of 0.

Related information:

List of XDR Programming References

eXternal Data Representation (XDR) Overview for Programming

Understanding XDR Library Filter Primitives

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 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

Item	Description
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 subroutine returns a value of 1. If unsuccessful, it returns a value of 0.

Related information:

List of XDR Programming References

eXternal Data Representation (XDR) Overview for Programming

Understanding XDR Library Filter Primitives

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

Item	Description
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.

Related information:

List of RPC Programming References

eXternal Data Representation (XDR) Overview for Programming

Remote Procedure Call (RPC) Overview for Programming

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

Item	Description
xdrs	Points to the eXternal Data Representation (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.

Related information:

List of RPC Programming References

eXternal Data Representation (XDR) Overview for Programming

Remote Procedure Call (RPC) Overview for Programming

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 language 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

Item	Description
xdrs	Points to the eXternal Data Representation (XDR) stream handle.
cp	Points to the character.

Related information:

List of XDR Programming References

eXternal Data Representation (XDR) Overview for Programming

Understanding XDR Library Filter Primitives

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.

Parameters

Item	Description
xdrs	Points to the XDR stream handle.

Related information:
List of XDR Programming References
eXternal Data Representation (XDR) Overview for Programming
Understanding XDR Non-Filter Primitives

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

Item	Description
xdrs	Points to the eXternal Data Representation (XDR) stream handle.
ep	Specifies the address of the enumeration data.

Return Values

Upon successful completion, this subroutine returns a value of 1. If unsuccessful, it returns a value of 0.

Related information:
List of XDR Programming References
eXternal Data Representation (XDR) Overview for Programming
Understanding XDR Library Filter Primitives

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 language floats (normalized single-precision floating-point numbers) and their external representations.

Parameters

Item	Description
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.

Related information:

List of XDR Programming References

eXternal Data Representation (XDR) Overview for Programming

Understanding XDR Library Filter Primitives

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 *proc* parameter specifies the eXternal Data Representation (XDR) routine for the object being freed. The *objp* parameter is a pointer to the object itself.

Note: The pointer passed to this routine is *not* freed, but the object it points to *is* freed (recursively).

Parameters

Item	Description
proc	Points to the XDR stream handle.
objp	Points to the object being freed.

Related information:

List of XDR Programming References

eXternal Data Representation (XDR) Overview for Programming

Understanding XDR Non-Filter Primitives

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.

Parameters

Item	Description
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, it returns a value of -1, even though the value has no meaning.

Related reference:

"xdr_setpos Macro" on page 122

Related information:

List of XDR Programming References

eXternal Data Representation (XDR) Overview for Programming

Understanding XDR Non-Filter Primitives

xdr_hyper Subroutine

Purpose

Translates long integers from C language to their external representations.

Library

C Library (**libc.a**)

Syntax

```
int xdr_hyper(XDR *xdrs, long long *lp)
```

Description

A filter primitive that translates ANSI C long integers to their external representations. This subroutine returns 1 if it succeeds, otherwise returns a value of 0.

Parameters

Item	Description
xdrs	Points to the eXternal Data Representation (XDR) stream handle.
ulp	Specifies the address of the long integer.

Return Values

Upon successful completion, the `xdr_hyper` subroutine returns a value of 1. If unsuccessful, it returns a value of 0.

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 subroutine associated with the eXternal Data Representation (XDR) stream pointed to by the *xdrs* parameter. The subroutine 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. The **xdr_inline** macro may return a value of null if it cannot return a buffer segment of the requested size.

Parameters

Item	Description
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.

Related information:

List of XDR Programming References

eXternal Data Representation (XDR) Overview for Programming

Understanding XDR Non-Filter Primitives

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

Item	Description
xdrs	Points to the eXternal Data Representation (XDR) stream handle.
ip	Specifies the address of the integer.

Return Values

Upon successful completion, this subroutine returns a value of 1. If unsuccessful, it returns a value of 0.

Related information:

List of XDR Programming References

eXternal Data Representation (XDR) Overview for Programming

Understanding XDR Library Filter Primitives

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

Item	Description
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 subroutine returns a value of 1. If unsuccessful, it returns a value of 0.

When in 64 BIT mode, if the value of the long integer can not be expressed in 32 BIT, **xdr_long** will return a value of 0.

Related information:

List of XDR Programming References

eXternal Data Representation (XDR) Overview for Programming

Understanding XDR Library Filter Primitives

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

Item	Description
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.

Related information:

List of XDR Programming References

eXternal Data Representation (XDR) Overview for Programming

Understanding XDR Library Filter Primitives

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

Item	Description
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.

Related information:

List of RPC Programming References

eXternal Data Representation (XDR) Overview for Programming

Remote Procedure Call (RPC) Overview for Programming

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

Item	Description
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.

Related information:

portmap subroutine

List of RPC Programming References

eXternal Data Representation (XDR) Overview for Programming

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

Item	Description
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.

Related information:

portmap subroutine

List of RPC Programming References

eXternal Data Representation (XDR) Overview for Programming

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

Item	Description
xdrs	Points to the eXternal Data Representation (XDR) stream handle.
objpp	Points to the character pointer of the data structure.
objsize	Specifies 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.

Related information:

List of XDR Programming References

eXternal Data Representation (XDR) Overview for Programming

Understanding XDR Non-Filter Primitives

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 any pointers within one structure that are referenced by another structure.

The **xdr_reference** subroutine does not attach special meaning to a null 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, when the pointer is null.

Parameters

Item	Description
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	Translates 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 subroutine returns a value of 1. If unsuccessful, it returns a value of 0.

Related information:

List of XDR Programming References

eXternal Data Representation (XDR) Overview for Programming

Understanding XDR Library Filter Primitives

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

Item	Description
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.

Related information:

List of RPC Programming References

eXternal Data Representation (XDR) Overview for Programming

Remote Procedure Call (RPC) Overview for Programming

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

Item	Description
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.

Related information:

List of RPC Programming References

eXternal Data Representation (XDR) Overview for Programming

Remote Procedure Call (RPC) Overview for Programming

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** macro. The **xdr_setpos** macro returns a value of false if the set position is not valid 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 macro to fail. This macro also fails if the programmer requests a position that is not in the stream's boundaries.

Parameters

Item	Description
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 macro returns a value of 1. If unsuccessful, it returns a value of 0.

Related reference:

"xdr_getpos Macro" on page 113

Related information:

List of XDR Programming References

eXternal Data Representation (XDR) Overview for Programming

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

Item	Description
xdrs	Points to the eXternal Data Representation (XDR) stream handle.
sp	Specifies the address of the short integer.

Return Values

Upon successful completion, this subroutine returns a value of 1. If unsuccessful, it returns a value of 0.

Related information:

List of XDR Programming References

eXternal Data Representation (XDR) Overview for Programming

Understanding XDR Library Filter Primitives

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

Item	Description
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 subroutine returns a value of 1. If unsuccessful, it returns a value of 0.

Related reference:

"xdr_wrapstring Subroutine" on page 130

Related information:

List of XDR Programming References

eXternal Data Representation (XDR) Overview for Programming

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

Item	Description
xdrs	Points to the eXternal Data Representation (XDR) stream handle.
ucp	Points to an unsigned integer.

Return Values

Upon successful completion, this subroutine returns a value of 1. If unsuccessful, it returns a value of 0.

Related information:

List of XDR Programming References

eXternal Data Representation (XDR) Overview for Programming

Understanding XDR Library Filter Primitives

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

Item	Description
xdrs	Points to the eXternal Data Representation (XDR) stream handle.
up	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.

Related information:

List of XDR Programming References

eXternal Data Representation (XDR) Overview for Programming

Understanding XDR Library Filter Primitives

xdr_u_long Subroutine

Purpose

Translates the unsigned long integers from the C language to 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 the unsigned long integers from the C language to their external representations.

Note: The `xdr_u_long` subroutine encodes or decodes a 32-bit value, irrespective of whether the application is compiled in 32-bit mode or in 64-bit mode. If a 64-bit value is passed to the `xdr_u_long` subroutine, the resulting high-order 32-bit values are not determined.

Parameters

Item	Description
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.

Related information:

List of XDR Programming References

eXternal Data Representation (XDR) Overview for Programming

Understanding XDR Library Filter Primitives

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

Item	Description
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 subroutine returns a value of 1. If unsuccessful, it returns a value of 0.

Related information:

List of XDR Programming References

eXternal Data Representation (XDR) Overview for Programming

Understanding XDR Library Filter Primitives

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);
```

Description

The **xdr_union** subroutine is a filter primitive that translates between discriminated C unions and their 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, this subroutine translates the union located at the address pointed to by the *unp* parameter.

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 specified process is called to translate the union. The end of the **xdr_discrim** structure array is denoted by a routine having a null value. If the discriminant is not found in the choices array, then the *defaultarm* structure is called (if it is not null).

Parameters

Item	Description
xdrs	Points to the eXternal Data Representation (XDR) stream handle.
dscmp	Specifies the address of the union's discriminant. The discriminant is an enumeration (**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. This parameter can have a null value.

Return Values

Upon successful completion, this subroutine returns a value of 1. If unsuccessful, it returns a value of 0.

Related information:

List of XDR Programming References

eXternal Data Representation (XDR) Overview for Programming

Understanding XDR Library Filter Primitives

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

Item	Description
xdrs	Points to the eXternal Data Representation (XDR) stream handle.
arrp	Specifies 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.

Related information:

List of XDR Programming References

eXternal Data Representation (XDR) Overview for Programming

Understanding XDR Library Filter Primitives

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 is passed to other Remote Procedure Call (RPC) subroutines that require a function parameter, but does not transmit data.

Return Values

This subroutine always returns a value of 1.

Related information:

List of XDR Programming References
eXternal Data Representation (XDR) Overview for Programming
Understanding XDR Library Filter Primitives

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) subroutines as parameters, and the **xdr_string** subroutine requires three.

Parameters

Item	Description
xdrs	Points to the XDR stream handle.
sp	Specifies the address of the pointer to the string.

Return Values

Upon successful completion, this subroutine returns a value of 1. If unsuccessful, it returns a value of 0.

Related reference:
"xdr_string Subroutine" on page 124
Related information:
List of XDR Programming References
eXternal Data Representation (XDR) Overview for Programming

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

Item	Description
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.

Related information:

List of RPC Programming References

eXternal Data Representation (XDR) Overview for Programming

Remote Procedure Call (RPC) Overview for Programming

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 language double-precision numbers and their external representations.

Parameters

Item	Description
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.

Related information:

List of XDR Programming References

eXternal Data Representation (XDR) Overview for Programming

Understanding XDR Library Filter Primitives

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

Item	Description
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**.

Related information:

List of XDR Programming References

eXternal Data Representation (XDR) Overview for Programming

Understanding XDR Non-Filter Primitives

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. As a result, additional bytes are in the stream to provide record boundary information.

Parameters

Item	Description
xdrs	Points to the XDR stream handle.
sendsize	Sets the size of the input buffer to which data is written. If 0 is specified, the buffers are set to the system defaults.
recvsize	Sets the size of the output buffer from which data is read. 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.

Related reference:

"xdrrec_endofrecord Subroutine" on page 134

"xdrrec_eof Subroutine" on page 134

"xdrrec_skiprecord Subroutine" on page 135

Related information:

List of XDR Programming References

eXternal Data Representation (XDR) Overview for Programming

Understanding XDR Non-Filter Primitives

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. If the value of the *sendnow* parameter is nonzero, the data in the output buffer is marked as a completed record and the output buffer is optionally written out.

Parameters

Item	Description
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 subroutine returns a value of 1. If unsuccessful, it returns a value of 0.

Related reference:

"xdrrec_create Subroutine" on page 133

Related information:

List of XDR Programming References

Understanding XDR Non-Filter Primitives

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 the stream reached the end of the file. This subroutine can only be invoked on streams created by the **xdrrec_create** subroutine.

Parameters

Item	Description
xdrs	Points to the eXternal Data Representation (XDR) stream handle.

Return Values

After consuming the rest of the current record in the stream, this subroutine returns a value of 1 if the stream has no more input, and a value of 0 otherwise.

Related reference:

"xdrrec_create Subroutine" on page 133

Related information:

List of XDR Programming References

eXternal Data Representation (XDR) Overview for Programming

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.

Parameters

Item	Description
xdrs	Points to the XDR stream handle.

Return Values

Upon successful completion, this subroutine returns a value of 1. If unsuccessful, it returns a value of 0.

Related reference:

"xdrrec_create Subroutine" on page 133

Related information:

List of XDR Programming References

eXternal Data Representation (XDR) Overview for Programming

Understanding XDR Non-Filter Primitives

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 an XDR stream calls the **fflush** function on the *file* stream, but never calls the **fclose** function.

Parameters

Item	Description
xdrs	Points to the XDR stream handle to initialize.
file	Points to the standard I/O device that data is written to or read from.
op	Specifies an XDR direction. The possible choices are **XDR_ENCODE**, **XDR_DECODE**, or **XDR_FREE**.

Related information:

List of XDR Programming References

eXternal Data Representation (XDR) Overview for Programming

Understanding XDR Non-Filter Primitives

AIX 3270 Host Connection Program (HCON)

This topic collection includes function that perform on the host application.

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 or 0 to occur.

The **cfxfer** function is part of the Host Connection Program (HCON).

C Parameters

Item	Description
sxfer	Specifies an **fxs** structure as defined in the **fxfer.h** file. The **fxs** C structure is:

```
struct fxs  {
    int     fxs_bytcnt;
    char   *fxs_src;
    char   *fxs_dst;
    char   *fxs_ctime;
    int     fxs_stat;
    int     fxs_errno;
}
```

Pascal Parameters

Item	Description
Sfxfer	Specifies a record of type **fxs** as defined within the **fxfer.inc** file. The Pascal **fxs** record format is:

```
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

Item	Description
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

Item	Description
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 Values

The **cfxfer** function returns the following:

Value	Description
0	Ready status-success. The structure member *fxs.fxs_stat* contains status of **fxfer** function.
-1	Error status. Failure of **cfxfer** function. The **fxs** structure has NOT been set.
1	Status is not yet available.

The **fx_stat***xxxxxx* 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.

Files

Item	Description
$HOME/fx_stat*xxxxxx*	Temporary file used for status

Related reference:

"fxfer Function"

"g32_fxfer Function" on page 148

Related information:

fxfer subroutine

fxfer Function

Purpose

Initiates a file transfer from within a program.

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*, *INPUTFLD*, *CODESET*, *SESSIONNAME*

INT *FLAGS*, *RECL*, *BLKSIZE*, *SPACE*, *INCR*, *UNIT*, *RC*

RC = FFXFER (*SRCF*, *DSTF*, *LOGID*, *FLAGS*, *RECL*, *BLKSIZE*,

+ *SPACE*, *INCR*, *UNIT*, *INPUTFLD*, *CODESET*, *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 **fxfer** function is part of the Host Connection Program (HCON).

The **fxfer** function requires one or more adapters used to connect to a host.

This function requires one of the following operating system environments be installed on the mainframe host: VM/SP CMS, VM/XA CMS, MVS/SP TSO/E, MVS/XA, TSO/E, CICS/VS, VSE/ESA, or VSE/SP.

This function requires that the System/370 Host-Supported File Transfer Program (**IND$FILE** or its equivalent) be installed on the mainframe host.

The file names are character strings. The local-system file names must be in operating system format. The host file names must conform to the host naming convention, which must be one of the following formats:

Format	Description
VM/CMS	*FileName FileType FileMode*
MVS/TSO	*DataSetName* [(*MemberName*)][**/***Password*]
CICS/VS	*FileName* (up to 8 characters)
VSE/ESA	*FileName* (up to 8 characters) **Note:** The VSE host is not supported in a double-byte character set (DBCS) environment.

C Parameters

Item	Description
xfer	Specifies a pointer to the **fxc** structure defined in the **fxfer.h** file.
sessionname	Points to the name of a session. The session profile for that session specifies the host connectivity to be used by the file transfer programming interface. The session name is a single character in the range of a through z. Capital letters are interpreted as lowercase letters. Session variables are defined in a HCON session profile. If the value of the *sessionname* parameter is set to a null value, the **fxfer** function assumes you are running in an **e789** subshell.

Pascal Parameters

Item	Description
xfer	Specifies a record of **fxc** type within the **fxfer.inc** file.
sessionname	Points to the name of a session. The session profile indicated by the *sessionname* parameter 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 through z. Capital letters are interpreted as lowercase letters. Session variables are defined in an HCON session profile. If the *sessionname* parameter is set to `char(0)`, the **pfxfer** function assumes you are running in an **e789** subshell.

FORTRAN Parameters

Item	Description
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 host logon ID.
SESSIONNAME	Points to the name of a session. The *SESSIONNAME* parameter names a session profile that 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 through z. Capital letters are interpreted as lowercase letters. Session variables are defined in a HCON session profile. If the *SESSIONNAME* parameter 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: **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 **4096** Host system CICS/VS **8192** Host system VSE/ESA
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: **-1** Specifies the number of TRACKS. **-2** Specifies the number of CYLINDERS. A positive number indicates the number of bytes to allocate.
INPUTFLD	Specifies the host input table field.

Item	Description
CODESET	Specifies an alternate code set to use for ASCII to EBCDIC and EBCDIC to ASCII translations: **CHAR(0)** Uses current operating-system ASCII code page. **IBM®-932** Uses IBM code page 932 for translation in a DBCS environment. **ISO8859-1** Uses ISO 8859-1 Latin alphabet number 1 code page. **ISO8859-7** Uses ISO 8859-7 Greek alphabet. **ISO8859-9** Uses ISO 8859-9 Turkish alphabet. **IBM-eucJP** Uses IBM Extended UNIX code for translation in the Japanese Language environment. **IBM-eucKR** Translates Korean language. **IBM-eucTW** Translates traditional Chinese language.

Note:

1. All FORTRAN character array strings must be terminated by a null character, as in the following example:
   ```
   SRCF = 'rtfile'//CHAR(0)
   ```
2. The VSE host system is not supported in a DBCS environment.
3. The unique DBCS file-transfer flags are not supported by this function.

Return Values

If the **fxfer** function is called synchronously, it returns a value of 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 0. 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_stat***xxxxxx* status file, the **cfxfer** function returns a -1. If the status is not ready, the **cfxfer** function returns a -2.

The **fx_stat***xxxxxx* 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.

Related reference:

"cfxfer Function" on page 137

g32_alloc Function

Purpose

Initiates interaction with a host application.

Libraries

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 3270 operatorless interface.

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

The **g32_alloc** function is part of the Host Connection Program (HCON).

The **g32_alloc** function requires one or more adapters used to connect to a host.

CICS® and VSE do not support **API/API** or **API/API_T** modes.

C Parameters

Item	Description
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 to be executed. This string should be the entire string necessary to start the application, including any necessary parameters or options. When specifying an *applname* parameter, place the host application name in double quotes ("Testload") or specify a pointer to a character string.
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 lets local system applications act like a 3270 operatorless interface. Applications in this mode use the 3270 operator less interface to communicate with the host application. In API/3270 mode, if the value of the *applname* parameter is a null pointer, no host application is started. **MODE_API** The API/API mode is a private protocol 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. The API program must use HCON's API and must have a corresponding host API program that uses HCON's host API for the programs to communicate. **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. The keyboard is locked. **MODE_API_T** The API_T mode is the same as the **MODE_API** type 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 tables used are determined by the language characteristic 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

Item	Description
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.
mode	Specifies the mode desired for the session.

FORTRAN Parameters

Item	Description
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

Item	Description
0	Indicates successful completion.
-1	Indicates an error has occurred. • The `errcode` field in the **g32_api** structure is set to an error code identifying the error. • The `xerrinfo` field can be set to give more information about the error.

Examples

The following example illustrates the use of the **g32_alloc** function in C language:

```
#include <g32_api.h>             /* API include file         */
main ()
{
struct g32_api *as, asx;         /* API status       */
int session_mode = MODE_API      /* api session mode. Other
                                    modes are MODE_API_T
                                    and MODE_3270    */
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);
.
.
.
```

Related reference:

"G32ALLOC Function" on page 182

g32_close Function

Purpose

Detaches from a session.

Libraries

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 disconnects from a 3270 session. If the **g32_open** or **g32_openx** function created a session, the **g32_close** function logs off from the host and terminates the session. A session must be terminated (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. Application programs using the FORTRAN language for the HCON API must include and link both the C and FORTRAN libraries.

The **g32_close** function is part of the Host Connection Program (HCON).

The **g32_close** function requires one or more adapters used to connect to a host.

C Parameters

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

Pascal Parameters

Item	Description
as	Specifies a **g32_api** structure.

FORTRAN Parameters

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

Return Values

Item	Description
0	Indicates successful completion.
-1	Indicates an error has occurred. • The `errcode` field in the **g32_api** structure is set to an error code identifying the error. • The `xerrinfo` field can be set to give more information about the error.

Examples

The following example fragment illustrates the use of the **g32_close** function in C language:

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

g32_dealloc Function

Purpose

Ends interaction with a host application.

Libraries

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 operating system 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. Application programs using the FORTRAN language for the HCON API must include and link both the C and FORTRAN libraries.

The **g32_dealloc** function is part of the Host Connection Program (HCON).

The **g32_dealloc** function requires one or more adapters used to connect to a host.

C Parameters

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

Pascal Parameters

Item	Description
as	Specifies the **g32_api** structure.

FORTRAN Parameters

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

Return Values

Item	Description
0	Indicates successful completion.
-1	Indicates an error has occurred. • The errcode field in the **g32_api** structure is set to an error code identifying the error. • The xerrinfo field can be set to give more information about the error.

Examples

The following example illustrates the use of the **g32_dealloc** function in C language:

```
#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);
.
.
.
```

g32_fxfer Function

Purpose

Invokes a file transfer.

Libraries

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, INPUTFLD, CODESET
INTEGER FLAGS,RECL,BLKSIZE,SPACE,INCR,UNIT

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

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 in a programmatic fashion, 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 detach from the session without terminating it, run the specified file transfer, and then reattach to the session.

If a **g32_alloc** function has been issued before invoking the **g32_fxfer** command, be sure that the corresponding **g32_dealloc** function 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. Application programs using the FORTRAN language for the HCON API must include and link both the C and FORTRAN libraries.

The **g32_fxfer** function is part of the Host Connection Program (HCON).

The **g32_fxfer** function requires one or more adapters used to connect to a host.

This function requires that the Host-Supported File Transfer Program (**IND$FILE** or its equivalent) be installed on the host.

C Parameters

Item	Description
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

Item	Description
as	Specifies a record of type **g32_api**.
xfer	Specifies a record of type **fxc** within the **fxfer.inc** file.

FORTRAN Parameters

Item	Description
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
	4096 Host System CICS/VS
	8192 Host System VSE/ESA
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 Number of TRACKS
	-2 Number of CYLINDERS.
	A positive number indicates the number of blocks to be allocated.
INPUTFLD	Specifies the host input table field.

Item	Description
CODESET	Specifies an alternate code set to use for ASCII to EBCDIC and EBCDIC to ASCII translations. The following code sets are supported: **CHAR(0)** Uses current operating system ASCII code page. **IBM850** Uses IBM code page 850 for translation in a single byte code set (SBCS) environment. **IBM932** Uses IBM code page 932 for translation in a double byte code set (DBCS) environment. **ISO8859-1** Uses ISO 8859-1 Latin alphabet number 1 code page. **ISO8859-7** Uses ISO 8859-7 Greek alphabet. **ISO8859-9** Uses ISO 8859-9 Turkish alphabet. **IBMeucJP** Uses IBM Extended UNIX Code for translation in the Japanese Language environment. **IBMeucKR** Korean language. **IBMeucTW** Traditional Chinese language.

Note:

1. All FORTRAN character array strings must be null-terminated. For
2. example:

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

3. The Host System VSE is not supported in the DBCS environment.
4. The unique DBCS file transfer flags are not supported by this function.

Return Values

Item	Description
0	Indicates successful completion. The user may call the **cfxfer** function to get the status of the file transfer.
1	Indicates the file transfer did not complete successfully. The user may call the **cfxfer** function to get the status of the file transfer.
-1	Indicates the **g32_fxfer** command failed while accessing the link. The `errcode` field in the **g32_api** structure is set to an error code identifying the error. The `xerrinfo` field can be set to give more information about the error.

Examples

The following example fragment illustrates the use of the **g32_fxfer** function in an **api_3270** mode program in C language:

```
#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_opts.f_inputfld = (char *)0;   /* Set Input Field
                                               to NULL        */
   xfer->fxc_opts.f_aix_codepg = (char *)0; /* Set Alternate
                                               Codepg to NULL */
   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);
}
```

The following example fragment illustrates the use of the **g32_fxfer** function in an **api_3270** mode program in Pascal language.

```
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.
```

The following example fragment illustrates the use of the **g32_fxfer** function in an **api_3270** mode program in FORTRAN language:

```
 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*64   INPUTFLD
 CHARACTER*8    CODESET
 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 = '/home/test.case1'//CHAR(0)
C   Source and Destination files for the fcfxfer status
C   check command
   SRC = CHAR(0)
   DST = CHAR(0)
C   Set Input Field to NULL
   INPUTFLD = CHAR(0)
C   Set Alternate AIX codeset to NULL
   CODESET = CHAR(0)
C   Set the G32FXFER file transfer flags and options
C   Take the defaults for Logical Record Length, Block Size,
C   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,INPUTFLD,CODESET)
   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
```

Related reference:
"cfxfer Function" on page 137

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.

Libraries

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.

The **g32_get_cursor** function is part of the Host Connection Program (HCON).

The **g32_get_cursor** function requires one or more adapters used to connect to a host.

C Parameters

Item	Description
as	Specifies a pointer to the **g32_api** structure. This structure contains the row (**row**) and column (**column**) address of the cursor. Status information is also set in this structure.

Pascal Parameters

Item	Description
as	Specifies the **g32_api** structure.

FORTRAN Parameters

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

Return Values

Item	Description
0	Indicates successful completion. • The corresponding **row** element of the *as* structure is the row position of the cursor. • The corresponding **column** element of the *as* structure is the column position of the cursor.
-1	Indicates an error has occurred. • The `errcode` field in the **g32_api** structure is set to the error code identifying the error. • The `xerrinfo` field can be set to give more information about the error.

Examples

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

The following example fragment illustrates, in C language, the use of the **g32_get_cursor** function in an api_3270 mode program:

```
#include <g32_api.h>      /* API include file */
#include  <g32_keys.h>
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 position is 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 */
return = g32_send_keys(as,CLEAR);
/* Turn notification off */
```

```
return = g32_notify(as,0);
.
.
.
```

g32_get_data Function

Purpose

Obtains current specified display data from the presentation space.

Libraries

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. The transfer continues until either the transfer length is exhausted or the starting point is reached. If the transfer length is greater than the presentation space, then the **g32_get_data** function only reads data that equals one presentation space and leaves the rest of the buffer unchanged.

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.

The **g32_get_data** function is part of the Host Connection Program (HCON).

The **g32_get_data** function requires one or more adapters used to connect to a host.

In a double-byte character set (DBCS) environment, the **g32_get_data** function only obtains SBCS data from the presentation space even if Kanji or Katakana characters are displayed on the screen. The DBCS data are not available.

C Parameters

Item	Description
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

Item	Description
as	Specifies the **g32_api** structure.
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. **Note:** The address of a packed array can be obtained by using the **addr()** system call: `buffer := addr (<message array name> [1]).`

FORTRAN Parameters

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

Item	Description
0	Indicates successful completion.
-1	Indicates an error has occurred. • The `errcode` field in the **g32_api** structure is set to the error code identifying the error. • The `xerrinfo` field can be set to give more information about the error.

Examples

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

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 */
#include  <g32_keys.h>
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 position is 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 */
return = g32_send_keys(as,CLEAR);
/* Turn notification off */
return = g32_notify(as,0);
.
.
.
```

g32_get_status Function

Purpose

Returns status information of the logical path.

Libraries

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 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.

The **g32_get_status** function can only be used in API/API, API/API_T, and API/3270 modes.

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

The **g32_get_status** function is part of the Host Connection Program (HCON).

The **g32_get_status** function requires one or more adapters used to connect to a host.

C Parameters

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

Pascal Parameters

Item	Description
as	Specifies the **g32_api** structure.

FORTRAN Parameters

Item	Description
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

Item	Description
0	Indicates successful completion.

Error Codes

The values of `errcode` are as follows:

Error Code	Description
G32_NO_ERROR	0, indicates no error has occurred.
G32_COMM_CHK	-1, indicates a communications check has occurred.
G32_PROG_CHK	-2, indicates a program check has occurred within the emulator.
G32_MACH_CHK	-3, indicates a machine check has occurred.
G32_FATAL_ERROR	**-4,** indicates a fatal error has occurred within the emulator.
G32_COMM_REM	-5, indicates a communications check reminder has occurred.

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

Value	Description
-1	Indicates an error has occurred. • The `errcode` field in the **g32_api** structure is set to the error code identifying the error. • The `xerrinfo` field can be set to give more information about the error.

Examples

The following example fragment illustrates the use of the **g32_get_status** function in C language:

```
#include <g32_api.h>      /* API include file */
main()
{
struct g32_api *as;      /* g32 structure */
int return;
return = g32_write(as, mssg, length);
```

```
           /* see if unsuccessful */
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

g32_notify Function

Purpose

Turns data notification on or off.

Libraries

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 or off. The **g32_notify** subroutine may be used only by applications in an 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 whose 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. Application programs using the FORTRAN language for the HCON API must include and link both the C and FORTRAN libraries.

The **g32_notify** function is part of the Host Connection Program (HCON).

The **g32_notify** function requires one or more adapters used to connect to a host.

C Parameters

Item	Description
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

Item	Description
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

Item	Description
AS	Specifies a **g32_api** equivalent structure as an array of integers.
Note	Specifies to turn notification off (if the *Note* parameter is zero) or on (if the *Note* parameter is nonzero).

Return Values

Item	Description
0	Indicates successful completion.
-1	Indicates an error has occurred. • The `errcode` field in the **g32_api** structure is set to the error code identifying the error. • The `xerrinfo` field can be set to give more information about the error.

Examples

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

The example fragment illustrates, in C language, the use of the **g32_notify** function in an **api_3270** mode program:

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/poll.h>
#include <sys/msg.h>
#include "g32_api.h"

*****************************************************************
Note that the following function is an example of g32_notify function use.
It is meant to be called from an API application program that has already
performed a g32_open()or g32_openx() and a g32_alloc() function call. The
function will accept the as structure, a search pattern, and a timeout
(in seconds) as arguments. The purpose for calling this function is to
search  for a certain pattern on the "screen" within a given amount of
time. As soon as the host updates the screen (presentation space),the
notification is sent (the poll returns with a success). This data may
not be your desired pattern, so this routine will retry until the timeout
```

```
is reached. The function will poll on the message queue and search the
presentation space each time the API is notified. If the pattern is found,
a success is returned. If the pattern is not found in the specified timeout
period, a failure (-1) is returned. The application should pass the timeout
value in seconds.
*****************************************************************/
search_pres_space (as,pattern,timeout)
  struct g32_api *as;          /* Pointer to api structure */
  char *pattern;               /* Pattern to search for in
                                  presentation space */
  int timeout;                 /* The maximum time to wait before
                                  returning a failure */
{
  char done=0;                 /* Flag used to test if loop is
                                  finished */
  int rc;                      /* return code */
  long smsg;                   /* message buffer */
  unsigned long nfdmsgs;       /* Specified number of file
                                  descriptors and number of
                                  message queues to check. Low
                                  order 16 bits is the number of
                                  elements in array of pollfd.
                                  High order 16 bits is number of
                                  elements in array of pollmsg.*/
  struct pollmsg msglstptr;    /* structure defined in poll.h
                                  contains message queue id,
                                  requested events, and returned
                                  events */
  timeout *= 1000              /* convert to milliseconds for
                                  poll call */
  g32_notify (as, 1);          /* turn on the notify */
  rc = g32_search(as,pattern); /* search the presentation space
                                  for the pattern */
  if (rc == 0) {
     done = 1;
  }
  /*Loop while the pattern not found and the timeout has not been
     reached */
  /* Note that this is done in 500 ms. increments */
  while ( !(done) && (timeout > 0) ) {
    /* wait a max of 500 ms for a response from the host */
    /* This is done via the poll system call             */
    nfdmsgs = (1<<16);           /* One element in the msglstptr
                                    array. Since the low order
                                    bits are zero, they will be
                                    ignored by the poll */
    msglstptr.msgid = as->eventf; /* The message queue id */
    msglstptr.reqevents = POLLIN; /*Set flag to check if input is
                                     present on message queue */
      /* poll on the message queue. A return code of 1 signifies
         data from the host. An rc of 0 signifies a timeout. An
         rc < 0 signifies an error */
      rc = poll (&msglstptr,nfdmsgs,(long)500);
      rc = rc >> 16;             /* shift return code into low
                                    order bits */
      /* If the poll found something, do another search */
       if (rc = 1) {
         /* call msgrcv system call, retrying until success */
         /* This is done to flush the IPC queue */
         do {
            rc = msgrcv(as->eventf,(struct msgbuf *)&smsg,

                                (size_t)0,(long)1,IPC_NOWAIT|IPC_NOERROR);
         }
         while ( rc == G32ERROR);
```

```
        rc = g32_search (as,pattern);  /* Search for pattern */
        /* if pattern is found, set done flag to exit loop */
        if (rc == 0) {
            done = 1;
         }
      }
      timeout -= 500;   /* decrement the timeout by 500ms */
    } /* end while */
    g32_notify (as,0);  /* turn the notify off again */
    if (done) {
      return (0);       /* search was successful */
    }
    else {
      return (-1);      /* failure */
      }
}
```

g32_open Function

Purpose

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

Libraries

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 automatically. The user is logged on to the host if requested. 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 automatic login is performed.

The **g32_open** function can be nested for multiple opens as long as a distinct *as* structure is created and passed to each open. Corresponding API functions will map to each open session according to the *as* structure passed to each.

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

The **g32_open** function is part of the Host Connection Program (HCON).

The **g32_open** function requires one or more adapters used to connect to a host.

CICS/VS and VSE/ESA do not support **API/API** or **API/API_T** modes.

C Parameters

Item	Description
as	Specifies a pointer to the **g32_api** structure. Status is returned in this structure.
flag	Signals whether the login procedure should be performed. Flag values are as follows: • If the emulator is running and the user is logged in to the host, the value of the *flag* parameter must be 0. • If the emulator is running, the user is not logged in to the host, and the API logs in to the host, the value of the *flag* parameter must be set to 1. • If the emulator is not running and the API application executes an automatic login/logoff procedure, the value of the *flag* parameter is ignored.
uid	Specifies a pointer to the login ID string if the **g32_open** function logs in to the host. If the login ID is a null string, the login procedure prompts the user for both the login 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 login 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 login 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 login 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 through z. Capital letters are interpreted as lowercase letters.

Pascal Parameters

Item	Description
as	Specifies the **g32_api** structure.
flag	Signals whether the login procedure should be performed. • If the emulator is running, the user is logged in to the host, and the API application executes as a subshell of the emulator, the value of the *flag* parameter must be 0. • If the emulator is running, the user is not logged in to the host, and the API application executes as a subshell of the emulator and the application is to perform an automatic login/logoff procedure, the value of the *flag* parameter must be set to 1. • If the emulator is not running and the API application executes an automatic login/logoff procedure, the value of the *flag* parameter is ignored.
uid	Specifies a pointer to the login ID string. If the user ID is a null string, the login 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 login ID string. If it points to a null string, the login 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 through 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)`.

Parameter	Description
AS	Specifies the **g32_api** equivalent structure as an array of integers.
FLAG	Signals whether the login procedure should be performed.
UID	Specifies a pointer to the login ID string. If the user ID is a null string, the login 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 login ID string. If the parameter specifies a null string, the login 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 through z. Capital letters are interpreted as lowercase letters.

Return Values

Upon successful completion:

- A value of 0 is returned.
- The `lpid` field in the **g32_api** structure is set to the session ID.

Upon unsuccessful completion:

- A value of -1 is returned.
- The `errcode` field in the **g32_api** structure is set to an error code identifying the error.
- The `xerrinfo` field can be set to give more information about the error.

Examples

The following example fragment illustrates the use of the **g32_open** function in an **api_3270** mode program in C language:

```
#include <g32_api.h>
main()
{
   struct g32_api *as, asx;  /* asx is statically
```

```
                declared*/
   int flag=0;
   int ret;
   as = &asx;      /* as points to an
               allocated structure */
   ret=g32_open(as,flag,"mike","mypassword","a");
   .
   .
   .
}
```

The following example fragment illustrates the use of the **g32_open** function in an **api_3270** mode program in Pascal language:

```
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.
```

The following example fragment illustrates the use of the **g32_open** function in an **api_3270** mode program in FORTRAN language:

```
INTEGER G32OPEN
INTEGER RC, AS(9), FLAG
CHARACTER*20 UID
CHARACTER*10 PW
CHARACTER*2 SN
EXTERNAL G32OPEN
UID = CHAR(0)
PW = CHAR(0)
SN = 'a'//CHAR(0)
FLAG = 0
RC = G32OPEN(AS, FLAG, UID, PW, SN)
   .
   .
   .
```

g32_openx Function

Purpose

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

Libraries

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 G32OPENX

CHARACTER* XX  UID,  PW,  SESSIONNAME

RC = G32OPENX (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 automatic login. The user is logged in 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 automatically, the function performs an automatic login.

The **g32_openx** function can be nested for multiple opens as long as a distinct *as* structure is created and passed to each open. Corresponding API functions will map to each open session according to the *as* structure passed to each.

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.

The **g32_openx** function is part of the Host Connection Program (HCON).

The **g32_openx** function requires one or more adapters used to connect to a host.

CICS and VSE do not support **API/API** or **API/API_T** modes.

C Parameters

The **g32_openx** function allows for a varying number of parameters after the *flag* parameter. The *as* and *flag* parameters are required; the *uid*, *pw*, *session*, and *timeout* parameters are optional.

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 (like the **exec** function). Providing an integer of 1 forces a default on a parameter. Use the default to provide a placeholder for optional parameters that you do not need to supply.

Parameter	Description
as	Specifies a pointer to the **g32_api** structure.
flag	Requires one of the following: • Set the *flag* parameter to 0, if the emulator is running and the user is logged on to host. • Set the *flag* parameter to 1 if the emulator is running, the user is not logged on to host, and the API application is to perform the login/logoff procedure. The **g32_openx** function ignores the *flag* parameter, if the emulator is not running and the API application executes an automatic login/logoff procedure.
uid	Specifies a pointer to the login ID string. If the login ID is a null string, the login procedure prompts the user for both the login 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 login ID is a string consisting of the host user ID and an optional list of additional variables separated by commas, as shown in the example: `userid,var1,var2,...` In this example, *var1* is the login script name (when using AUTOLOG) and *var2* is the optional trace and time values. The list is passed to the automatic procedure.
pw	Specifies a pointer to the password string associated with the login 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 automatically, the login 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 through 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 that specifies the amount of nonactive time in seconds allowed to occur between the workstation and the host operations (that is, **g32_read** and **G32WRITE**). This parameter is optional. If no value is provided in the calling statement, the default value is 15. 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 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.

Parameter	Description
as	Specifies the **g32_api** structure.
flag	Signals whether the login procedure should be performed: • Set the *flag* parameter to 0. If the emulator is running, the user is logged on to host. • Set the *flag* parameter to 1. If the emulator is running, the user is not logged on to host, and the API application performs the login/logoff procedure. • If the emulator is not running and the API application executes an automatic login/logoff procedure, the value of *flag* is ignored.
uid	Specifies a pointer to the login ID string. If the login ID is a null string, the login procedure prompts the user for both the login 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 login 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 automatically, the login 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 through 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 that specifies the amount of nonactive time in seconds allowed to occur between the workstation and the host operations (that is, **g32_read** and **g32WRITE**). This parameter is optional. If no value is provided in the calling statement, the default value is 15. The minimum value allowed is 1. 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 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)`.

Parameter	Description
AS	Specifies the **g32_api** equivalent structure as an array of integers.
FLAG	Signals that the login procedure should be performed: • Set the *FLAG* parameter to 0, if the emulator is running, the user is logged on to host. • Set the *FLAG* parameter to 1, 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 automatic login/logoff procedure, the value of the *FLAG* parameter is ignored.
UID	Specifies a pointer to the login ID string. If the login ID is a null string, the login procedure prompts the user for both the login 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 login 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 automatically, the login 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 through z. Capital letters are interpreted as lowercase letters. Parameters for each session are specified in a per session profile.
TIMEOUT	Specifies a numerical string 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.

Return Values

Item	Description
0	Indicates successful completion. The `lpid` field in the **g32_api** structure is set to the session ID.
-1	Indicates an error has occurred. • The `errcode` field in the **g32_api** structure is set to an error code identifying the error. • The `xerrinfo` field can be set to give more information about the error.

Examples

1. To use the **g32_openx** function with fewer than four optional string constant parameters specified and with AUTOLOG, enter:

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

2. To use the **g32_openx** function with fewer than four optional string constant parameters specified and with the automatic login facility, enter:

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

3. To use the **g32_openx** function with all optional parameters not specified, enter:

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

   OR

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

4. To use the **g32_openx** function with four variable optional parameters, enter:

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

5. To use the **g32_openx** function with fewer than four variable optional parameters, enter:

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

6. To use the **g32_openx** function with two default optional parameters, enter:

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

7. To use the **g32_openx** function with a mixture:

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

8. To use the **g32_openx** function within a program segment in the C language:

   ```
   #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;

      sn = &nm;
      as = &asx;            /* as points to an allocated structure */
      ret=g32_openx(as,flag,"mike","mypassword","a","60");
      .
      .
      .
   }
   ```

   Note: Only the first two parameters are mandatory. The remaining parameters can be terminated with a `0`. For example:

   ```
   ret = g32_openx(as.flag,0);
   ```

   Null characters may be substituted for any of the string values if profile or command values are desired.

9. To use the **g32_openx** function within a program segment in the Pascal language:

```
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.
```

10. To use the **g32_openx** function within a program segment in the FORTRAN language:

```
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)
   .
   .
   .
```

g32_read Function

Purpose

Receives a message from a host application.

Libraries

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
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. Application programs using the FORTRAN language for the HCON API must include and link both the C and FORTRAN libraries.

The **g32_read** function is part of the Host Connection Program (HCON).

The **g32_read** function requires one or more adapters used to connect to a host.

In a DBCS environment, the **g32_read** function only reads SBCS data from a host in the **MODE_API_T** mode.

C Parameters

Item	Description
as	Specifies a pointer to a **g32_api** structure.
msgbuf	Specifies a pointer to a buffer where a message from the host is placed. The API obtains space for this buffer by using the **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 but less than or equal to the maximum I/O buffer size parameter specified in the HCON session profile.

Pascal Parameters

Item	Description
as	Specifies the **g32_api** structure.
buffer	Specifies a **stringptr** structure. The API obtains space for this buffer by using the **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

Item	Description
AS	Specifies the **g32_api** equivalent structure.
BUFLEN	Specifies the size, in bytes, of the value contained in the *MSGBUF* parameter. The message length must be greater than 0 and less than or equal to the maximum I/O buffer size parameter specified in the HCON session profile.
MSGBUF	Specifies the storage area for the character data read from the host.

Return Values

Item	Description
> 0 (greater than or equal to zero)	Indicates successful completion.
-1	Indicates an error has occurred. • The `errcode` field in the **g32_api** structure is set to the error code identifying the error. • The `xerrinfo` field can be set to give more information about the error.

Examples

The following example illustrates the use of the **g32_read** function in C language.

```
#include <g32_api>       /* API include file */
main()
{
struct g32_api *as, asx  /* 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 */
.
.
.
as = &asx;
msg_buff = &messg;       /* point to a string */
return = g32_read(as, msg_buff, &msg_len);
.
.
.
```

```
free (*msg_buff);
.
.
.
```

g32_search Function

Purpose

Searches for a character pattern in a presentation space.

Libraries

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. If you start at 0 for row and column, an invalid position error will result.

The **g32_search** function is part of the Host Connection Program (HCON).

The **g32_search** function requires one or more adapters used to connect to a host.

In a DBCS environment, the **g32_search** function only searches the presentation space for an SBCS character pattern. This function does not support Katakana or DBCS characters.

Pattern Matching

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

Character	Description
?	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.

Note: The pattern cannot 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. Application programs using the FORTRAN language for the HCON API must include and link both the C and FORTRAN libraries.

C Parameters

Item	Description
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

Item	Description
as	Specifies the **g32_api** structure.
pattern	Specifies a 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

Item	Description
AS	Specifies a **g32_api** equivalent structure as an array of integers.
PATTERN	Specifies a string that is searched for in the presentation space.

Return Values

Item	Description
0	Indicates successful completion. • The corresponding `row` field of the *as* structure is the row position of the beginning of the matched string. • The corresponding `column` field of the *as* structure is the column position of the beginning of the matched string. • The corresponding `length` field of the *as* structure is the length of the matched string.
-1	Indicates an error has occurred. • The `errcode` field in the **g32_api** structure is set to the error code identifying the error. • The `xerrinfo` field can be set to give more information about the error.

Examples

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

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

```
#include <g32_api.h>             /* API include file */
#include <g32_keys.h>
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 position is 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 */
return = g32_send_keys(as,CLEAR);
/* Turn notification off */
return = g32_notify(as,0);
.
.
.
```

g32_send_keys Function

Purpose

Sends key strokes to the terminal emulator.

Libraries

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.

Note: The **g32_send_keys** function can only send 128 characters per call. The **g32_send_keys** function can be chained when more than 128 characters must be sent.

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

The **g32_send_keys** function is part of the Host Connection Program (HCON).

The **g32_send_keys** function requires one or more adapters used to connect to a host.

In a DBCS environment, the **g32_send_keys** function only sends SBCS keystrokes, including ASCII characters, to a terminal emulator. DBCS characters are ignored.

C Parameters

Item	Description
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

Item	Description
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

Item	Description
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    ATTN
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
                        CURSEL
```

Return Values

Item	Description
0	Indicates successful completion.
-1	Indicates an error has occurred. • The `errcode` field in the **g32_api** structure is set to the error code identifying the error. • The `xerrinfo` field can be set to give more information about the error.

Examples

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

The following example fragment illustrates, in C language, the use of the **g32_send_keys** function in an **api_3270** mode program:

```
#include <g32_api.h>            /* API include file */
*include <g32_keys.h>
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 position is 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 */
return = g32_send_keys(as,CLEAR);
/* Turn notification off */
return = g32_notify(as,0);
.
.
.
```

g32_write Function

Purpose

Sends a message to a host application.

Libraries

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.

The **g32_write** function is part of the Host Connection Program (HCON).

The **g32_write** function requires one or more adapters used to connect to a host.

In a DBCS environment, the **g32_write** function only sends SBCS data to a host in the MODE_API_T mode.

C Parameters

Item	Description
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

Item	Description
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]).
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

Item	Description
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

Item	Description
> 0 (greater than or equal to zero)	Indicates successful completion.
-1	Indicates an error has occurred. • The `errcode` field in the **g32_api** structure is set to the error code identifying the error. • The `xerrinfo` field can be set to give more information about the error.

Examples

The following example illustrates, in C language, 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);
.
.
.
```

G32ALLOC Function

Purpose

Starts interaction with an API application running simultaneously on the local system.

Syntax

G32ALLOC

Description

The **G32ALLOC** function starts a session with an application program interface (API) application by sending a message to the **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 application program.

The **G32ALLOC** function is part of the Host Connection Program (HCON).

The **G32ALLOC** function requires one or more adapters used to connect to a mainframe host.

Return Values

This call sets register 0 to the following values:

Value	Description
> 0	Indicates a normal return or a successful call. The value returned indicates the maximum number of bytes that may be transferred to an operating system application by way of **G32WRITE** or received from an operating systems application by way of **G32READ**.
< 0	Indicates less than 0. Host API error condition.

Examples

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
.
.
.
```

Related reference:

"g32_alloc Function" on page 142

"G32DLLOC Function"

G32DLLOC Function

Purpose

Terminates interaction with an API application running simultaneously on the local system.

Syntax

G32DLLOC

Description

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

The **G32DLLOC** function requires one or more adapters used to connect to a mainframe host.

Return Values

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

Value	Description
0	Indicates a normal return or a successful call.
< 0	Indicates less than zero. An error condition exists.

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.        */
.
.
.
```

Related reference:

"G32ALLOC Function" on page 182

"G32READ Function"

G32READ Function

Purpose

Receives a message from the API application running simultaneously on the local system.

Syntax

G32READ

Description

The **G32READ** function receives a message from an application programming interface (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 information:

Return	Description
R0	Indicates the number of bytes read.
R1	Indicates the address of the message buffer.

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:

The **G32READ** function is part of the Host Connection Program (HCON).

The **G32READ** function requires one or more adapters used to connect to a mainframe host.

```
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.

> **Attention:** 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 the API/3270 mode to exit. You will not be able to get the API/API mode back until you send the Enter key.

Return Values

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

Value	Description
> 0	Normal return. Indicates the length of the message as the number of bytes read.
< 0	Less than zero. Indicates a host API error condition.

Examples

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
           .
           .
           .
```

Related reference:
"G32DLLOC Function" on page 183

G32WRITE Function

Purpose

Sends a message to an API application running simultaneously on the local system.

Syntax

G32WRITE *MSG*, *LEN*

Description

The **G32WRITE** function sends a message to an API application. The maximum number of bytes that may be transferred is specified by the value returned in register zero (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.

The **G32WRITE** function requires one or more adapters used to connect to a mainframe host.

Parameters

Item	Description
MSG	Gives 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	Specifies the length, 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 to the following values:

Value	Description
0	Indicates a normal return; call successful.
< 0	Less than 0. Indicates a 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)
```

Network Information Services

This topic collection includes subroutines that derive information from the Network Information Services lookup.

yp_all Subroutine

Purpose

Transfers all of the key-value pairs from the Network Information Services (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) used by other NIS subroutines. 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, identifying a subroutine and map in the normal manner, and supplying a subroutine to process each key-value pair within the map.

The memory pointed to by the *inkey* and *inval* parameters is private to the **yp_all** subroutine. This memory is overwritten with each new key-value pair processed. The **foreach** function uses 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 a new-line or null character in the server's map are not terminated by a new-line or null character in the client's map.

Note: The remote procedure call is returned to the **yp_all** subroutine only after the transaction is completed (successfully or unsuccessfully) or after the **foreach** function rejects any more key-value pairs.

Parameters

Item	Description
data	Specifies state information between the **foreach** function and the mainline code (see also the *indata* parameter).
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 transferred.
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** file.
inkey	Points to the current key of the key-value pair as returned from the server's database.
inkeylen	Returns the length, in bytes, of the *inkey* parameter.
inval	Points to the current value of the key-value pair 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. The *indata* parameter is optional because no part of the NIS client package inspects its contents.

Return Values

The **foreach** subroutine returns a value of 0 when 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.

Related information:

Network Information Service (NIS) Overview for System Management

Remote Procedure Call (RPC) Overview for Programming

yp_bind Subroutine

Purpose

Used in programs to call the **ypbind** daemon directly for processes that use backup strategies when Network Information Services (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 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.
- RPCs to the server fail.

Parameters

Item	Description
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.

Related information:

ypbind command

Network Information Service (NIS) Overview for System Management

Remote Procedure Call (RPC) Overview for Programming

yp_first Subroutine

Purpose

Returns the first key-value pair from the named Network Information Services (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

Item	Description
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, in bytes, of the *outkey* parameter.
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, in bytes, of the *outval* parameter.

Return Values

Upon successful completion, this subroutine returns a value of 0. If unsuccessful, it returns an error as described in the **rpcsvc/yp_prot.h** file.

Related information:

malloc subroutine

Network Information Service (NIS) Overview for System Management

Remote Procedure Call (RPC) Overview for Programming

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 Services (NIS) lookup calls require both a map name and a domain name. 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 NIS remote procedure calls.

Parameters

Item	Description
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 should not be freed by the application.

Return Values

Upon successful completion, this routine returns a value of 0. If unsuccessful, it returns an error as described in the **rpcsvc/ypclnt.h** file.

Related information:

malloc subroutine

Network Information Service (NIS) Overview for System Management

Remote Procedure Call (RPC) Overview for Programming

yp_master Subroutine

Purpose

Returns the machine name of the Network Information Services (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 NIS master server for a map.

Parameters

Item	Description
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 uninitialized 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 subroutine returns a value of 0. If unsuccessful, it returns one of the error codes described in the **rpcsvc/yp_prot.h** file.

Related information:

malloc subroutine

Network Information Service (NIS) Overview for System Management

Remote Procedure Call (RPC) Overview for Programming

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 Services (NIS) map exactly because pattern matching is not available in NIS.

Parameters

Item	Description
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, in bytes, of the key.
outval	Specifies the address of the uninitialized 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, in bytes, of the *outval* parameter.

Return Values

Upon successful completion, this subroutine returns a value of 0. If unsuccessful, it returns one of the error codes described in the **rpcsvc/yp_prot.h** file.

Related information:

malloc subroutine

Network Information Service (NIS) Overview for System Management

Remote Procedure Call (RPC) Overview for Programming

yp_next Subroutine
Purpose

Returns each subsequent value it finds in the named Network Information Services (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** subroutine 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. This will return the second key-value pair associated with the map. To show every entry in the NIS map, the **yp_first** subroutine is called with the **yp_next** subroutine called repeatedly. Each time the **yp_next** subroutine returns a key-value, use it as the *inkey* parameter for the next call.

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, non-NIS database information files or the numerical sorting order of the keys, values, or key-value pairs. If the **yp_first** subroutine is called on a specific map with the **yp_next** subroutine called repeatedly until the process returns a **YPERR_NOMORE** message, every entry in the NIS map is 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, entries may be seen twice or not 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

Item	Description
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, in bytes, of the *inkey* parameter.
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, in bytes, of the *outkey* parameter.
outval	Specifies the address of the uninitialized 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, in bytes, of the *outval* parameter.

Return Values

Upon successful completion, this routine returns a value of 0. If unsuccessful, it returns one of the error codes described in the **rpcsvc/yp_prot.h** file.

Related information:

malloc subroutine

Network Information Service (NIS) Overview for System Management

Remote Procedure Call (RPC) Overview for Programming

yp_order Subroutine

Purpose

Returns the order number for an Network Information Services (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 a NIS map that identifies when the map was built. The number determines whether the local NIS map is more current than the master NIS database.

Parameters

Item	Description
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 10-digit ASCII integer that represents the operating system time, in seconds, when the map was built.

Return Values

Upon successful completion, this routine returns a value of 0. If unsuccessful, it returns one of the error codes described in the **rpcsvc/yp_prot.h** file.

Related information:

Network Information Service (NIS) Overview for System Management

Remote Procedure Call (RPC) Overview for Programming

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 the domain are also freed.

Parameters

Item	Description
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. If unsuccessful, it returns one of the error codes described in the **rpcsvc/yp_prot.h** file.

Related information:

ypbind command

Remote Procedure Call (RPC) Overview for Programming

Sockets Overview

yp_update Subroutine

Purpose

Makes changes to an Network Information Services (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

Note: This routine depends upon the secure Remote Procedure Call (RPC) protocol, and will not work unless the network is running it.

The **yp_update** subroutine is used to make changes to a NIS map. The syntax is the same as that of the **yp_match** subroutine except for the additional *ypop* parameter, which may take on one of the following four values:

Value	Description
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 regardless of whether the item already exists. No error is returned in either case.
ypop_DELETE	Deletes an entry from the map.

Parameters

Item	Description
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, in bytes, of the *inkey* parameter.
indata	Points to the data used as input to the subroutine.
indatalen	Specifies the length, in bytes, of the data used as input to the subroutine.

Return Values

Upon successful completion, this routine returns a value of 0. If unsuccessful, it returns one of the error codes described in the **rpcsvc/yp_prot.h** file.

Files

Item	Description
/var/yp/updaters	A makefile for updating NIS maps.

Related information:

Network Information Service (NIS) Overview for System Management

Remote Procedure Call (RPC) Overview for Programming

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.

Parameters

Item	Description
incode	Contains Network Information Services (NIS) error codes as described in the **rpcsvc/yp_prot.h** file.

Return Values

This subroutine returns a pointer to an error message string corresponding to the *incode* parameter.

Related reference:

"ypprot_err Subroutine"

Related information:

Network Information Service (NIS) Overview for System Management

ypprot_err Subroutine

Purpose

Takes an Network Information Services 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 a NIS protocol error code as input and returns an error code to be used as input to a **yperr_string** subroutine.

Parameters

Item	Description
incode	Specifies the NIS protocol error code used as input to the subroutine.

Return Values

This subroutine returns a corresponding error code to be passed to the **yperr_string** subroutine.

Related reference:

"yperr_string Subroutine" on page 196

Related information:

Network Information Service (NIS) Overview for System Management

Remote Procedure Call (RPC) Overview for Programming

New Data Manager (NDBM)

This topic collection includes functions for data management on the database.

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.

Parameters

Item	Description
db	Specifies the database to close.

Related reference:

"dbmclose Subroutine" on page 203

Related information:

List of NDBM and DBM Programming References

NDBM Overview

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

Item	Description
db	Specifies a database.
key	Specifies the key to delete.

Return Values

Upon successful completion, this subroutine returns a value of 0. If unsuccessful, the subroutine returns a negative value.

Related reference:

"delete Subroutine" on page 204

Related information:

List of NDBM and DBM Programming References

NDBM Overview

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

Item	Description
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 the subroutine is unsuccessful, a null value is indicated in the `dptr` field of the **datum** structure.

Related reference:

"fetch Subroutine" on page 205

Related information:

List of NDBM and DBM Programming References

NDBM Overview

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.

Parameters

Item	Description
db	Specifies the database to access.

Return Values

Upon successful completion, this subroutine returns a **datum** structure containing the value returned for the specified key. If the subroutine is unsuccessful, a null value is indicated in the `dptr` field of the **datum** structure.

Related reference:

"firstkey Subroutine" on page 206

Related information:

List of NDBM and DBM Programming References

NDBM Overview

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.

Parameters

Item	Description
db	Specifies the database to access.

Return Values

Upon successful completion, this subroutine returns a **datum** structure containing the value returned for the specified key. If the subroutine is unsuccessful, a null value is indicated in the `dptr` field of the **datum** structure.

Related reference:

"nextkey Subroutine" on page 206

Related information:

List of NDBM and DBM Programming References

NDBM Overview

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. The subroutine opens 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

Item	Description
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.

For more information about the *flags* and *mode* parameters, see the **open**, **openx**, or **creat** subroutine.

Return Values

Upon successful completion, this subroutine returns a pointer to the DBM structure. If unsuccessful, it returns a null value.

Related reference:

"dbminit Subroutine" on page 203

Related information:

open subroutine

List of NDBM and DBM Programming References

NDBM Overview

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

Item	Description
db	Specifies the database to store.
key	Specifies the input key.
content	Specifies the value associated with the key to store.
flags	Contains either the **DBM_INSERT** or **DBM_REPLACE** flag.

Return Values

Upon successful completion, this subroutine returns a value of 0. If unsuccessful, the subroutine returns a negative value. When the **dbm_store** subroutine is called with the *flags* parameter set to the **DBM_INSERT** flag and an existing entry is found, it returns a value of 1. If the *flags* parameter is set to the **DBM_REPLACE** flag, the entry will be replaced, even if it already exists.

Related reference:

"store Subroutine" on page 207

Related information:

List of NDBM and DBM Programming References

NDBM Overview

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.

Parameters

Item	Description
db	Specifies the database to close.

Related reference:

"dbm_close Subroutine" on page 198

Related information:

List of NDBM and DBM Programming References

NDBM Overview

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.

Parameters

Item	Description
file	Specifies the path name of the database to open.

Return Values

Upon successful completion, this subroutine returns a value of 0. If unsuccessful, the subroutine returns a negative value.

Related reference:

"dbm_open Subroutine" on page 201

Related information:

List of NDBM and DBM Programming References

NDBM Overview

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.

Parameters

Item	Description
key	Specifies the key to delete.

Return Values

Upon successful completion, this subroutine returns a value of 0. If unsuccessful, the subroutine returns a negative value.

Related reference:

"dbm_delete Subroutine" on page 198

Related information:

List of NDBM and DBM Programming References

NDBM Overview

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.

Parameters

Item	Description
key	Specifies the input key.

Return Values

Upon successful completion, this subroutine returns data corresponding to the specified key. If the subroutine is unsuccessful, a null value is indicated in the `dptr` field of the returned **datum** structure.

Related reference:

"dbm_fetch Subroutine" on page 199

Related information:

List of NDBM and DBM Programming References

NDBM Overview

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.

Return Values

Returns a **datum** structure containing the first key value pair.

Related reference:
"dbm_firstkey Subroutine" on page 200

Related information:
List of NDBM and DBM Programming References
NDBM Overview

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

Item	Description
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.

Related reference:

"dbm_nextkey Subroutine" on page 201

Related information:

List of NDBM and DBM Programming References

NDBM Overview

store Subroutine

Purpose

Places data under a key.

Library

DBM Library (**libdbm.a**)

Syntax

```
#include <dbm.h>

int store ( key,  content)
datum key, content;
```

Description

The **store** subroutine places data under a key.

Parameters

Item	Description
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. If unsuccessful, the subroutine returns a negative value.

Related reference:

"dbm_store Subroutine" on page 202

Related information:

List of NDBM and DBM Programming References

NDBM Overview

Remote Procedure Calls (RPC)

This topic collection includes subroutines uses RPC to perform different functions.

a

The following RPC subroutines begin with the letter a.

auth_destroy Macro

Important: The macro is exported from both the **libc** and the **libnsl** libraries.

auth_destroy Macro Exported from the libc Library

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.

Parameters

Item	Description
auth	Points to the authentication information structure to be destroyed.

auth_destroy Macro Exported from the libnsl Library

Purpose

Destroys authentication information.

Library

Network Services Library (**libnsl.a**)

Syntax

```
#include <rpc/rpc.h>
void auth_destroy ( auth)
AUTH *auth;
```

Description

The **auth_destroy** macro destroys the client authentication information associated with the *auth* parameter. The *auth* parameter, which points to an authentication structure that is present in the client handle (the **cl_auth** field), is passed to the server when a remote procedure call (RPC) is made. The

private data structures are deallocated when the authentication information structure is destroyed. The usage of the *auth* parameter is undefined after a call to this macro.

Parameters

Item	Description
auth	Points to the authentication information structure to be destroyed.

Examples

```
#include <rpc/rpc.h>
#include <stdio.h>

int main()
{
  rpcprog_t PROGNUM = 0x3fffffffL;
  rpcvers_t PROGVER = 0x1L ;
  char *nettype = "visible";
  char hostname[255] ;      /* The name of remote host */
  AUTH  *auth

  /* Create client handle */
  if ((cl=clnt_create( hostname, PROGNUM, PROGVER, nettype)) == NULL)
  {
    fprintf(stdout, "clnt_create : failed.\n");
    exit(EXIT_FAILURE);
  }

  /* Create default authentication structure */
  auth = authsys_create_default();
  cl->cl_auth = auth;

  /*
   * Make a CLNT_CALL
   */

  /* Destroy the authentication information */
  auth_destroy(cl->cl_auth);

  /* Destroy the client handle */
  clnt_destroy(cl);

  return 0;
}
```

Related reference:

"authnone_create Subroutine" on page 214

"authunix_create Subroutine" on page 217

"authunix_create_default Subroutine" on page 218

authdes_create Subroutine

Purpose

Enables the use of Data Encryption Standard (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 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

Item	Description
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. However, 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.

Related information:

List of RPC Programming References

Remote Procedure Call (RPC) Overview for Programming

authdes_getucred Subroutine

Important: The subroutine is exported from both the **libc** and the **libnsl** libraries.

authdes_getucred Subroutine Exported from the libc Library

Purpose

Maps a Data Encryption Standard (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 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

Item	Description
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.

authdes_getucred Subroutine Exported from the libnsl Library

Purpose

Maps a Data Encryption Standard (DES) credential into a UNIX credential.

Library

Network Services Library (**libnsl.a**)

Syntax

```
#include <rpc/rpc.h>

int authdes_getucred (authdes_credential, user_idp, group_idp, grouplen, groups)
const struct authdes_cred *authdes_credential ;
uid_t *user_idp ;
gid_t *group_idp ;
short *grouplen ;
gid_t *groups ;
```

Description

The **authdes_getucred** subroutine, which belongs to the secure RPC category, is used on server side to convert an operating-system-independent **AUTH_DES** credential into an **AUTH_SYS** UNIX credential.

Note: The **keyserv** daemon must be running for the **AUTH_DES** authentication mechanism to work. You must run the **keylogin** command before calling the subroutine.

Parameters

Item	Description
authdes_credential	Points to the DES credential structure.
user_idp	Specifies the effective user ID (UID) of the caller.
group_idp	Specifies the effective group ID (GID) of the caller.
grouplen	Specifies the group's length.
groups	Points to the group's array.

Return Values

Item	Description
1	successful
0	unsuccessful

Examples

```
#include <rpc/rpc.h>
static void dispatch(struct svc_req *, SVCXPRT *);

main()
{
  rpcprog_t  RPROGNUM = 0x3fffffffL;
  rpcvers_t  RPROGVER = 0x1L;

  /* Create service handle for RPROGNUM, RPROGVER and tcp transport */
  if(!svc_create( dispatch, RPROGNUM, RPROGVER, "tcp")) {
    fprintf(stderr,"\nsvc_create() failed\n");
    exit(EXIT_FAILURE);
  }

  svc_run();
}

/* The server dispatch function */
static void dispatch(struct svc_req *rqstp, SVCXPRT *transp)
{
  struct authdes_cred *des_cred;
  uid_t uid;
  gid_t gid;
  int gidlen;
  gid_t gidlist[10];

  switch (rqstp->rq_cred.oa_flavor) {

  case AUTH_DES :
  /* AUTH_DES Authentication flavor */
  des_cred = (struct authdes_cred *) rqstp->rq_clntcred;
  if (!authdes_getucred(des_cred, &uid, &gid, &gidlen, gidlist)) {
    svcerr_systemerr(transp);
    return;
  }
  break;

  default :
  /* Other Authentication flavor */
  break;
  }

  /* The Dispatch Routine code continues .. */
}
```

authdes_seccreate Subroutine
Purpose

Maps a UNIX credential into a data encryption standard (DES) credential .

Library

Network Services Library (**libnsl.a**)

Syntax

```
#include <rpc/rpc.h>

AUTH *authdes_seccreate(netname, time_window, time_host, deskey)
const char *netname;
const uint_t time_window;
const char *time_host;
const des_block *deskey;
```

Description

The **authdes_seccreate** subroutine, which belongs to the secure remote procedure call (RPC) category, implements the **AUTH_DES** authentication flavor. This subroutine is used on the client side to convert a UNIX credential to an operating-system-independent **AUTH_DES** credential. When the time difference between the client clock and the server clock exceeds the valid time period, the server rejects client credentials. In such case, you can consult with the host specified by the *time_host* parameter to resynchronize the client and server clocks. The *time_host* and *deskey* parameters are optional. When you set the *time_host* parameter to a null value, the local clock is always in sync with the clock on the specified host. When you set the *deskey* parameter to a null value, a random DES key is generated for encrypting client credentials.

Note: The **AUTH_DES** authentication mechanism works only when the **keyserv** daemon is running. Also, you must have run the **keylogin** command.

Parameters

Item	Description
netname	Specifies the network name of the owner of the server process.
time_window	Specifies the time period during which a client credential is valid.
time_host	Specifies the host that is consulted in the case of clock drift.
deskey	Specifies the DES key for encrypting client credentials.

Return Values

Item	Description
a valid authentication handle	successful
a null value	unsuccessful

Examples

In the following example, the **authdes_seccreate** subroutine creates and returns an authentication handle, so that the communication between the client and the server takes place using the AUTH_DES authentication.

```
#include <stdlib.h>
#include <rpc/rpc.h>

int main()
{
```

```
   char netname[255]; /* contains netname of owner of server process */
   char rhost[255]; /* Remote host netname on which server resides */
   rpcprog_t PROGNUM = 0x3fffffffL;
   rpcvers_t PROGVER = 0x1L;
   CLIENT *clnt;

   /* Obtain network netname of remote host */
   if (!host2netname(netname, rhost, NULL))
   {
      fprintf(stderr, "\nhost2netname() failed\n");
      exit(EXIT_FAILURE);
}

   /* Create a client handle for remote host rhost for PROGNUM & PROGVER on tcp transport */
   clnt = clnt_create(rhost, PROGNUM, PROGVER, "tcp");
   if (clnt == (CLIENT *) NULL) {
      fprintf(stderr,"client_create() error\n");
      exit(1);
   }

   clnt->cl_auth = authdes_seccreate(netname, 80, rhost, (des_block *)NULL);

   /*
    * Make a call to clnt_call() subroutine
    */

   /* Destroy the authentication handle */
   auth_destroy(clnt->cl_auth);

   /* Destroy the client handle in the end */
   clnt_destroy(clnt);

   return 0;
}
```

Related information:

keyserv subroutine

Transport Independent Remote Procedure Call

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.

Related reference:

"authunix_create Subroutine" on page 217
"auth_destroy Macro" on page 208
"authunix_create_default Subroutine" on page 218

Related information:
List of RPC Programming References
Remote Procedure Call (RPC) Overview for Programming

authsys_create or authsys_create_default Subroutine

Purpose

Creates and returns the authentication handle of a remote procedure call (RPC).

Library

Network Services Library (**libnsl.a**)

Syntax

```
#include <rpc/rpc.h>

AUTH * authsys_create( hostname, user_id, group_id, length, aup_gids)
const char *hostname;
const uid_t user_id;
const gid_t group_id;
const int length;
const gid_t *aup_gids;

AUTH *authsys_create_default( void )
```

Description

The **authsys_create** and **authsys_create_default** subroutines belong to the secure-RPC category. The **authsys_create** or **authsys_create_default** subroutine creates and returns an RPC-authentication handle. The authentication information that is passed to the server on each RPC is the AUTH_SYS authentication information.

The **authsys_create_default** subroutine, which is basically a wrapper around the **authsys_create** subroutine, calls the **authsys_create** subroutine with appropriate parameters.

Note: Application programs assign the RPC authentication handle to the **cl_auth** field of the client handle.

Parameters

Item	Description
hostname	Specifies the host name of the server where the authentication information is created.
user_id	Specifies the user ID.
group_id	Specifies the current group ID of the user.
length	Specifies the number of groups to which the user belongs, that is, the length of the *aup_gids* parameter.
aup_gids	Specifies an array of groups to which the user belongs.

Return Values

The **authsys_create** or **authsys_create_default** subroutine returns a pointer to an RPC-authentication handle.

Examples

1.

In the following example, the **authsys_create** subroutine is called after a client handle is created. An authentication handle is returned and assigned to the **cl_auth** field of the client handle. Then, after a successful client call, the **auth_destroy** macro destroys the authentication information associated with the **cl_auth** field.

```
#include <stdlib.h>
#include <rpc/rpc.h>

int main()
{
    rpcprog_t PROGNUM = 0x3fffffffL;
    rpcvers_t PROGVER = 0x1L;
    char *nettype = ="visible";
    char hostname[255];      /* The name of remote hostname */
    AUTH *auth;
    gid_t gids[100];
    int length ;

    /* Set the number of groups to which the user belongs.
     *This value is passed to authsys_create()
     */
    if ((length = getgroups(NGROUPS_MAX, gids)) < 0) {
        printf("failed in getgroups()\n");
        exit(2);
    } else
        length = (length > 16) ? 16 : length;

    if ((cl=clnt_create( hostname, PROGNUM, PROGVER, nettype)) == NULL)
    {
        fprintf(stdout, "clnt_create : failed.\n");
        exit(EXIT_FAILURE);
    }

    /* Set the AUTH structure using AUTH_SYS authentication flavor */
    auth = authsys_create(hostname, getuid(), getgid(), length, gids);
    cl->cl_auth = auth;

    /*
     * Make a CLNT_CALL
     */

    /* Destroy the authentication information */
    auth_destroy(cl->cl_auth);

    /* Destroy the client handle */
    clnt_destroy(cl);

    return 0;

}
```

2. In the following example, the **authsys_create_default** subroutine is called after a client handle is created. An authentication handle is returned and assigned to the **cl_auth** field of the client handle. Then, after a successful client call, the **auth_destroy** macro destroys the authentication information associated with the **cl_auth** field.

```
#include <stdlib.h>
#include <rpc/rpc.h>
int main()
{
    rpcprog_t PROGNUM = 0x3fffffffL;
    rpcvers_t PROGVER = 0x1L;
    char *nettype = "visible";
    char hostname[255];      /* The name of remote host */
    AUTH *auth
```

```
        if ((cl=clnt_create( hostname, PROGNUM, PROGVER, nettype)) == NULL)
        {
            fprintf(stdout, "clnt_create : failed.\n");
            exit(EXIT_FAILURE);
        }

        /* Set the AUTH structure using AUTH_SYS authentication flavor */
        auth = authsys_create_default();
        cl->cl_auth = auth;

        /*
         * Make a CLNT_CALL
        */

        /* Destroy the authentication information */
        auth_destroy(cl->cl_auth);

        /* Destroy the client handle */
        clnt_destroy(cl);

        return 0;
    }
```

Related information:

Transport Independent Remote Procedure Call

eXternal Data Representation Overview for Programming

authunix_create Subroutine

Purpose

Creates an authentication handle with operating system 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 operating system permissions.

Parameters

Item	Description
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.

Related reference:

"authnone_create Subroutine" on page 214

"auth_destroy Macro" on page 208

"authunix_create_default Subroutine"

Related information:

List of RPC Programming References

Remote Procedure Call (RPC) Overview for Programming

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 operating system authentication handle.

Return Values

Upon successful completion, this subroutine returns an authentication handle.

Related reference:

"authnone_create Subroutine" on page 214

"authunix_create Subroutine" on page 217

"auth_destroy Macro" on page 208

C

The following RPC subroutines begin with the letter c.

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

Item	Description
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.

Related reference:

"clnt_broadcast Subroutine" on page 221

"registerrpc Subroutine" on page 302

cbc_crypt, des_setparity, or ecb_crypt Subroutine
Purpose

Implements Data Encryption Standard (DES) encryption routines.

Library

DES library (**libdes.a**)

Syntax

```
# include <des_crypt.h>

int ecb_crypt ( key,  data,  datalen, mode)
char *key;
char *data;
unsigned datalen;
unsigned mode;

int cbc_crypt(key, data, datalen,  mode,  ivec)
char *key;
char *data;
unsigned datalen;
unsigned mode;
char ivec;
void des_setparity(key)
char *key;
```

Description

The **ecb_crypt** and **cbc_crypt** subroutines implement DES encryption routines, set by the National Bureau of Standards.

- The **ecb_crypt** subroutine encrypts in ECB (Electronic Code Book) mode, which encrypts blocks of data independently.
- The **cbc_crypt** subroutine encrypts in CBC (Cipher Block Chaining) mode, which chains together successive blocks. CBC mode protects against insertions, deletions, and substitutions of blocks. Also, regularities in the clear text will not appear in the cipher text.

  These subroutines are not available for export outside the United States.

  Note: The DES library must be installed to use these subroutines.

Parameters

Item	Description
data	Specifies that the data is to be either encrypted or decrypted.
datalen	Specifies the length in bytes of data. The length must be a multiple of 8.
key	Specifies the 8-byte encryption key with parity. To set the parity for the key, which for DES is in the low bit of each byte, use the **des_setparity** subroutine.
ivec	Initializes the vector for the chaining in 8-byte. This is updated to the next initialization vector upon return.
mode	Specifies whether data is to be encrypted or decrypted. This parameter is formed by logically ORing the **DES_ENCRYPT** or **DES_DECRYPT** symbols. For software versus hardware encryption, logically OR the **DES_HW** or **DES_SW** symbols. These four symbols are defined in the **/usr/include/des_crypt.h** file.

Return Values

Item	Description
DESERR_BADPARAM	Specifies that a bad parameter was passed to routine.
DESERR_HWERR	Specifies that an error occurred in the hardware or driver.
DESERR_NOHWDEVICE	Specifies that encryption succeeded, but was done in software instead of the requested hardware.
DESERR_NONE	Specifies no error.

Note: Given the **stat** variable, for example, which contains the return value for either the **ecb_crypt** or **cbc_crypt** subroutine, the **DES_FAILED(stat)** macro is false only for the **DESERR_NONE** and **DESERR_NOHWDEVICE** return values.

Files

Item	Description
/usr/include/des_crypt.h	Defines macros and needed symbols for the *mode* parameter.

Related information:
Secure NFS
Example Using DES Authentication

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)
enum clnt_stat clnt_broadcast (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

Item	Description
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.

Related reference:

"callrpc Subroutine" on page 218

Related information:

List of RPC Programming References

Remote Procedure Call (RPC) Overview for Programming

clnt_call Macro

Important: The macro is exported from both the **libc** and the **libnsl** libraries.

clnt_call Macro Exported from the libc Library

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

Item	Description
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.

clnt_call Macro Exported from the libnsl Library

Purpose

Calls the remote procedure associated with the client handle.

Library

Network Services Library (**libnsl.a**)

Syntax

```
#include <rpc/rpc.h>

clnt_call(clnt, procnum, inproc, in, outproc, out, tout)
CLIENT * clnt;
rpcproc_t procnum;
xdrproc_t  inproc;
caddr_t in;
xdrproc_t  outproc;
caddr_t out;
struct timeval  tout;
```

Description

The **clnt_call** macro calls the remote procedure associated with the client file handle. The handle is obtained by calling any of the client creation subroutines. You can specify the eXternal Data Representation (XDR) procedure that encodes the procedure parameters along with the address of the parameters. Similarly, you can specify the XDR procedure that decodes the procedure results and address where those results are to be placed.

Parameters

Item	Description
clnt	Specifies a generic client handle created by a successful call to the **clnt_create** subroutine.
procnum	Specifies the remote procedure number.
inproc	Specifies the XDR procedure that encodes the procedure parameters.
in	Specifies the address of the procedure arguments.
outproc	Specifies the XDR procedure that decodes the procedure results.
out	Specifies the address where results are placed.
tout	Sets the time given to the server to respond.

Return Values

Item	Description
RPC_SUCCESS	successful
nonzero	unsuccessful

Error Codes

The **clnt_call** macro returns failure when one or more of the following codes are true.

Item	Description
RPC_PROCUNAVAIL	The remote procedure is not available.
RPC_TIMEDOUT	The timeout value has expired.
RPC_AUTHERROR	Authentication failure occurred.
RPC_FAILED	An unspecified error occurred.
RPC_UNKNOWNPROTO	The protocol is unknown.
RPC_UNKNOWNADDR	The remote address is unknown.
RPC_CANTENCODEARGS	The specified XDR procedures cannot encode arguments or cannot decode results.
RPC_PROGUNAVAIL	The program is not available.
RPC_INTR	The call is interrupted.

Examples

```
#include <stdlib.h>
#include <rpc/rpc.h>

int main()
{
    CLIENT *client ;
    char hostname[255] ;
    /* The Remote host on which server is implemented */
    rpcprog_t program_number = 0x3fffffffL;
    rpcvers_t version_number = 0x1L;
    rpcproc_t procedure_number = 0x1L;
    struct timeval total_timeout = { 25 , 0 } ;
    enum clnt_stat cs ;

    /* Create client handle by calling clnt_create subroutine */
    client = clnt_create(hostname, program_number, version_number, "tcp");
    if (client == (CLIENT *)NULL)
    {
        fprintf(stderr,"Couldn't create client\n");
        exit(1);
        }

    /* Calls the remote procedure */
    cs = clnt_call(client, procedure_number, (xdrproc_t)xdr_void,
                  NULL, (xdrproc_t)xdr_void, NULL, total_timeout);
    if (cs != RPC_SUCCESS)
    {
        fprintf(stderr,"\n Client Call failed\n");
        exit(1);
    }

    /* Destroy the client handle at the end */
    clnt_destroy(client);

    return 0;
}
```

Related reference:

"rpc_call Subroutine" on page 308

clnt_control Macro

Important: The macro is exported from both the **libc** and the **libnsl** libraries.

clnt_control Macro Exported from the libc Library

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.
CLGET_SERVER_ADDR	**struct sockaddr**	Gets server's address.

The following operations are valid for UDP only:

Values for the req Parameter	Argument Type	Function
CLSET_RETRY_TIMEOUT	**struct timeval**	Sets the retry time out.
CLGET_RETRY_TIMEOUT	**struct timeval**	Gets the retry time out.

Note:

1. If the time out is set using the **clnt_control** subroutine, the time-out parameter passed to the **clnt_call** subroutine will be ignored in all future calls.
2. 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

Item	Description
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.

clnt_control Macro Exported from the libnsl Library

Purpose

Changes or retrieves information of the client handle.

Library

Network Services Library (**libnsl.a**)

Syntax

```
#include <rpc/rpc.h>

bool_t clnt_control(cl, req, info)
CLIENT * cl;
int req;
char * info;
```

Description

The **clnt_control** macro subroutine is a simplified-level subroutine for transport-independent remote procedure calls (TI_PRC). You can use the subroutine to change or retrieve information about the client handle. The generic client handle that is obtained from various client create subroutines is supplied as the input parameter. You need to specify the type of operation along with the pointer to the information. For both connectionless and connection-oriented transports, we can either set or get various information about client objects. The data type of the *info* parameter changes according to the type of operation. For example, you can specify the *req* parameter with the following values:

CLGET_VERS
Gets the version number of the RPC program associated with the client.

CLSET_VERS
Sets the version number of the RPC program associated with the client.

For the *req* parameter in the example, the value of the *info* parameter is of the **rpcvers_t** type.

Values for the *req* Parameter	Argument Type	Function
CLSET_TIMEOUT	**struct timeval ***	Sets the total timeout.
CLGET_TIMEOUT	**struct timeval ***	Gets the total timeout.
CLGET_SVC_ADDR	**struct netbuf ***	Gets the address of the server.
CLGET_FD	**int ***	Gets associated file descriptor.
CLSET_RETRY_TIMEOUT	**struct timeval ***	Sets the retry timeout.
CLGET_RETRY_TIMEOUT	**struct timeval ***	Gets the retry timeout.
CLGET_VERS	**rpcvers_t**	Gets the version number of the RPC program.
CLSET_VERS	**rpcvers_t**	Sets the version number of the RPC program.

Values for the *req* Parameter	Argument Type	Function
CLGET_XID	**uint32_t**	Gets XID of the previous RPC.
CLSET_XID	**uint32_t**	Sets XID of the previous RPC.
CLGET_PROG	**rpcprog_t**	Gets the program number.
CLSET_PROG	**rpcprog_t**	Sets the program number.

Parameters

Item	Description
cl	Points to the structure of the generic RPC-client handle .
req	Indicates the type of the operation.
info	Points to the information for request type. The *info* parameter is expected to be a pointer to an appropriate structure. The nature of the structure depends on the type of the operation.

Return Values

Item	Description
1	successful
0	unsuccessful

Examples

In the following example, the **clnt_control** macro subroutine returns the version number of the program that is specified by the *versnum* parameter.

```
#include <stdlib.h>
#include <rpc/rpc.h>

int main()
{
    struct timeval t;
    CLIENT *cl ;
    rpcprog_t PROG = 0x3fffffffL;
    rpcvers_t versnum,PROGVER = 0x1L;
    char hostname[255] /* The Remote Host */
    char *nettype = "visible" ;

    /* Create generic client handle */
    cl = clnt_create( hostname, PROG, PROGVER, nettype);
    if(cl==NULL)
    {
        fprintf(stderr,"Couldnot create client handle");
        exit(1);
    }

    if(!clnt_control(cl, CLGET_VERS, versnum))
    {
        fprintf(stderr,"Failed in clnt_control routine");
        exit(1);
    }
    fprintf(stdout,"\n VERSION NUMBER = %lu \n", versnum);

    /* Destroy the client handle at the end */
    clnt_destroy(cl);
    return 0;
}
```

clnt_create Subroutine

Important: The subroutine is exported from both the **libc** and the **libnsl** libraries.

clnt_create Subroutine Exported from the libc Library

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.

Remote Procedure Calls (RPC) messages transported by User Datagram Protocol/Internet Protocol (UDP/IP) can hold up to 8KB of encoded data. Use this transport for procedures that take arguments or return results of less than 8KB.

Note: When the **clnt_create** subroutine is used to create a RPC client handle, the timeout value provided on subsequent calls to **clnttcp_call** are ignored. Using the **clnt_create** subroutine has the same effect as using **clnttcp_create** followed by a call to **clnt_control** to set the timeout value for the RPC client handle. If the timeout paramater is used on the **clnttcp_call** interface, use the **clnttcp_create** interface to create the client handle.

Parameters

Item	Description
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, either UDP or Transmission Control Protocol (TCP).

Return Values

Upon successful completion, this subroutine returns a client handle.

clnt_create Subroutine Exported from the libnsl Library

Purpose

Creates and returns a generic client handle for a remote program.

Library

Network Services Library (**libnsl.a**)

Syntax

```
#include <rpc/rpc.h>
CLIENT *clnt_create(host, prognum, versnum, nettype)
const char *host;
const rpcprog prognum;
const rpcvers_t versnum;
char *nettype;
```

Description

The **clnt_create** subroutine is a top-level API for transport independent remote procedure calls (TI_PRC). The subroutine creates and returns a generic client handle for the specified program and version. This generic client handle is returned from the remote host on which the server is running. This operation is done with the available transport service of the class that is specified by the *nettype* parameter. The **clnt_create** subroutine chooses the first successful transport from the NETPATH environment variable and then from the **netconfig** database in a top-to-bottom order. A default timeout value specifies the time for the **clnt_create** subroutine to return. If the timeout value expires, the subroutine returns NULL. You can modify the timeout value using the **clnt_control** macro subroutine.

Note:

1. The **clnt_pcreateerror** subroutine can be used to obtain the reason for failure of the creation of an RPC-client handle.
2. The **clnt_create** subroutine returns a valid client handle even if the specified version number is not supported by the server. The **clnt_call** subroutine will recognize the condition and return failure.

Parameters

Item	Description
host	Specifies the host name where the server resides.
prognum	Specifies the program number of the remote program.
versnum	Specifies the version number of the remote program.
nettype	Defines a class of transports that can be used for a particular application.

Return Values

Item	Description
a generic client handle that is valid	successful
NULL	unsuccessful

Error Codes

The **clnt_create** subroutine returns failure when one or more of the following codes are true.

Item	Description
RPC_UNKNOWNPROTO	• The value specified by the *nettype* parameter is not valid. • The value specified by the *nettype* parameter is set to **netpath**, and the NETPATH environment variable is set to a transport service that is not valid.
RPC_UNKNOWNHOST	The host name is not valid.
RPC_PROGNOTREGISTERED	The program number is not valid.

Examples

```
#include <stdlib.h>
#include <rpc/rpc.h>

int main()
{
```

```
  CLIENT *cl;
  rpcprog_t PROGNUM = 0x3fffffffL;
  rpcvers_t PROGVER = 0x1L;
  char *nettype = "visible";
  char hostname[255]; /* The name of remote host */

  /*
  * make the clnt_create call with this nettype and
  * observe the result
  */

  if ((cl=clnt_create( hostname, PROGNUM, PROGVER, nettype)) == NULL)
  {
    fprintf(stdout, "clnt_create : failed.\n");
    exit(EXIT_FAILURE);
  }

  /*
   * Make a call to clnt_call() subroutine
   */

  /* Destroy the client handle at the end */
  clnt_destroy(cl);

  return 0;
}
```

Related reference:

"clnt_tp_create Subroutine" on page 260

clnt_create_timed Subroutine

Purpose

Creates and returns a generic client handle for a remote program within the specified time.

Library

Network Services Library (**libnsl.a**)

Syntax

```
#include <rpc/rpc.h>

clnt_create_timed(host, prognum, versnum, nettype, timeout)
const char *host;
const rpcprog_t prognum;
const rpcvers_t versnum;
const char *nettype;
const struct timeval *timeout;
```

Description

The **clnt_create_timed** subroutine is a top-level API for transport-independent remote procedure calls (TI_PRC). The subroutine creates and returns a generic client handle for the specified program and version specified. This generic client handle is returned from the remote host on which server is running. The operation is done using the available transport service of the class that is specified by the *nettype* parameter. The **clnt_create_timed** subroutine chooses the first successful transport from the NETPATH environment variable and then from the **netconfig** database in a top-to-bottom order. The value of the *timeout* parameter specifies the time for the **clnt_create_timed** subroutine to return. If the timeout value expires, the subroutine returns a null value.

Note: The subroutine returns a valid client handle even if the version number specified by the *versnum* parameter is not supported by the server. The **clnt_call** subroutine can recognize the error and return failure.

Parameters

Item	Description
host	Specifies the host name where the server resides.
prognum	Specifies the program number of the remote program.
versnum	Specifies the version number of the remote program.
nettype	Defines a class of transports that can be used for a particular application.
timeout	Specifies the maximum time spent for each transport class in the *nettype* class.

Return Values

Item	Description
a generic client handle that is valid	successful
a null value	unsuccessful

Error Codes

The **clnt_create_timed** subroutine returns failure when one or more of the following codes are true.

Item	Description
RPC_UNKNOWNPROTO	• The value specified by the *nettype* parameter is not valid. • The value specified by the *nettype* parameter is set to **netpath**, and the NETPATH environment variable is set to a transport service that is not valid.
RPC_UNKNOWNHOST	The host name is not valid.
RPC_TIMEDOUT	The timeout value has expired.
RPC_PROGNOTREGISTERED	The program number is not valid.

Examples

```
#include <stdlib.h>
#include <rpc/rpc.h>

int main()
{
    CLIENT *cl;
    rpcprog_t PROGNUM  =  0x3fffffffL;
    rpcvers_t PROGVER  =  0x1L ;
    char *nettype   =  "visible";
    char hostname[255] ;   /* The name of remote host */
    struct timeval tv;

    tv.tv_sec = 5 ;
    tv.tv_usec = 0 ;

    /*
    * make the clnt_create_timed call with this nettype and
    * observe the result
    */

    if ((cl=clnt_create_timed( hostname, PROGNUM, PROGVER, nettype, &tv)) == NULL)
    {
        fprintf(stdout, "clnt_create_timed : failed.\n");
        exit(EXIT_FAILURE);
    }

    /*
```

```
    * Make a call to clnt_call() subroutine
    */

    /* Destroy the client handle at the end */
    clnt_destroy(cl);

    return 0;
}
```

Related reference:

"clnt_tp_create_timed Subroutine" on page 262

"clnt_create_vers_timed Subroutine" on page 234

Related information:

Transport Independent Remote Procedure Call, IPv6 concerns for Transport Independent Remote Procedure Call

clnt_create_vers Subroutine

Purpose

Creates and returns a generic client handle for a remote program and the registered version number that is the highest within the specified range.

Library

Network Services Library (**libnsl.a**)

Syntax

```
#include <rpc/rpc.h>

clnt_create_vers(host, prognum, progver_out, progver_low, progver_high, nettype)
const char *host;
const rpcprog_t prognum;
rpcvers_t *progver_out;
const rpcvers_t progver_low;
const rpcvers_t progver_high;
const char *nettype;
```

Description

The **clnt_create_vers** subroutine creates and returns a generic client handle for the specified program and the highest registered version that falls within the range bounded by the values specified by the *progver_low* and *progver_high* parameters. You must specify the *progver_low* and *progver_high* parameters. When the function returns successfully, the value of the *progver_out* parameter is set to the highest registered version within the specified range (*progver_low* <= *progver_out* <= *progver_high*). The subroutine returns a generic client handle from the remote host where server is located. The operation is done with the available transport service of the class that is specified by the *nettype* parameter. The **clnt_create_vers** subroutine uses first successful transport from the NETPATH environment variable and then from the **netconfig** database if required. You can modify the default timeout value using the **clnt_control** subroutine.

Note: The subroutine returns a null value if no version is registered within the specified range.

Parameters

Item	Description
host	Specifies the host name where the server resides.
prognum	Specifies the program number of the remote program.
versnum	Specifies the version number of the remote program.
progver_out	The highest version number that is registered at the server. The version number is returned within the specified range.
progver_low	The lower limit of the version number specified by the application.
progver_high	The upper limit of the version number specified by the application.
nettype	Defines a class of transports that can be used for a particular application.

Return Values

Item	Description
a generic client handle that is valid	successful
a null value	unsuccessful

Note: You can use the **clnt_pcreateerror** subroutine to obtain the reason for failure.

Error Codes

The **clnt_create_vers** subroutine returns failure when one or more of the following codes are true.

Item	Description
RPC_UNKNOWNPROTO	• The value specified by the *nettype* parameter is not valid. • The value specified by the *nettype* parameter is set to **netpath**, and the NETPATH environment variable is set to a transport service that is not valid.
RPC_UNKNOWNHOST	The host name is not valid.
RPC_PROGVERSMISMATCH	No version is registered at the server within the range bounded by the values specified by the *progver_low* and *progver_high* parameters.
RPC_PROGNOTREGISTERED	The program number is not valid.

Examples

```
#include <stdlib.h>
#include <rpc/rpc.h>

int main()
{
    CLIENT *cl;
    rpcprog_t PROGNUM = 0x3fffffffL;
    rpcvers_t PROGVER_OUT ;
    char *nettype = "visible";
    rpcvers_t PROGVER_LOW = 1;
    rpcvers_t PROGVER_HIGH = 10;
    char hostname[255]; /* The Remote host on which the server resides */

    /*
    * make the clnt_create_vers call with this nettype and
    * observe the result
    */
    if ((cl=clnt_create_vers( hostname, PROGNUM, &PROGVER_OUT,
                   PROGVER_LOW, PROGVER_HIGH, nettype)) == NULL)
    {
         fprintf(stdout, "clnt_create_vers : failed.\n");
         exit(EXIT_FAILURE);
    }
    /*
     * Make a call to clnt_call() subroutine
     */
    /* Destroy the client handle at the end */
```

```
    clnt_destroy(cl);

     return 0;
}
```

Related information:

Transport Independent Remote Procedure Call, IPv6 concerns for Transport Independent Remote Procedure Call

eXternal Data Representation Overview for Programming

clnt_create_vers_timed Subroutine

Purpose

Creates and returns a generic client handle for a remote program and the registered version number that is the highest in the specified range with the specified timeout.

Library

Network Services Library (**libnsl.a**)

Syntax

```
#include <rpc/rpc.h>

clnt_create_vers_timed(host, prognum, progver_out, progver_low, progver_high, nettype, timeout)
const char *host;
const rpcprog_t prognum;
rpcvers_t *progver_out;
const rpcvers_t progver_low;
const rpcvers_t progver_high;
const char *nettype;
const struct timeval *timeout;
```

Description

The **clnt_create_vers_timed** subroutine creates and returns a generic client handle for the specified program and the highest registered version that falls within the range bounded by the values specified by the *progver_low* and *progver_high* parameters. You must specify the *progver_low* and *progver_high* parameters. When the function returns successfully, the value of the *progver_out* parameter is set to the highest registered version within the specified range (*progver_low* <= *progver_out* <= *progver_high*). The subroutine returns a generic client handle from the remote host where server is located. The operation is done with the available transport service of the class specified by the *nettype* parameter. The **clnt_create_vers_timed** subroutine uses first successful transport from the NETPATH environment variable and then from the **netconfig** database if required. The value of the *timeout* parameter indicates the maximum amount of time that is spent for each transport class.

Note: The subroutine returns a null value if no version is registered within the specified range or when the timeout value expires.

Parameters

Item	Description
host	Specifies the host name where the server resides.
prognum	Specifies the program number of the remote program.
versnum	Specifies the version number of the remote program.
progver_out	The highest version number that is registered at the server. The version number is returned within the specified range.
progver_low	The lower limit of the version number specified by the application.
progver_high	The upper limit of the version number specified by the application.
nettype	Defines a class of transports that can be used for a particular application.
timeout	Specifies the maximum time that is spent for each transport class in the *nettype* class.

Return Values

Item	Description
a generic client handle that is valid	successful
a null value	unsuccessful

Note: You can use the **clnt_pcreateerror** subroutine to obtain the reason for failure.

Error Codes

Item	Description
RPC_UNKNOWNPROTO	• The value specified by the *nettype* parameter is not valid. • The value specified by the *nettype* parameter is set to **netpath**, and the NETPATH environment variable is set to a transport service that is not valid.
RPC_UNKNOWNHOST	The host name is not valid.
RPC_PROGVERSMISMATCH	No version is registered at the server within the range bounded by the values specified by the *progver_low* and *progver_high* parameters.
RPC_TIMEDOUT	The timeout value has expired.
RPC_PROGNOTREGISTERED	The program number is not valid.

Examples

```
#include <stdlib.h>
#include <rpc/rpc.h>

int main()
{
    CLIENT *cl;
    rpcprog_t PROGNUM = 0x3fffffffL;
    rpcvers_t PROGVER_OUT ;
    char *nettype = "visible";
    rpcvers_t PROGVER_LOW = 1;
    rpcvers_t PROGVER_HIGH = 10;
    struct timeval tv;
    char hostname[255] ; /* The Remote host on which the server resides */

    tv.tv_sec = 25;
    tv.tv_usec = 0;
      /*
    * make the clnt_create_vers_timed call with this nettype and
    * observe the result
    */
    if ((cl=clnt_create_vers_timed( hostname, PROGNUM, &PROGVER_OUT,
            PROGVER_LOW, PROGVER_HIGH, nettype, &tv)) == NULL)
    {
        fprintf(stdout, "clnt_create_vers_timed : failed.\n");
        exit(EXIT_FAILURE);
    }
    /*
```

```
     * Make a call to clnt_call() subroutine
     */
    /* Destroy the client handle at the end */
    clnt_destroy(cl);

     return 0;
}
```

Related reference:

"clnt_create_timed Subroutine" on page 230

Related information:

Transport Independent Remote Procedure Call, IPv6 concerns for Transport Independent Remote Procedure Call

eXternal Data Representation Overview for Programming

clnt_destroy Macro

Important: The macro is exported from both the **libc** and the **libnsl** libraries.

clnt_destroy Macro Exported from the libc Library

Purpose

Destroys the client's Remote Procedure Call (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 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.

Parameters

Item	Description
clnt	Points to the structure of the client handle.

clnt_destroy Macro Subroutine Exported from the libnsl Library

Purpose

Destroys the handle of a remote procedure call (RPC) client.

Library

Network Services Library (**libnsl.a**)

Syntax

```
#include <rpc/rpc.h>

void clnt_destroy(clnt)
CLIENT *clnt;
```

Description

The **clnt_destroy** macro subroutine is a top-level API for transport-independent remote procedure calls (TI_PRC). The macro subroutine destroys the handle of the RPC client that is obtained after a successful call to any of the client-creation subroutines. The *clnt* parameter is deallocated along with other private data structures. After a call to this macro subroutine, the use of the *clnt* parameter is undefined. Any associated file descriptor will be closed if the RPC library has opened the associated file descriptor or was set using the **clnt_control** subroutine.

Parameters

Item	Description
clnt	Points to the structure of the client handle.

Return Values

The **clnt_destroy** macro subroutine fails if the specified client handle has a null value.

Examples

In the following example, the **clnt_destroy** macro subroutine successfully destroys the client handle that is returned by the **clnt_create** subroutine.

```
#include <stdlib.h>
#include <rpc/rpc.h>

int main()
{
    CLIENT *cl;
    rpcprog_t PROGNUM =      x3fffffffL ;
    rpcvers_t PROGVER =      x1L ;
    char *nettype = "visible";
    char hostname[255];      /* The name of remote host */

      /*
    * make the clnt_create call with this nettype and
    * observe the result
    */
    if ((cl=clnt_create( hostname, PROGNUM, PROGVER, nettype)) == NULL)
    {
         fprintf(stdout, "clnt_create : failed.\n");
         exit(EXIT_FAILURE);
    }
    /*
     * Make a call to clnt_call() subroutine
     */
    /* Destroy the client handle when no more needed */
     clnt_destroy( cl );

    return 0;
}
```

clnt_dg_create Subroutine
Purpose

Creates and returns a generic client handle for a remote program using a connectionless transport.

Library

Network Services Library (**libnsl.a**)

Syntax

```
#include <rpc/rpc.h>
CLIENT * clnt_dg_create(fd, svcaddr, prognum, versnum, sendsize, recvsize)
int fd;
const struct netbuf *svcaddr;
const rpcprog_t prognum;
const rpcvers_t versnum;
const uint_t sendsize;
const uint_t recvsize
```

Description

The **clnt_dg_create** subroutine is a bottom-level API for transport-independent remote procedure calls (TI_PRC). With the subroutine, applications can control all the options. The **clnt_dg_create** subroutine creates and returns a generic client handle for the specified program and version. The subroutine uses a connectionless transport. The generic client handle is returned from the remote host where the server is located. The subroutine uses an open and bound file descriptor through the connectionless transport and the specified address of the remote program to call the remote program. If you set the sizes of the send and receive buffers that can be specified by the *sendsize* and *recvsize* parameter to 0, the default sizes of the buffers are used. This subroutine resends the call message after an interval of 15 seconds until the subroutine receives a response, or the call times out. The **clnt_call** subroutine specifies the timeout value. You can use the **clnt_control** subroutine to modify the retry time and timeout values.

Note: If you set the value of the *fd* parameter to RPC_ANYFD or set the value of the *svcaddr* parameter to a null value, the subroutine fails and returns a null value.

Parameters

Item	Description
fd	Specifies the open and bound file descriptor on a connectionless transport.
svcaddr	Specifies the address of the remote program.
prognum	Specifies the program number of the remote program.
versnum	Specifies the version number of the remote program.
sendsize	Specifies the size of the send buffer.
recvsize	Specifies the size of the receive buffer.

Return Values

Item	Description
a generic client handle that is valid	successful
a null value	unsuccessful

You can use the **clnt_pcreateerror** subroutine to obtain the reason for failure.

Error Codes

The **clnt_dg_create** subroutine returns failure if one or more of the following codes are true.

Item	Description
RPC_TLIERROR	The file descriptor is not valid.
RPC_UNKNOWNADDR	The value of the *svcaddr* parameter that holds the address of the remote program is NULL.
RPC_CANTENCODEARGS	The size of the send or receive buffer is less than that of the sent packet.

Examples

```
#include <stdlib.h>
#include <rpc/rpc.h>

int main()
{
    CLIENT *cl ;
    int fd;
    rpcprog_t PROGNUM = 0x3fffffffL;
    rpcvers_t PROGVER = 0x1L;
    struct netconfig *nconf ;
    struct netbuf svcaddr;
    char hostname[255];      /* The name of remote host */

    if ((nconf = getnetconfigent("udp")) == (struct netconfig *)NULL)
    {
        fprintf(stderr, "Cannot get netconfig entry for UDP\n");
        exit(1);
    }

    if (!rpcb_getaddr(PROGNUM, PROGVER, nconf,&svcaddr, hostname))
    {
        fprintf(stderr, "rpcb_getaddr failed!!\n");
        exit(1);
    }
    /* Get the file descriptor for connection oriented transport */
    fd = . . .

    if ((cl = clnt_dg_create(fd, &svcaddr,
                             PROGNUM, PROGVER, 0, 0))==NULL);
    {
        fprintf(stdout, "clnt_dg_create : failed.\n");
        exit(1);
    }

   /*
    * Make a call to clnt_call() subroutine
    */

    /* Destroy the client handle at the end */
    clnt_destroy(cl);

    return 0;
}
```

Related information:

Transport Independent Remote Procedure Call, IPv6 concerns for Transport Independent Remote Procedure Call

eXternal Data Representation Overview for Programming

clnt_door_create Subroutine

Purpose

Creates and returns a generic client handle for a program over the doors-transport mechanism.

Library

Network Services Library (**libnsl.a**)

Syntax

```
#include <rpc/rpc.h>

CLIENT * clnt_door_create(prognum, versnum, sendsize)
const rpcprog_t prognum;
const rpcvers_t versnum;
const uint_t sendsize;
```

Description

The **clnt_door_create** subroutine creates and returns a generic client handle for the specified program and version. The subroutine creates the client handle over the doors-transport mechanism that can accelerate the data transfer between different processes on the same machine. If you set the size of the send buffer that is specified by the *sendsize* parameter to 0, the default size of 16KB is used.

Parameters

Item	Description
prognum	Specifies the program number of the remote program.
versnum	Specifies the version number of the remote program.
sendsize	Specifies the size of the send buffer.

Return Values

Item	Description
a generic client handle that is valid	successful
NULL	unsuccessful

Error Codes

The **clnt_door_create** subroutine returns failure when one or more of the following codes are true.

Item	Description
RPC_UNKNOWNHOST	The host name is not valid.
RPC_PROGVERSMISMATCH	The version number is not valid.

Examples

```
#include <stdlib.h>
#include <rpc/rpc.h>

int main()
{
    CLIENT *cl;
    rpcprog_t PROGNUM = 0x3fffffffL;
    rpcvers_t PROGVER = 0x1L ;

    /*
     * make the clnt_door_create call with this nettype and
     * observe the result
     */

    if ((cl=clnt_door_create( PROGNUM, PROGVER, 0)) == NULL)
    {
         fprintf(stdout, "clnt_door_create : failed.\n");
         exit(EXIT_FAILURE);
```

```
    }

    /*
     * Make a call to clnt_call() subroutine
     */

    /* Destroy the client handle in the end */
    clnt_destroy(cl);

    return 0;
}
```

Related information:

Transport Independent Remote Procedure Call, IPv6 concerns for Transport Independent Remote Procedure Call

eXternal Data Representation Overview for Programming

clnt_freeres Macro

Important: The macro is exported from both the **libc** and the **libnsl** libraries.

clnt_freeres Macro Exported from the libc Library

Purpose

Frees data that was allocated by the Remote Procedure Call/eXternal Data Representation (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 RPC/XDR system. This data was allocated when the RPC/XDR system decoded the results of an RPC call.

Parameters

Item	Description
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.

clnt_freeres Macro Exported from the libnsl Library

Purpose

Frees data that was allocated by the Remote Procedure Call/eXternal Data Representation (RPC/XDR) system.

Library

Network Services Library (**libnsl.a**)

Syntax

```
#include <rpc/rpc.h>

bool_t clnt_freeres (clnt, outproc, out)
CLIENT *clnt;
xdrpoc_t outproc;
caddr_t out;
```

Description

The **clnt_freeres** macro frees data allocated by the RPC/XDR system. This data is allocated when the RPC/XDR system decoded the results of an RPC call. You must specify the address of the results along with the procedure to decode it.

Parameters

Item	Description
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.

Return Values

Item	Description
1	successful
0	unsuccessful

Examples

```
#include <stdlib.h>
#include <rpc/rpc.h>
#include <sys/types.h>
#include <sys/select.h>
int main()
{

  rpcprog_t PROGNUM = 0x3fffffffL;
  rpcvers_t PROGVER = 0x1L;
  rpcproc_t procnum 0x1L;
  CLIENT *clnt;
  enum clnt_stat stat;
  struct timeval timeout = {25,0};
  char *nettype = "tcp";
  char hostname[255] ; /* The Remote Host */

  struct arguments{
     unsigned int size;
     char *data;
  };
  struct arguments input_arguments ;
  struct arguments output_results  ;

  if ((clnt=clnt_create(hostname, PROGNUM, PROGVER, nettype))==NULL)
  {
    fprintf(stderr,"clnt_create() subroutine failed");
    exit(1);
  }
```

```
  stat = clnt_call(clnt, procnum, (xdrproc_t)xdr_array,
           (char *)&input_arguments, (xdrproc_t)xdr_array,
           (char *)&output_results, timeout);

  if(!clnt_freeres(clnt, (xdrproc_t)xdr_array,(caddr_t )&output_results))
  {
    fprintf(stderr,"clnt_freeres failed");
  }

  /* Destroy client handle in the end */
  clnt_destroy(clnt);

  return 0;
}
```

clnt_geterr Macro

Important: The macro is exported from both the **libc** and the **libnsl** libraries.

clnt_geterr Macro Exported from the libc Library

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

Item	Description
clnt	Points to the structure of the client handle.
errp	Specifies the address of the error structure.

clnt_geterr Macro Exported from the libnsl Library

Purpose

Copies error information from a client handle.

Library

Network Services Library (**libnsl.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

Item	Description
clnt	Points to the structure of the client handle.
errp	Specifies the address of the error structure.

Examples

```
#include <stdlib.h>
#include <rpc/rpc.h>
#include <sys/time.h>

int main()
{
  rpcprog_t PROGNUM = 0x3fffffffL;
  rpcvers_t PROGVER = 0x1L;
  rpcproc_t procnum = 0x1L;
  CLIENT *clnt;
  enum clnt_stat cs;
  struct rpc_err client_error;
  char hostname[255] ; /* The Remote Host */
  char *nettype = "tcp";
  struct timeval total_timeout = {25,0};

  int input_arguments , output_results  ;

  if ((clnt=clnt_create(hostname, PROGNUM, PROGVER, nettype))==NULL)
  {
    fprintf(stderr,"clnt_create() subroutine failed");
    exit(1);
  }

  cs = clnt_call(clnt, procnum, (xdrproc_t)xdr_int,
           (char *)&input_arguments, (xdrproc_t)xdr_int,
           (char *)&output_results, total_timeout);

  if (cs != RPC_SUCCESS)
      clnt_geterr(clnt,&client_error);

  /* Destroy client handle in the end */
  clnt_destroy(clnt);

  return 0;
}
```

clnt_pcreateerror Subroutine

Important: The subroutine is exported from both the **libc** and the **libnsl** libraries.

clnt_pcreateerror Subroutine Exported from the libc Library

Purpose

Indicates why a client Remote Procedure Call (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 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

Item	Description
s	Points to a character string that represents the error text.

clnt_pcreateerror Subroutine Exported from the libnsl Library

Purpose

Prints an error message that is related to the creation of an RPC client handle to the standard error.

Library

Network Services Library (**libnsl.a**)

Syntax

```
#include <rpc/clnt.h>
void clnt_pcreateerror(error_msg)
const char * error_msg;
```

Description

The **clnt_pcreateerror** subroutine is used for error handling. The subroutine displays the actual cause of failure of the creation of an RPC client handle on the standard error. The actual error message, which is appended with a newline, is preceded by the string specified by *error_msg* parameter and a colon.

Note: If the *error_msg* parameter has a null value, the output is a colon followed by the actual error message.

Parameters

Item	Description
error_msg	Specified an error-message string that is provided by an application.

Examples

In the following example, the **clnt_create** subroutine tries to register a program number that is not valid and hence will return a null value. The **clnt_pcreateerror** subroutine returns the actual error message, which is preceded by the specified string ("Invalid Program Number") and a colon.

```
#include <rpc/clnt.h>
#include <stdio.h>

int main()
{
  CLIENT *cl;
  char hostname[255] ;  /* The name of remote host */
  char *nettype = "visible" ;
  rpcprog_t PROGNUM ; /* Invalid Value */
  rpcvers_t PROGVER ;

  cl = clnt_create(hostname, PROGNUM, PROGVER, nettype);

  if(cl==NULL)
  {
    clnt_pcreateerror("Invalid Program Number ");
    exit(1);
  }

  /*
  * Make a call to clnt_call() subroutine
  */

  /* Destroy the client handle at the end */
  clnt_destroy(cl);

  return 0;
}
```

Related reference:
"clntraw_create Subroutine" on page 266

clnt_perrno Subroutine

Important: The subroutine is exported from both the **libc** and the **libnsl** libraries.

clnt_perrno Subroutine Exported from the libc Library

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 **clnt_call** 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

Item	Description
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:

Item	Description
RPC_SUCCESS = 0	Call succeeded.
RPC_CANTENCODEARGS = 1	Cannot encode 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.

clnt_perrno Subroutine Exported from the libnsl Library

Purpose

Specifies the reason for failure of the procedure call.

Library

Network Services Library (**libnsl.a**)

Syntax

```
#include <rpc/rpc.h>
void clnt_perrno ( stat)
const 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 **clnt_call** 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, the **clnt_sperrno** subroutine is used instead of the **clnt_perrno** subroutine.

Parameters

Item	Description
stat	Specifies the client error status of the remote procedure call.

Error Codes

The following table list some error status codes that the **clnt_perrno** subroutine can translate and display. You can find a complete list of error codes in the **clnt_stat.h** file.

Item	Description
RPC_SUCCESS = 0	The call succeeded.
RPC_CANTENCODEARGS = 1	Arguments cannot be encoded.
RPC_CANTDECODERES = 2	Results cannot be decoded .
RPC_CANTSEND = 3	A failure occurred in sending call.
RPC_CANTRECV = 4	A failure occurred in receiving result.
RPC_TIMEDOUT = 5	The call timed out.

Examples

In the following example, the **clnt_perrno** subroutine displays the condition of the *cs* parameter.

```
#include <stdlib.h>
#include <rpc/rpc.h>
#include <sys/time.h>

int main()
{
  char hostname[255] ; /* The Remote host on which server is implemented */
  rpcprog_t program_number ;
  rpcvers_t version_number ;
   rpcproc_t procedure_number ;
  enum clnt_stat cs ;
  char *nettype = "visible";

  cs = rpc_call(hostname, program_number, version_number, procedure_number,
                (xdrproc_t)xdr_void, NULL, (xdrproc_t)xdr_void, NULL, nettype);
  if (cs != RPC_SUCCESS)
  {
    fprintf(stderr,"\n RPC Call failed\n");
    clnt_perrno(cs) ;
    exit(1);
  }

return 0 ;
}
```

clnt_perror Subroutine

Important: The subroutine is exported from both the **libc** and the **libnsl** libraries.

clnt_perror Subroutine Exported from the libc Library

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 preceded by the string pointed to by the *s* parameter and a colon.

This subroutine is used after the **clnt_call** macro.

Parameters

Item	Description
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.

clnt_perror Subroutine Exported from the libnsl Library

Purpose

Indicates why a remote procedure call failed.

Library

Network Services Library (**libnsl.a**)

Syntax

```
#include <rpc/rpc.h>

void clnt_perror ( clnt, s)
const CLIENT *clnt;
const char *s;
```

Description

The **clnt_perror** subroutine writes a message to standard error output indicating why a remote procedure call failed. The message is preceded by the string pointed to by the *s* parameter and a colon. The message is appended by a newline. This subroutine is used after the **clnt_call** macro.

Parameters

Item	Description
clnt	Points to the structure of the client handle.
s	Points to a character string that represents the error text.

Examples

In the following example, the **clnt_perror** subroutine displays the reason for failure of a remote procedure call.

```
#include <stdio.h>
#include <rpc/rpc.h>
int main()
{
  CLIENT *client ;
  char hostname[255] ; /* The Remote host on which server is implemented */
  rpcprog_t program_number = 0x3fffffffL;
  rpcvers_t version_number = 0x1L;
  rpcproc_t procedure_number = 0x1L;
  struct timeval total_timeout = { 25 , 0 } ;
  enum clnt_stat cs ;

  /* Create client handle */
  client = clnt_create(hostname, program_number, version_number, "tcp");
  if (client == (CLIENT *)NULL)
  {
    fprintf(stderr,"Couldn't create client\n");
    exit(1);
  }

  /* Make a call to remote procedure associated with client handle */
  cs = clnt_call(client, procedure_number, (xdrproc_t)xdr_void, NULL,
                   (xdrproc_t)xdr_void, NULL, total_timeout);
  if (cs != RPC_SUCCESS)
  {
    clnt_perror(client,"Client Call failed");
    exit(1);
  }

  /* Destroy client handle in the end */
  clnt_destroy(client);

  return 0;
}
```

clnt_raw_create Subroutine
Purpose

Creates and returns a generic client handle for the specified program and version.

Library

Network Services Library (**libnsl.a**)

Syntax

```
#include <rpc/rpc.h>

CLIENT * clnt_raw_create( prognum, versnum )
const rpcprog_t prognum;
const rpcvers_t versnum;
```

Description

The **clnt_raw_create** subroutine creates and returns a generic client handle for the specified program and version. For this subroutine, the server must be in the same address space as the client because the transport that is used for the client-server communication is the buffer in the process-address space of the client. This facilitates measurement of remote procedure call (RPC) overheads, such as round trip times, without any kernel or networking interference.

Parameters

Item	Description
prognum	Specifies the program number of the remote program.
versnum	Specifies the version number of the remote program.

Return Values

Item	Description
a generic client handle that is valid	successful
NULL	unsuccessful

Error Codes

The **clnt_raw_create** subroutine returns failure when one or more of the following error codes are true.

Item	Description
RPC_PROGNOTREGISTERED	The program number is not valid.
RPC_PROGVERSMISMATCH	The version number is not valid.

Examples

```
#include <stdlib.h>
#include <rpc/rpc.h>

int main()
{
    CLIENT *cl;
    rpcprog_t PROGNUM = 0x3fffffffL;
    rpcvers_t PROGVER = 0x1L ;

    /*
    * make the clnt_door_create call with this nettype and
    * observe the result
    */

    if ((cl=clnt_raw_create( PROGNUM, PROGVER ) == NULL)
    {
          fprintf(stdout, "clnt_raw_create : failed.\n");
          exit(EXIT_FAILURE);
    }

    /*
     * Make a call to clnt_call() subroutine
     */

    /* Destroy client handle in the end */
    clnt_destroy(cl);

    return 0;
}
```

Related information:

Transport Independent Remote Procedure Call, IPv6 concerns for Transport Independent Remote Procedure Call

eXternal Data Representation Overview for Programming

clnt_spcreateerror Subroutine

Important: The subroutine is exported from both the **libc** and the **libnsl** libraries.

clnt_spcreateerror Subroutine Exported from the libc Library

Purpose

Indicates why a client Remote Procedure Call (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 RPC handle was not created.

Note: This subroutine returns the pointer to static data that is overwritten on each call.

Parameters

Item	Description
s	Points to a character string that represents the error text.

clnt_spcreateerror Subroutine Exported from the libnsl Library

Purpose

Returns an error message that is related to the remote procedure call (RPC) client-handle creation.

Library

Network Services Library (**libnsl.a**)

Syntax

```
#include <rpc/rpc.h>

char * clnt_spcreateerror( error_msg );
const char *error_msg ;
```

Description

The **clnt_spcreateerror** subroutine is used for error handling. The subroutine displays the actual cause of failure of the creation of an RPC client handle. The actual error message is preceded by the string specified by *error_msg* parameter and a colon. However, the actual error message is not appended with a newline.

Note: If the *error_msg* parameter has a null value, the output is a colon followed by the actual error message.

Parameters

Item	Description
error_msg	Specified an error-message string that is provided by an application.

Example

In the following example, the **clnt_create** subroutine tries to register a program number that is not valid and hence returns a null value. The **clnt_spcreateerror** subroutine returns the actual error message, which is preceded by the specified string ("Invalid Program Number") and a colon.

```
#include <rpc/clnt.h>
#include <stdio.h>

int main()
{
  CLIENT *cl;
  char hostname[255] ;  /* The name of remote host */
  char *nettype = "visible" ;
  rpcprog_t PROGNUM ; /* Invalid Value */
  rpcvers_t PROGVER ;
  char *err_str;

  cl = clnt_create(hostname, PROGNUM, PROGVER, nettype);

  if(cl==NULL)
  {
    err_str = clnt_spcreateerror("Invalid Program Number ");
    printf("\n%s",err_str);
    exit(1);
  }

  /*
  * Make a call to clnt_call() subroutine
  */

  /* Destroy the client handle at the end */
  clnt_destroy(cl);

  return 0;
}
```

clnt_sperrno Subroutine

Important: The subroutine is exported from both the **libc** and the **libnsl** libraries.

clnt_sperrno Subroutine Exported from the libc Library

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 **clnt_call** 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

Item	Description
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:

Message	Description
RPC_SUCCESS = 0	Call succeeded.
RPC_CANTENCODEARGS = 1	Cannot encode 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.

clnt_sperrno Subroutine Exported from the libnsl Library

Purpose

Specifies the reason for failure of the procedure call.

Library

Network Services Library (**libnsl.a**)

Syntax

```
#include <rpc/rpc.h>
```

```
char *clnt_sperrno(stat)
const 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 **clnt_call** 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

Item	Description
stat	Specifies the client error status of the remote procedure call.

Error Codes

The following table list some error status codes that the **clnt_sperrno** subroutine can translate and display. You can find a complete list of error codes in the **clnt_stat.h** file.

Item	Description
RPC_SUCCESS = 0	The call succeeded.
RPC_CANTENCODEARGS = 1	Arguments cannot be encoded.
RPC_CANTDECODERES = 2	Results cannot be decoded .
RPC_CANTSEND = 3	A failure occurred in sending call.
RPC_CANTRECV = 4	A failure occurred in receiving result.
RPC_TIMEDOUT = 5	The call timed out.

Examples

In the following example, the **clnt_sperrno** subroutine returns the status message in the string pointed to by the *err_str* parameter.

```
#include <rpc/clnt.h>
#include <stdio.h>
#include <sys/time.h>

int main()
{
  char hostname[255] ; /* The Remote host on which server is implemented */
  rpcprog_t program_number ;
  rpcvers_t version_number ;
   rpcproc_t procedure_number ;
  enum clnt_stat cs ;
  char *nettype = "visible";
  char *err_str;

  cs = rpc_call(hostname, program_number, version_number, procedure_number, (xdrproc_t)xdr_void, NULL,
                   (xdrproc_t)xdr_void, NULL, nettype);
  if (cs != RPC_SUCCESS)
  {
    err_str = clnt_sperrno(cs) ;
```

```
    fprintf(stdout,"\n%s",err_str) ;
  }

  return 0 ;
}
```

clnt_sperror Subroutine

Important: The subroutine is exported from both the **libc** and the **libnsl** libraries.

clnt_sperror Subroutine Exported from the libc Library

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

Item	Description
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.

clnt_sperror Subroutine Exported from the libnsl Library

Purpose

Returns the error message indicating why a remote procedure call failed.

Library

Network Services Library (**libnsl.a**)

Syntax

```
#include <rpc/rpc.h>
```

```
char *clnt_sperror ( cl, s)
const CLIENT *cl;
const char *s;
```

Description

The **clnt_sperror** subroutine returns an error message indicating why a remote procedure call failed. The message is preceded by the string that is pointed to by the *s* parameter and a colon. The message is not appended by a newline. This subroutine is used after the **clnt_call** macro. The difference between the **clnt_sperror** and **clnt_perror** subroutines is that **clnt_perror** displays the error message on standard error whereas **clnt_sperror** just returns a pointer to the buffer that holds this error message.

Note: In a single thread, the subroutine uses the same buffer that is overwritten by the error message on successive calls. However, in multithreaded applications, the buffers used are thread-specific.

Parameters

Item	Description
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.

Examples

In the following example, the **clnt_sperror** subroutine returns the reason for failure of a remote procedure call that is pointed to by the *err_str* parameter.

```
#include <rpc/clnt.h>
#include <stdio.h>
#include <sys/time.h>

int main()
{
  CLIENT *client ;
  char hostname[255] ; /* The Remote host on which server is implemented */
  rpcprog_t program_number = 0x3fffffffL;
  rpcvers_t version_number = 0x1L;
  rpcproc_t procedure_number = 0x1L;
  struct timeval total_timeout = { 25 , 0 } ;
  enum clnt_stat cs ;
  char *err_str ;

  /* Create client handle */
  client = clnt_create(hostname, program_number, version_number, "tcp");
  if (client == (CLIENT *)NULL)
  {
    fprintf(stderr,"Couldn't create client\n");
    exit(1);
  }

  /* Call remote procedure associated with client handle */
  cs = clnt_call(client, procedure_number, (xdrproc_t)xdr_void, NULL,
                 (xdrproc_t)xdr_void, NULL, total_timeout);
  if (cs != RPC_SUCCESS)
  {
      err_str = clnt_sperror(client,"Client Call failed");
      fprintf(stderr,"%s",err_str);
      exit(1);
  }
```

```
  /* Destroy client handle in the end */
  clnt_destroy(client);

  return 0;
}
```

clnt_tli_create Subroutine

Purpose

Creates and returns a generic client handle for a remote program using the specified transport.

Library

Network Services Library (**libnsl.a**)

Syntax

```
#include <rpc/clnt.h>

CLIENT * clnt_tli_create(fd, nconf, svcaddr, prognum, versnum, sendsize, recvsize)
const int fd;
const struct netconfig *nconf;
const struct netbuf *svcaddr ;
const rpcprog_t prognum;
const rpcvers_t versnum;
const uint_t sendsize;
const uint_t recvsize;
```

Description

The **clnt_tli_create** subroutine is an expert-level API for transport-independent remote procedure calls (TI_PRC). The subroutine specifies transport-related parameters. The **clnt_tli_create** subroutine creates and returns a generic client handle for the specified program and version. This generic client handle is returned from the remote host. The subroutine uses an open and bound file descriptor through the specified transport and the address of the remote program to call the remote program. If you set the sizes of the send and receive buffers that are specified by the *sendsize* and *recvsize* parameters to 0, the default sizes of the buffers are used.

Note: If you set the value of the *nconf* parameter to a connection-oriented transport and set the value of the *svcaddr* parameter to NULL, the file descriptor is assumed to be connected. If you set the value of the *nconf* parameter to a connectionless transport and set the value of the *svcaddr* parameter to NULL, an error is returned. If you set the value of the *fd* parameter to RPC_ANYFD, a suitable file descriptor is opened and bound on the specified transport. If you set the value of the *fd* parameter to RPC_ANYFD and set the value of the *nconf* parameter to NULL, an error is returned.

Parameters

Item	Description
fd	Specifies the file descriptor that is open, bound, and connected on the specified transport.
nconf	Specifies the transport to use.
svcaddr	Specifies the address of the remote program
prognum	Specifies the program number of the remote program.
versnum	Specifies the version number of the remote program.
sendsize	Specifies the size of the send buffer.
recvsize	Specifies the size of the receive buffer.

Return Values

Item	Description
a generic client handle that is valid	successful
NULL	unsuccessful

You can use the **clnt_pcreateerror** subroutine to obtain the reason for failure.

Error Codes

The **clnt_tli_create** subroutine returns failure if one or more of the following are true.

Item	Description
RPC_TLIERROR	The file descriptor is not valid.
RPC_UNKNOWNADDR	The value of the *svcaddr* parameter that holds the address of the remote program is NULL.

Examples

In the following example, the **clnt_tli_create** subroutine returns a generic client handle for the remote program using the specified transport on successful completion.

```
int main()
{
    char hostname[255]; /* The Remote Host */
    rpcprog_t PROGNUM = 0x3fffffffL ;
    rpcvers_t PROGVER = 0x1L ;
    struct netconfig *nconf
    struct netbuf svcaddr ;
    CLIENT *cl ;
    char *transport ;            /* Can be set to TCP or UDP */

    if ((nconf = getnetconfigent(transport)) == (struct netconfig *)NULL)
    {
          fprintf(stderr, "Cannot get netconfig entry for UDP\n");
          exit(2);
    }

    if (!rpcb_getaddr(PROGNUM, PROGVER, nconf, &svcaddr, hostname))
    {
          fprintf(stderr, "rpcb_getaddr failed!!\n");
          exit(2);
    }

    /*
     * make the clnt_tli_create call with nconf and
     * observe the result
     */
    cl = clnt_tli_create(RPC_ANYFD, nconf, &svcaddr, PROGNUM, PROGVER, 0, 0);
    if( cl==NULL )
    {
          fprintf(stdout, "clnt_tli_create : failed.\n");
          exit(EXIT_FAILURE);
    }

    /*
     * Make a call to clnt_call() subroutine
     */

    /* Destroy the client handle at the end */
    clnt_destroy(cl);

    return 0;
}
```

Related reference:

"clnt_tp_create Subroutine"

Related information:

Transport Independent Remote Procedure Call, IPv6 concerns for Transport Independent Remote Procedure Call

eXternal Data Representation Overview for Programming

clnt_tp_create Subroutine

Purpose

Creates a client handle for a remote program using the specified class of transport.

Library

Network Services Library (**libnsl.a**)

Syntax

```
#include <rpc/rpc.h>

CLIENT * clnt_tp_create(host, prognum, versnum, nconf)
const char *host;
const rpcprog_t prognum;
const rpcvers_t versnum;
const struct netconfig *nconf;
```

Description

The **clnt_tp_create** subroutine is an intermediate-level API. The subroutine enables the application to have a better control over the transport service to be used. The subroutine creates a client handle for the specified program and version. This client handle is created and returned from a remote host where the server is located. The operation is done with a transport service that is specified by *nconf* parameter. The service can be a connection-oriented or a connectionless service.

Note: If you set the value of the *nconf* parameter to NULL, the subroutine fails.

Parameters

Item	Description
host	Specifies the host name where the server resides.
prognum	Specifies the program number of the remote program.
versnum	Specifies the version number of the remote program.
nconf	Defines a specific transport service to use.

Return Values

Item	Description
a generic client handle that is valid	successful
NULL	unsuccessful

You can use the **clnt_pcreateerror** subroutine to obtain the reason for failure.

Error Codes

The **clnt_tp_create** subroutine returns failure if one or more of the following are true.

Item	Description
RPC_UNKNOWNPROTO	• The transport service that is specified by the *nconf* parameter is NULL. • The value of the *nconf* parameter is set to **netpath**, and the NETPATH environment variable is set to a transport service that is not valid.
RPC_UNKNOWNHOST	The host name is not valid.
RPC_TLIERROR	The value of the *nconf* parameter is set to **udp** and a call to the **clnt_tp_create** subroutine is made from the client program with the *nconf* member (`nc_semantics = NC_TPI_COTS_ORD`).
RPC_PROGNOTREGISTERED	The program number is not valid.

Examples

In the following example, the **clnt_tp_create** subroutine returns a generic client handle using the **tcp** transport service on successful completion.

```
#include <stdlib.h>
#include <rpc/rpc.h>

int main()
{
    CLIENT *client;
    struct netconfig *nconf;
    char hostname[255] ; /* The Remote host where server is located */
    rpcprog_t PROGNUM = 0x3fffffffL;
    rpcvers_t PROGVER = 0x1L;

    /* getnetconfigent() returns a pointer to the struct  netconfig
     * structure  corresponding to tcp transport
     */
    if ((nconf = getnetconfigent("tcp")) == (struct netconfig *)NULL)
    {
        fprintf(stderr, "Cannot get netconfig entry for UDP\n");
        exit(2);
     }

    /* Create client handle using clnt_tp_create() */
    client = clnt_tp_create(hostname, PROGNUM, PROGVER, nconf);
    if (client == (CLIENT *)NULL)
    {
        fprintf(stderr,"Couldn't create client at inter lvl\n");
        clnt_pcreateerror("Inter lvl : ");
        exit(EXIT_FAILURE);
    }

    /*
    * Make a call to clnt_call() subroutine
    */

    /* Destroy client handle in the end */
    clnt_destroy(client);

    return 0 ;
}
```

Related reference:

"clnt_create Subroutine" on page 227

"clnt_tli_create Subroutine" on page 258

Related information:

Transport Independent Remote Procedure Call, IPv6 concerns for Transport Independent Remote Procedure Call

eXternal Data Representation Overview for Programming

clnt_tp_create_timed Subroutine
Purpose

Creates a client handle for a remote program using the specified class of transport within the specified timeout.

Library

Network Services Library (**libnsl.a**)

Syntax

```
#include <rpc/rpc.h>

CLIENT * clnt_tp_create_timed(host, prognum, versnum, nconf, timeout)
const char *host;
const rpcprog_t prognum;
const rpcvers_t versnum;
const struct netconfig *nconf;
const struct timeval *timeout;
```

Description

The **clnt_tp_create_timed** subroutine is an intermediate level API. The subroutine enables the application to have a better control over the transport service to be used. The subroutine creates a client handle for the specified program and version. This client handle is created and returned from the remote host where server is located. The operation is done with a transport service specified by *nconf* parameter. The service can be a connection-oriented or connectionless service. The *timeout* parameter specifies the time duration within which the subroutine returns. If the timeout value expires, the subroutine fails and returns NULL.

Note: If you set the value of the *nconf* parameter to NULL, the subroutine fails.

Parameters

Item	Description
host	Specifies the host name where the server resides.
prognum	Specifies the program number of the remote program.
versnum	Specifies the version number of the remote program.
nconf	Defines a specific transport service to use.
timeout	Specifies the timeout value.

Return Values

Item	Description
a generic client handle that is valid	successful
NULL	unsuccessful

You can use the **clnt_pcreateerror** subroutine to obtain the reason for failure.

Error Codes

The **clnt_tp_create_timed** subroutine returns failure if one or more of the following are true.

Item	Description
RPC_UNKNOWNPROTO	• The transport service specified by the *nconf* parameter is NULL. • The value of the *nconf* parameter is set to **netpath**, and the NETPATH environment variable is set to a transport service that is not valid.
RPC_UNKNOWNHOST	The host name is not valid.
RPC_TLIERROR	The value of the *nconf* parameter is set to **udp** and a call to the **clnt_tp_create** subroutine is made from the client program with *nconf* member (`nc_semantics = NC_TPI_COTS_ORD`).
RPC_PROGNOTREGISTERED	The program number is not valid.
RPC_TIMEDOUT	The timeout value has expired.

Examples

In the following example, the **clnt_tp_create_timed** subroutine returns a generic client handle using the **tcp** transport service on successful completion. If the timeout value expires, the subroutine returns NULL.

```
#include <stdlib.h>
#include <rpc/rpc.h>

int main()
{
    CLIENT *client;
    struct netconfig *nconf;
    char hostname[255]; /* The Remote host where server is located */
    rpcprog_t PROGNUM = 0x3fffffffL;
    rpcvers_t PROGVER = 0x1L;
    const struct timeval timeout = { 25 , 0 } ;

    /* getnetconfigent() returns a pointer to the struct  netconfig
     * structure  corresponding to tcp transport
     */
    if ((nconf = getnetconfigent("tcp")) == (struct netconfig *)NULL)
    {
        fprintf(stderr, "Cannot get netconfig entry for UDP\n");
        exit(2);
    }

    client = clnt_tp_create_timed(hostname, PROGNUM, PROGVER, nconf, &timeout);
    if (client == (CLIENT *)NULL)
    {
         fprintf(stderr,"Couldn't create client at inter lvl\n");
         clnt_pcreateerror("Inter lvl : ");
         exit(EXIT_FAILURE);
    }

    /*
     * Make a call to clnt_call() subroutine
     */

    /* Destroy client handle in the end */
    clnt_destroy(client);

    return 0 ;
}
```

Related reference:

"clnt_create_timed Subroutine" on page 230

Related information:

Transport Independent Remote Procedure Call, IPv6 concerns for Transport Independent Remote Procedure Call

eXternal Data Representation Overview for Programming

clnt_vc_create Subroutine

The **clnt_vc_create** subroutine is a bottom-level API for transport-independent remote procedure calls (TI_PRC). With the subroutine, applications can control all the options. The **clnt_vc_create** subroutine creates and returns a generic client handle for the specified program and version. The subroutine uses a connection-oriented transport. The generic client handle is returned from the remote host where the server is located. The subroutine uses an open and bound file descriptor through the connection-oriented transport and the specified address of the remote program to call the remote program. The can be specified by the *sendsize* and *recvsize* parameters. If you set the sizes of the send and receive buffers that are specified by the *sendsize* and *recvsize* parameters to 0, the default sizes of the buffers are used.

Note: If you set the value of the *fd* parameter to RPC_ANYFD or set the *svcaddr* subroutine to a null value, the subroutine fails and returns a null value.

Purpose

Creates and returns a generic client handle for a remote program using a connection-oriented transport.

Library

Network Services Library (**libnsl.a**)

Syntax

```
#include <rpc/rpc.h>

CLIENT * clnt_vc_create(fd, svcaddr, prognum, versnum, sendsize, recvsize)
int  fd;
const struct netbuf * svcaddr;
const rpcprog_t prognum;
const rpcvers_t versnum;
const  uint_t sendsize;
const  uint_t recvsize
```

Description

Parameters

Item	Description
fd	Specifies the open and bound file descriptor on a connection-oriented transport.
svcaddr	Specifies the address of the remote program.
prognum	Specifies the program number of the remote program.
versnum	Specifies the version number of the remote program.
sendsize	Specifies the size of the send buffer.
recvsize	Specifies the size of the receive buffer.

Return Values

Item	Description
a generic client handle that is valid	successful
a null value	unsuccessful

You can use the **clnt_pcreateerror** subroutine to obtain the reason for failure.

Error Codes

The **clnt_vc_create** subroutine returns failure if one or more of the following codes are true.

Item	Description
RPC_TLIERROR	The file descriptor is not valid.
RPC_UNKNOWNADDR	The value of the *svcaddr* parameter that holds the address of the remote program is a null value.
RPC_CANTENCODEARGS	The size of the send or receive buffer is less than that of the sent packet.

Examples

```
#include <stdlib.h>
#include <rpc/rpc.h>

int main()
{
    CLIENT *cl ;
    int fd;
    rpcprog_t PROGNUM    =          0x3fffffffL;
    rpcvers_t PROGVER        =      0x1L ;
    struct netconfig *nconf ;
    struct netbuf svcaddr ;
    char hostname[255] ;      /* The name of remote host */

    /* getnetconfigent() returns a pointer to the struct  netconfig
     * structure  corresponding to tcp transport
     */
    if ((nconf = getnetconfigent("tcp")) == (struct netconfig *)NULL)
    {
          fprintf(stderr, "Cannot get netconfig entry for UDP\n");
          exit(2);
    }

    /* Get address of service on remote host */
    if (!rpcb_getaddr(PROGNUM, PROGVER, nconf,
                    &svcaddr, hostname))
    {
        fprintf(stderr, "rpcb_getaddr failed!!\n");
        exit(2);
     }
    /* Get the open and bound file descriptor for connection oriented transport */
    fd = . . .
    if ((cl = clnt_vc_create(fd, &svcaddr,
              PROGNUM, PROGVER, 0, 0))==NULL);
     {
         fprintf(stdout, "clnt_vc_create : failed.\n");
         exit(EXIT_FAILURE);
     }

    /*
     * Make a call to clnt_call() subroutine
     */

    /* Destroy client handle in the end */
    clnt_destroy(cl);

    return 0;
}
```

Related information:

Transport Independent Remote Procedure Call, IPv6 concerns for Transport Independent Remote Procedure Call

eXternal Data Representation Overview for Programming

clntraw_create Subroutine
Purpose

Creates a toy Remote Procedure Call (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 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

Item	Description
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.

Related reference:

"clnt_pcreateerror Subroutine" on page 244

"svcraw_create Subroutine" on page 396

Related information:

List of RPC Programming References

Remote Procedure Call (RPC) Overview for Programming

clnttcp_create Subroutine

Important: The subroutine is exported from both the **libc** and the **libnsl** libraries.

clnttcp_create Subroutine Exported from the libc Library

Purpose

Creates a Transmission Control Protocol/Internet Protocol (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 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 **clnttcp_create** subroutine picks suitable default values.

Parameters

Item	Description
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.

clnttcp_create Subroutine Exported from the libnsl Library

Purpose

Creates a Transmission Control Protocol/Internet Protocol (TCP/IP) client transport handle.

Library

Network Services Library (**libnsl.a**)

Syntax

```
#include <rpc/rpc.h>
CLIENT *clnttcp_create (addr, prognum, versnum, fdp, sendsz, recvsz)
struct sockaddr_in *addr;
rpcprog_t prognum;
rpcvers_t versnum;
int *fdp;
uint_t sendsz, recvsz;
```

Description

The **clnttcp_create** subroutine creates a Remote Procedure Call (RPC) client transport handle for a remote program. This client uses 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 **clnttcp_create** subroutine picks suitable default values.

Parameters

Item	Description
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.
fdp	Specifies a pointer to a socket. If the value of the *fdp* parameter is **RPC_ANYSOCK**, the **clnttcp_create** subroutine opens a new socket and sets the *fdp* pointer to the new socket.
sendsz	Sets the size of the send buffer.
recvsz	Sets the size of the receive buffer.

Return Values

Item	Description
a valid TCP/IP client handle	successful
a null value	unsuccessful

Error Codes

Item	Description
RPC_PROGNOTREGISTERED	The program is not registered.
RPC_SYSTEMERROR	The file descriptor is not valid.

Examples

In the following example, the **clnttcp_create** subroutine creates and returns a TCP/IP client transport handle.

```
#include <rpc/rpc.h>
#include <stdio.h>

#define ADDRBUFSIZE 255

int main()
{
  CLIENT *clnt;
  rpcprog_t  PROGNUM = 0x3fffffffL;
  rpcvers_t  PROGVER = 0x1L;
  int    fd;
  uint_t  sendsz=0, recvsz=0;
  struct sockaddr_in addr;
  char       addrbuf[ADDRBUFSIZE];
  struct netbuf   svcaddr;
  struct netconfig *nconf;
  char host[255] ; /* The remote host name */

  svcaddr.len = 0;
  svcaddr.maxlen = ADDRBUFSIZE;
  svcaddr.buf = addrbuf;

  /* Get pointer to struct netconfig for tcp transport */
  nconf = getnetconfigent("tcp");
  if (nconf == (struct netconfig *) NULL) {
    printf("getnetconfigent() failed\n");
```

```
    exit(1);
  }

  /* Get the address of remote service */
  if (!rpcb_getaddr(PROGNUM, PROGVER, nconf, &svcaddr, host)) {
    printf("rpcb_getaddr() failed\n");
    exit(1);
  }
  memcpy(&addr, svcaddr.buf, sizeof(struct sockaddr_in));

  fd = ... /*   Code to obtain open and bound file descriptor on tcp transport  */

  clnt = (CLIENT *) clnttcp_create(&addr, PROGNUM, PROGVER, &fd, sendsz, recvsz);

  /*
   * Make a call to clnt_call() subroutine
   */

  /* Destroy the client handle in the end */
  clnt_destroy(clnt);

  return 0;
}
```

clntudp_bufcreate Subroutine

Purpose

Creates a User Datagram Protocol/Internet Protocol (UDP/IP) client transport handle with specified maximum packet size for UDP-based remote procedure call (RPC) messages.

Library

Network Services Library (**libnsl.a**)

Syntax

```
#include <rpc/rpc.h>

CLIENT *clntudp_bufcreate( addr, prognum, versnum, wait, fdp, sendsz, recvsz)
struct sockaddr_in * addr;
rpcprog_t prognum;
rpcvers_t versnum;
struct timeval wait;
int * fdp;
uint_t sendsz;
uint_t recvsz;
```

Description

The **clntudp_bufcreate** subroutine creates an RPC client transport handle for a remote program. The client uses UDP as the transport to pass messages to the service. The remote program is located at the internet address specified by the *addr* parameter. The *fdp* parameter represents the open and bound file descriptor. If it is set to RPC_ANYSOCK, the subroutine opens a new file descriptor and binds it to the UDP transport. The *sendsz* and *recvsz* parameters specify the size of send and receive buffers used for sending and receiving UDP-based RPC messages.

Parameters

Item	Description
addr	Points to the Internet address of the remote program. If the port number for this Internet address (**addr->sin_port**) is 0, the value of the *addr* parameter is set to the port that the remote program is listening on. The **clntudp_bufcreate** 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. You can use the **clnt_call** macro to set the total time for the call to time out.
fdp	Specifies the open and bound file descriptor. If the value of the *fdp* parameter is RPC_ANYSOCK, the **clntudp_bufcreate** subroutine opens a new file descriptor and sets the *fdp* pointer to the new file descriptor.
sendsz	Specifies the size of the send buffer.
recvsz	Specifies the size of the receive buffer.

Return Values

Item	Description
a valid UDP client handle	successful
a null value	unsuccessful

Error Codes

Item	Description
RPC_PROGNOTREGISTERED	The program is not registered.
RPC_SYSTEMERROR	The file descriptor is not valid.

Examples

In the following example, the **clntudp_bufcreate** subroutine creates and returns a UDP/IP client transport handle.

```
#include <stdlib.h>
#include <rpc/rpc.h>
#define ADDRBUFSIZE 255

int main()
{
    CLIENT *clnt;
    rpcprog_t PROGNUM = 0x3fffffffL;
    rpcvers_t PROGVER = 0x1L;
    int fd;
    struct timeval waittime = {25,0};
    struct sockaddr_in addr;
    char addrbuf[ADDRBUFSIZE];
    struct netbuf svcaddr;
    struct netconfig *nconf;
    uint_t sendsz, recvsz;
    char host[255] ; /* The remote host name */

    svcaddr.len = 0;
    svcaddr.maxlen = ADDRBUFSIZE;
    svcaddr.buf = addrbuf;

    /* Get pointer to struct netconfig for tcp transport */
    nconf = getnetconfigent("udp");
    if (nconf == (struct netconfig *) NULL) {
        printf("getnetconfigent() failed\n");
        exit(1);
    }
```

```
    /* Get the address of remote service */
    if (!rpcb_getaddr(PROGNUM, PROGVER, nconf, &svcaddr, host)) {
        printf("rpcb_getaddr() failed\n");
        exit(1);
    }
    memcpy(&addr, svcaddr.buf, sizeof(struct sockaddr_in));

    fd = ... /*    Code to obtain open and bound file descriptor on udp transport */

    clnt = (CLIENT *) clntudp_bufcreate(&addr, PROGNUM, PROGVER, waittime, &fd, sendsz, recvsz);

    /*
     * Make a call to clnt_call() subroutine
     */

    /* Destroy the client handle in the end */
    clnt_destroy(clnt);

    return 0;
}
```

Related information:

Transport Independent Remote Procedure Call

eXternal Data Representation Overview for Programming

clntudp_create Subroutine

Important: The subroutine is exported from both the **libc** and the **libnsl** libraries.

clntudp_create Subroutine Exported from the libc Library

Purpose

Creates a User Datagram Protocol/Internet Protocol (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 UDP as the transport to pass messages to the service.

RPC messages transported by UDP/IP can hold up to 8KB of encoded data. Use this subroutine for procedures that take arguments or return results of less than 8KB.

Parameters

Item	Description
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.

clntudp_create Subroutine Exported from the libnsl Library

Purpose

Creates a User Datagram Protocol/Internet Protocol (UDP/IP) client transport handle.

Library

Network Services Library (**libnsl.a**)

Syntax

```
#include <rpc/rpc.h>

CLIENT *clntudp_bufcreate (addr, prognum, versnum, wait, fdp)
struct sockaddr_in *addr;
rpcprog_t prognum;
rpcvers_t versnum;
struct timeval wait;
int *fdp;
```

Description

The **clntudp_create** subroutine creates a Remote Procedure Call (RPC) client transport handle for a remote program. The client uses UDP as the transport to pass messages to the service.

RPC messages transported by UDP/IP can hold up to 8KB of encoded data. Use this subroutine for procedures that take arguments or return results of less than 8KB.

Parameters

Item	Description
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.
fdp	Specifies a pointer to a socket. If the value of the *fdp* parameter is **RPC_ANYSOCK**, the **clnttcp_create** subroutine opens a new socket and sets the *fdp* pointer to the new socket.

Return Values

Item	Description
a valid UDP client handle	successful
a null value	unsuccessful

Error Codes

Item	Description
RPC_PROGNOTREGISTERED	The program is not registered.
RPC_SYSTEMERROR	The file descriptor is not valid.

Examples

In the following example, the **clntudp_create** subroutine creates and returns a UDP/IP client transport handle.

```
#include <rpc/rpc.h>
#include <stdio.h>

#define ADDRBUFSIZE 255
#define ADDRBUFSIZE 255

int main()
{
  CLIENT *clnt;
  rpcprog_tPROGNUM = 0x3fffffffL;
  rpcvers_tPROGVER = 0x1L;
  intfd;
  struct timeval waittime = {25,0};
  struct sockaddr_in addr;
  char     addrbuf[ADDRBUFSIZE];
  struct netbuf   svcaddr;
  struct netconfig *nconf;
  char host[255] ; /* The remote host name */

  svcaddr.len = 0;
  svcaddr.maxlen = ADDRBUFSIZE;
  svcaddr.buf = addrbuf;
  /* Get pointer to struct netconfig for tcp transport */
  nconf = getnetconfigent("udp");
  if (nconf == (struct netconfig *) NULL) {
    printf("getnetconfigent() failed\n");
    exit(1);
  }
  /* Get the address of remote service */
  if (!rpcb_getaddr(PROGNUM, PROGVER, nconf, &svcaddr, host)) {
    printf("rpcb_getaddr() failed\n");
    exit(1);
  }
```

```
  memcpy(&addr, svcaddr.buf, sizeof(struct sockaddr_in));
  fd = ... /*Code to obtain open and bound file descriptor on udp transport */

  clnt = (CLIENT *) clntudp_create(&addr, PROGNUM, PROGVER, waittime, &fd);
  /*
   * Make a call to clnt_call() subroutine
   */
  /* Destroy the client handle in the end */
  clnt_destroy(clnt);

  return 0;
}
```

g

The following RPC subroutines begin with the letter g.

get_myaddress Subroutine

Purpose

Gets the user's Internet Protocol (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 IP address without consulting the library routines that access the **/etc/hosts** file.

Parameters

Item	Description
addr	Specifies the address where the machine's IP address is placed. The port number is set to a value of **htons** (PMAPPORT).

Related information:

/etc/hosts subroutine

List of RPC Programming References

Internet Protocol

Remote Procedure Call (RPC) Overview for Programming

getnetname Subroutine

Important: The subroutine is exported from both the **libc** and the **libnsl** libraries.

getnetname Subroutine Exported from the libc Library

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.

Parameters

Item	Description
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.

getnetname Subroutine Exported from the libnsl Library

Purpose

Generates the operating-system-independent network name of the caller.

Library

Network Services Library (**libnsl.a**)

Syntax

```
#include <rpc/rpc.h>
getnetname ( name)
char *name;
```

Description

The **getnetname** subroutine, which belongs to the secure RPC category, is used in applications that use the **AUTH_DES** authentication flavor. This subroutine generates the network name (or netname) of the caller. If the caller has root authority, the net name of the host machine is generated.

Parameters

Item	Description
name	Represents the network name of the caller.

Return Values

Item	Description
1	successful
0	unsuccessful

Examples

```
#include <rpc/rpc.h>
int main()
{
  char name[255]; /* contains netname of owner of server process */
  char rhost[255]; /* Remote host name on which server resides */
  rpcprog_t  PROGNUM = 0x3fffffffL;
  rpcvers_t  PROGVER = 0x1L;

  if(!getnetname(name))
  {
    fprintf(stderr,"getnetname() error\n");
    exit(1);
  }

  /* Create a client handle for remote host rhost for PROGNUM & PROGVER on tcp transport */
  clnt = clnt_create(rhost, PROGNUM, PROGVER, "tcp");
  if (clnt == (CLIENT *) NULL) {
    fprintf(stderr,"client_create() error\n");
    exit(1);
  }

  clnt->cl_auth = authdes_seccreate(name, 80, rhost, (des_block *)NULL);

  /*
   * Make a call to clnt_call() subroutine
   */

  /* Destroy the authentication handle */
  auth_destroy(clnt->cl_auth);

  /* Destroy the client handle in the end */
  clnt_destroy(clnt);

  return 0;
}
```

host2netname Subroutine

Important: The subroutine is exported from both the **libc** and the **libnsl** libraries.

host2netname Subroutine Exported from the libc Library

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

Item	Description
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.

host2netname Subroutine Exported from the libnsl Library

Purpose

Converts a domain-specific host name to an operating-system-independent network name.

Library

Network Services Library (**libnsl.a**)

Syntax

```
#include <rpc/rpc.h>
int host2netname( name, host, domain)
char *name;
const char *host;
const char *domain;
```

Description

The **host2netname** subroutine, which belongs to the secure remote procedure call (RPC) category, is used in applications that use the **AUTH_DES** authentication flavor. This subroutine is generally used on the client side to generate network name (or netname) of the host on which the server program resides and to which the client needs to contact using **AUTH_DES** authentication flavor.

Note: When the *domain* parameter is set to a null value, the **host2netname** subroutine uses the default domain name of the machine. When the host parameter is set to a null value, the subroutine is the inverse of the **netname2host** subroutine.

Parameters

Item	Description
name	Represents the network name of the host after successful completion.
host	Specifies the domain-specific host name.
domain	Specifies the domain name of the host.

Return Values

Item	Description
1	successful
0	unsuccessful

Examples

```
#include <rpc/rpc.h>
int main()
{
  char name[255]; /* contains netname of owner of server process */
  char host[255]; /* Remote host name on which server resides */
  char  domain[255];
  rpcprog_t PROGNUM = 0x3fffffffL;
  rpcvers_t PROGVER = 0x1L;

  /* obtain the domainname of the host */
  if (getdomainname(domain, 255)) {
    fprintf(stderr, "\ngetdomainname() failed\n");
    exit(2);
  }

  /* Obtain network name of remote host */
  if (!host2netname(name, host, domain))
  {
    fprintf(stderr, "\nhost2netname() failed\n");
    exit(EXIT_FAILURE);
  }

  /* Create a client handle for remote host rhost for PROGNUM & PROGVER on tcp transport */
  clnt = clnt_create(host, PROGNUM, PROGVER, "tcp");
  if (clnt == (CLIENT *) NULL) {
    fprintf(stderr,"client_create() error\n");
    exit(1);
  }

  clnt->cl_auth = authdes_seccreate(name, 80, host, (des_block *)NULL);

  /*
   * Make a call to clnt_call() subroutine
   */

  /* Destroy the authentication handle */
  auth_destroy(clnt->cl_auth);

  /* Destroy the client handle in the end */
  clnt_destroy(clnt);

  return 0;
}
```

k

The following RPC subroutines begin with the letter k.

key_decryptsession Subroutine

Important: The subroutine is exported from both the **libc** and the **libnsl** libraries.

key_decryptsession Subroutine Exported from the libc Library

Purpose

Decrypts a server network name and a Data Encryption Standard (DES) key.

Library

C Library (**libc.a**)

Syntax

```
#include <rpc/rpc.h>

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 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

Item	Description
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.

key_decryptsession Subroutine Exported from the libnsl Library

Purpose

Decrypts the Data Encryption Standard (DES) key.

Library

Network Services Library (**libnsl.a**)

Syntax

```
#include <rpc/rpc.h>
```

```
int key_decryptsession ( remotename, deskey)
const char *remotename;
des_block *deskey;
```

Description

The **key_decryptsession** subroutine, which belongs to the secure remote procedure call (RPC) category, is an interface subroutine to the **keyserver** daemon. 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.

Note: This subroutine is the inverse of the **key_encryptsession** subroutine. You must run the **keyserv** daemon to enable this subroutine.

Parameters

Item	Description
remotename	Specifies the remote host name.
deskname	Specifies the DES key.

Return Values

Item	Description
0	successful
-1	unsuccessful

Examples

```
#include <rpc/rpc.h>
int main()
{
  des_block dblock;
  char name[MAXNETNAMELEN + 1]; /* contains netname of owner of server process */
  char rhost[255]; /* The Remote host */

  /* Obtain network name of remote host */
  if (!host2netname(name, rhost, NULL))
  {
    fprintf(stderr,"\nhost2netname() failed\n");
    exit(1);
  }

  if (key_decryptsession(name, &dblock)!=0) {
    fprintf(stderr,"\nkey_decryptsession() failed\n");
    exit(1);
  }
  return 0;
}
```

key_encryptsession Subroutine

Important: The subroutine is exported from both the **libc** and the **libnsl** libraries.

key_encryptsession Subroutine Exported from the libc Library

Purpose

Encrypts a server network name and a Data Encryption Standard (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 known as 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

Item	Description
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.

key_encryptsession Subroutine Exported from the libnsl Library

Purpose

Encrypts the Data Encryption Standard (DES) key.

Library

Network Services Library (**libnsl.a**)

Syntax

```
#include <rpc/rpc.h>
int key_encryptsession ( remotename,  deskey)
char *remotename;
des_block *deskey;
```

Description

The **key_encryptsession** subroutine, which belongs to the secure remote procedure call (RPC) category, is an interface subroutine to the **keyserver** daemon. The subroutine takes a server network name and a DES key and encrypts the key by using the public key of the server and the secret key associated with the effective UID of the calling process. However, user programs rarely need to call this subroutine.

Note: This subroutine is the inverse of the **key_decryptsession** subroutine. You must run the **keyserv** daemon to enable this subroutine.

Parameters

Item	Description
remotename	Specifies the remote host name.
deskname	Specifies the DES key.

Return Values

Item	Description
0	successful
-1	unsuccessful

Examples

```
#include <rpc/rpc.h>

int main()
{
  des_block dblock;
  char name[255]; /* contains netname of owner of server process */
  char rhost[255]; /* The Remote host */

  /* Obtain network name of remote host */
  if (!host2netname(name, rhost, NULL))
  {
    fprintf(stderr,"\nhost2netname() failed\n");
    exit(1);
  }

  strcpy(dblock.c, "deskey");
  if (key_encryptsession(name, &dblock) != 0) {
    fprintf(stderr, "\nkey_encryptsession() failed\n");
    exit(1);
  }

  return 0;
}
```

key_gendes Subroutine

Important: The subroutine is exported from both the **libc** and the **libnsl** libraries.

key_gendes Subroutine Exported from the libc Library

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 known 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

Item	Description
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.

key_gendes Subroutine Exported from the libnsl Library

Purpose

Gets a secure conversation key from the **keyserver** daemon.

Library

Network Services Library (**libnsl.a**)

Syntax

```
#include <rpc/rpc.h>

int key_gendes ( deskey)
des_block *deskey;
```

Description

The **key_gendes** subroutine, which belongs to the secure remote procedure call (RPC) category, is an interface subroutine to the **keyserver** daemon. The conversation key that is used to encrypt the timestamp is usually chosen at random. However, choosing a key at random is not suggested because the common ways of choosing random key, such as the current time, are not secure. Therefore, the **key_gendes** subroutine asks the **keyserver** daemon for a secure conversation key.

Note: The **keyserv** daemon must be running for this subroutine to work.

Parameters

Item	Description
deskey	Specifies the secure conversation key after successful completion.

Return Values

Item	Description
0	successful
-1	unsuccessful

Examples

```
#include <rpc/rpc.h>
int main()
{
  char    name[255]; /* contains netname of owner of server process */
  char    rhost[255]; /* Remote host name on which server resides */
  rpcprog_t  PROGNUM = 0x3fffffffL;
  rpcvers_t  PROGVER = 0x1L;
  CLIENT  *clnt;
  des_block  dblock;

  /* Obtain network name of remote host */
  if (!host2netname(name, rhost, NULL))
  {
    fprintf(stderr, "\nhost2netname() failed\n");
    exit(EXIT_FAILURE);
  }

  if (key_gendes(&dblock) == -1) {
    fprintf(stderr, "\nkey_gendes() failed\n");
    exit(EXIT_FAILURE);
  }

  /* Create a client handle for remote host rhost
   *for PROGNUM & PROGVER on tcp transport
   */
  clnt = clnt_create(rhost, PROGNUM, PROGVER, "tcp");
  if (clnt == (CLIENT *) NULL) {
    fprintf(stderr,"client_create() error\n");
    exit(1);
  }

  clnt->cl_auth = authdes_seccreate(name, 80, rhost, &dblock);

  /*
   * Make a call to clnt_call() subroutine
   */

  /* Destroy the authentication handle */
  auth_destroy(clnt->cl_auth);

  /* Destroy the client handle in the end */
  clnt_destroy(clnt);

  return 0;
}
```

key_secretkey_is_set Subroutine
Purpose

Determines whether a key is set for the effective UID of the calling process.

Library

Network Services Library (**libnsl.a**)

Syntax

```
#include <rpc/rpc.h>
int key_secretkey_is_set (void)
```

Description

The **key_secretkey_is_set** subroutine belongs to the secure remote procedure call (RPC) category. The subroutine is an interface subroutine to the **keyserver** daemon. The **keylogin** command fetches the key for the effective UID of the calling process and stores the key in the **keyserv** daemon. This subroutine is thus used to determine whether the **keyserv** daemon contains the key for the effective UID of the calling process.

Return Values

Item	Description
1	The key is stored in the **keyserver** daemon.
0	The key is not stored in the **keyserver** daemon.

Examples

```
#include <rpc/rpc.h>
int main()
{
  if (key_secretkey_is_set() != 1) {
  fprintf(stderr,"key_secretkey_is_set() failed");
  exit(1);
  }

  return 0;
}
```

Related information:

keylogin subroutine

keyserv subroutine

Transport Independent Remote Procedure Call

eXternal Data Representation Overview for Programming

key_setsecret Subroutine

Important: The subroutine is exported from both the **libc** and the **libnsl** libraries.

key_setsecret Subroutine Exported from the libc Library

Purpose

Sets the key for the effective user number (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 known as Data Encryption Standard (DES). This subroutine is used to set the key for the effective 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

Item	Description
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.

key_setsecret Subroutine Exported from the libnsl Library

Purpose

Sets the key for the effective user number (UID) of the calling process.

Library

Network Services Library (**libnsl.a**)

Syntax

```
#include <rpc/rpc.h>

int key_setsecret ( key)
const char *key;
```

Description

The **key_setsecret** subroutine, which belongs to the secure RPC category, is an interface routine to the **keyserver** daemon. User programs rarely need to call this subroutine.

Note: You must run the **keyserv** daemon to enable the subroutine.

Parameters

Item	Description
key	Specifies the key to be set for an effective user ID of the calling process.

Return Values

Item	Description
0	successful
-1	unsuccessful

Examples

```
#include <rpc/rpc.h>
int main()
{
  char key[255] = "deskey"; /* contains the key to be set */
```

```
  if (key_setsecret(key) != 0) {
    fprintf(stderr, "\nkey_setsecret() failed\n");
    exit(1);
  }
  return 0;
}
```

n

The following RPC subroutines begin with the letter n.

netname2host Subroutine

Important: The subroutine is exported from both the **libc** and the **libnsl** libraries.

netname2host Subroutine Exported from the libc Library

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

Item	Description
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.

netname2host Subroutine Exported from the libnsl Library

Purpose

Converts an operating-system-independent network name to a domain-specific host name.

Library

Network Services Library (**libnsl.a**)

Syntax

```
#include <rpc/rpc.h>

int netname2host( name, host, hostlen)
const char *name;
const char *host;
const int hostlen;
```

Description

The **netname2host** subroutine, which belongs to the secure remote procedure call (RPC) category, is used in applications that use the **AUTH_DES** authentication flavor. This subroutine is usually used on server side to convert the network name of a host to the domain-specific host name.

This subroutine is the inverse of the **host2netname** subroutine.

Parameters

Item	Description
name	Specifies the network name of the host.
host	Represents the domain-specific host name after successful completion.
hostlen	Specifies the maximum length of the host name.

Return Values

Item	Description
1	successful
0	unsuccessful

Examples

```
#include <rpc/rpc.h>
static void dispatch(struct svc_req *, SVCXPRT *);

main()
{
  rpcprog_t RPROGNUM = 0x3fffffffL;
  rpcvers_t RPROGVER = 0x1L;

  /* Create service handle for RPROGNUM, RPROGVER and tcp transport */
  if(!svc_create( dispatch, RPROGNUM, RPROGVER, "tcp")) {
    fprintf(stderr,"\nsvc_create() failed\n");
    exit(EXIT_FAILURE);
  }

  svc_run();
}

/* The server dispatch function */
static void dispatch(struct svc_req *rqstp, SVCXPRT *transp)
{
  char       hostp[300];
  struct authdes_cred *des_cred;

  switch (rqstp->rq_cred.oa_flavor) {

  case AUTH_DES :
  /* AUTH_DES Authentication flavor */
```

```
  des_cred = (struct authdes_cred *) rqstp->rq_clntcred;
  if (!netname2host(des_cred->adc_fullname.name, hostp, 300)) {
       svcerr_systemerr(transp);
    return;
  }
  fprintf(stdout,"The domain-specific host name is %s",hostp);
  break;

  default :
  /* Other Authentication flavor */
  break;
  }

  /* The Dispatch Routine code continues .. */
}
```

netname2user Subroutine

Important: The subroutine is exported from both the **libc** and the **libnsl** libraries.

netname2user Subroutine Exported from the libc Library

Purpose

Converts from an operating-system-independent network name to a domain-specific user number (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 UID. This subroutine is the inverse of the **user2netname** subroutine.

Parameters

Item	Description
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.

netname2user Subroutine Exported from the libnsl Library

Purpose

Converts from an operating-system-independent network name to a domain-specific user number (UID).

Library

Network Services Library (**libnsl.a**)

Syntax

```
#include <rpc/rpc.h>

int netname2user(name, uidp, gidp, gidlenp, gidlist)
const char *name;
uid_t *uidp;
gid_t *gidp;
int gidlenp;
gid_t gidlist[NGRPS];
```

Description

The **netname2user** subroutine, which belongs to the secure remote procedure call (RPC) category, is used in applications that use **AUTH_DES** authentication flavor. This subroutine is usually used on the server side to convert the network name of a user tothe domain-specific user-ID.

Note: This subroutine is the inverse of the **user2netname** subroutine.

Parameters

Item	Description
name	Specifies the network name of the host.
uidp	Specifies the effective user ID (UID) of the caller.
gidp	Specifies the effective group ID (GID) of the caller.
gidlenp	Specifies the length of array of groups to which the user belongs.
gidlist	Points to the group array.

Return Values

Item	Description
1	successful
0	unsuccessful

Examples

```
#include <rpc/rpc.h>
static void dispatch(struct svc_req *, SVCXPRT *);

main()
{
  rpcprog_t RPROGNUM = 0x3fffffffL;
  rpcvers_t RPROGVER = 0x1L;

  /* Create service handle for RPROGNUM, RPROGVER and tcp transport */
  if(!svc_create( dispatch, RPROGNUM, RPROGVER, "tcp")) {
    fprintf(stderr,"\nsvc_create() failed\n");
    exit(EXIT_FAILURE);
  }
  svc_run();
}
```

```
/* The server dispatch function */
static void dispatch(struct svc_req *rqstp, SVCXPRT *transp)
{
  struct authdes_cred *des_cred;
  uid_t uid;
  gid_t gid;
  int gidlen;
  gid_t gidlist[10];

  switch (rqstp->rq_cred.oa_flavor) {

    case AUTH_DES :
    /* AUTH_DES Authentication flavor */
    des_cred = (struct authdes_cred *) rqstp->rq_clntcred;
    if (!netname2user(des_cred->adc_fullname.name, &uid, &gid, &gidlist, gidlist))
    {
      svcerr_systemerr(transp);
      return;
    }
    break;
    default :
    /* Other Authentication flavor */
    break;
  }
  /* The Dispatch Routine code continues .. */
}
```

p

The following RPC subroutines begin with the letter p.

pmap_getmaps Subroutine

Important: The subroutine is exported from both the **libc** and the **libnsl** libraries.

pmap_getmaps Subroutine Exported from the libc Library

Purpose

Returns a list of the current Remote Procedure Call (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 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.

Parameters

Item	Description
addr	Specifies the address where the machine's IP address is placed.

Return Values

If there is no list of current RPC programs, this procedure returns a value of null.

pmap_getmaps Subroutine Exported from the libnsl Library

Purpose

Returns a list of the current Remote Procedure Call (RPC) program-to-port mappings on the host.

Library

C Library (**libnsl.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 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.

Parameters

Item	Description
addr	Specifies the address where the machine's IP address is placed.

Return Values

Item	Description
a pointer to the first element of the list	successful
a null value	unsuccessful

Examples

In the following example, the **pmap_getmaps** subroutine obtains a list of the current RPC program-to-port mappings on the host.

```
#include <rpc/rpc.h>
#include <stdio.h>

int main()
{
  rpcprog_t PROGNUM = 0x3fffffffL ;
  rpcvers_t PROGVER = 0x1L ;
  struct hostent *hp;
  struct sockaddr_in addr;
  struct pmaplist *plist = NULL;
```

```
  /* Obtain the host information */
  hp = (struct hostent *) gethostbyname(hostname);
  if (hp == NULL) {
    printf("host information not found\n");
    exit(2);
  }

  addr.sin_family = hp->h_addrtype;
  memcpy(&addr.sin_addr.s_addr, hp->h_addr_list[0], hp->h_length);

  plist = (struct pmaplist *) pmap_getmaps(&addr);
  if(plist==NULL)
  {
    fprintf(stderr,"pmap_getmaps() failed");
    exit(1);
  }

return 0;
}
```

pmap_getport Subroutine

Important: The subroutine is exported from both the **libc** and the **libnsl** libraries.

pmap_getport Subroutine Exported from the libc Library

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

Item	Description
addr	Points to the Internet Protocol (IP) address of the host where the remote program supporting 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 the service recognizes.

Return Values

Upon successful completion, the **pmap_getport** subroutine returns the port number of the requested program; otherwise, 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.

pmap_getport Subroutine Exported from the libnsl Library

Purpose

Requests the port number on which a service waits.

Library

Network Services Library (**libnsl.a**)

Syntax

```
#include <rpc/rpc.h>

u_short pmap_getport (addr, prognum, versnum, protocol)
struct sockaddr_in * addr;
rpcprog_t prognum;
rpcvers_t versnum;
rpcprot_t 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

Item	Description
addr	Points to the Internet Protocol (IP) address of the host where the remote program supporting 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 the service recognizes, which can be IPPROTO_TCP or IPPROTO_UDP.

Return Values

Upon successful completion, the **pmap_getport** subroutine returns the port number of the requested program; otherwise, 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.

Examples

```
#include <rpc/rpc.h>
int main()
{
  struct sockaddr_in addr;
  u_short   port    = 0;
  rpcprog_t PROGNUM = 0x3ffffff0L;
  rpcvers_t PROGVER = 0x1L;
  struct hostent *hp;
  char hostname[255]; /* Remote host name */

  /* Get the information of host */
  hp = (struct hostent *) gethostbyname(hostname);
  if (hp == NULL) {
    printf("host information for %s not found\n", hostname);
    exit(1);
  }
  /* Retrieve the address of host */
  addr.sin_family = hp->h_addrtype;
  memcpy(&addr.sin_addr.s_addr, hp->h_addr_list[0], hp->h_length);

  port = pmap_getport(&addr, PROGNUM, PROGVER, IPPROTO_TCP);
  if(port==0)
  {
    printf("pmap_getport() failed");
    exit(1);
  }

  return 0;
}
```

pmap_getport6 Subroutine

Important: The subroutine is exported from the **libc** library.

Purpose

Requests the port number on which a service waits for IPv6.

Library

C Library (**libc.a**)

Syntax

```
#include <rpc/rpc.h>

u_short pmap_getport6 (addr, prognum, versnum, protocol)
struct sockaddr_in6* addr;
rpcprog_t  prognum;
rpcprog_t  versnum;
rpcprog_t  protocol;
```

Description

The **pmap_getport6** subroutine acts as a user interface to the **portmap** daemon in order to return the port number on which a service waits for IPv6.

Parameters

Item	Description
addr	Points to the Internet Protocol version 6 (IPv6) address of the host where the remote program supporting 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

Upon successful completion, the **pmap_getport6** subroutine returns the port number of the requested program; otherwise, 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.

pmap_rmtcall Subroutine

Important: The subroutine is exported from both the **libc** and the **libnsl** libraries.

pmap_rmtcall Subroutine Exported from the libc Library

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)\
enum clnt_stat pmap_rmtcall (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 (RPC). Clients consult the **portmap** daemon when sending out RPC calls for given program numbers. The **portmap** daemon tells the client the ports to which to send the calls.

Parameters

Item	Description
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.

pmap_rmtcall Subroutine Exported from the libnsl Library

Purpose

Instructs the **portmap** daemon to make a remote procedure call.

Library

Network Services Library (**libnsl.a**)

Syntax

```
#include <rpc/rpc.h>

enum clnt_stat pmap_rmtcall(addr, prognum, versnum, procnum, in, inproc, out, outproc, tout, portp)
struct sockaddr_in *addr;
rpcprog_t prognum;
rpcvers_t versnum;
rpcproc_t procnum;
caddr_t in;
xdrproc_t inproc;
caddr_t out;
xdrproc_t outproc;
struct timeval tout;
rpcport_t *portp;
```

Description

The **pmap_rmtcall** subroutine is a user interface to the **portmap** daemon. The subroutine instructs the host **portmap** daemon to make a remote procedure call (RPC) to a procedure on that host on behalf of the caller. Clients consult the **portmap** daemon when sending out RPC calls for the specified program and version numbers. The **portmap** daemon tells the client the port number to which to send the calls.

Use the **rpcb_rmtcall** subroutine instead of the **pmap_rmtcall** subroutine. The **pmap_rmtcall** subroutine is compatible only with earlier versions of AIX.

Parameters

Item	Description
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	Specifies the program port number. You can set the parameter value to 0.

Error Codes

The subroutine fails when the following error code is true.

Item	Description
RPC_TIMEDOUT	• The timeout value is too small. • The specified program number is not registered at the server side. • The specified version number is not registered at the server side. • The specified procedure number is not registered at the server side. • The server supports only the TCP transport.

Examples

```
#include <rpc/rpc.h>

int main()
{
  rpcprog_t       PROGNUM=0x3fffffffL;
  rpcvers_t       PROGVER=0x1L;
  rpcproc_t       PROCNUM=0x1L;
  struct sockaddr_in addr;
  int in, out;
  struct timeval  timeout = {25, 0};
  rpcport_t       portp=0;
  enum clnt_stat  cs;

  /*
   * Get the IP address of remote host, on which the procedure to be, called is located.
   * Store the value in addr.
   */

  /* Make a call to pmap_rmtcall() subroutine */
  cs = pmap_rmtcall( &addr, PROGNUM, PROGVER, PROCNUM, &in,
   xdr_int, &out, xdr_int, timeout, portp);

  if(cs!=RPC_SUCCESS)
  {
    fprintf(stderr,"pmap_rmtcall failed");
    exit(1);
  }

  return 0;
}
```

pmap_set Subroutine

Important: The subroutine is exported from both the **libc** and the **libnsl** libraries.

pmap_set Subroutine Exported from the libc Library

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

Item	Description
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.

pmap_set Subroutine Exported from the libnsl Library

Purpose

Creates a mapping of the triplet (the program, version, and protocol) to a port.

Library

Network Services Library (**libnsl.a**)

Syntax

```
#include <rpc/rpc.h>

bool_t pmap_set (prognum, versnum, protocol, port)
rpcprog_t prognum;
rpcvers_t versnum;
rpcprot_t 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. The **pmap_set** subroutine is called by the **svc_register** subroutine.

Note: The subroutine only fails if the port is bound.

Parameters

Item	Description
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 **portmap** daemon of the machine.

Return Values

Item	Description
1	successful
0	unsuccessful

Examples

```
#include <rpc/rpc.h>
u_short get_free_port(void)
{
  /* Code to obtain a free port */
}

int main()
{
  u_short   port    = 0;
  rpcprog_t PROGNUM = 0x3ffffff0L;
  rpcvers_t PROGVER = 0x1L;

  /* Obtain a free port */
  port = get_free_port();

  /* Set the mapping between triplet [PROGNUM,PROGVER,PROTOCOL] and port */
  if (pmap_set(PROGNUM, PROGVER, IPPROTO_TCP, port) == 0)
  {
    printf("pmap_set() failed");
    exit(1);
  }

  return 0;
}
```

pmap_unset Subroutine

Important: The subroutine is exported from both the **libc** and the **libnsl** libraries.

pmap_unset Subroutine Exported from the libc Library

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

Item	Description
prognum	Specifies the program number of the remote program.
versnum	Specifies the version number of the remote program.

pmap_unset Subroutine Exported from the libnsl Library

Purpose

Destroys the mappings between the triplet (the program, version, and protocol) and the port.

Library

Network Services Library (**libnsl.a**)

Syntax

```
#include <rpc/rpc.h>

bool_t pmap_unset ( prognum, versnum)
rpcprog_t prognum;
rpcvers_t versnum;
```

Description

The **pmap_unset** subroutine destroys mappings between the triplet (the program number, version number, and protocol) and the port of a remote procedure call and the ports on the host **portmap** daemon. The mapping can be established by the **pmap_set** subroutine.

Parameters

Item	Description
prognum	Specifies the program number of the remote program.
versnum	Specifies the version number of the remote program.

Examples

```
#include <rpc/rpc.h>
u_short get_free_port(void)
{
  /* Code to obtain a free port */
}

int main()
{
  u_short port = 0;
  rpcprog_t PROGNUM = 0x3ffffff0L;
  rpcvers_t PROGVER = 0x1L;
```

```
  /* Obtain a free port */
  port = get_free_port();

  /* Set the mapping between triplet [PROGNUM,PROGVER,PROTOCOL] and port */
  if (pmap_set(PROGNUM, PROGVER, IPPROTO_TCP, port) == 0)
  {
    printf("pmap_set() failed");
    exit(1);
  }

  if(pmap_unset(PROGNUM, PROGVER)==0)
  {
    printf("pmap_unset() failed");
    exit(1);
  }

  return 0;
}
```

r

The following RPC subroutines begin with the letter r.

registerrpc Subroutine

Purpose

Registers a procedure with the Remote Procedure Call (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 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

Item	Description
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.

Related reference:

"callrpc Subroutine" on page 218

"svcudp_create Subroutine" on page 399

Related information:

List of RPC Programming References

Remote Procedure Call (RPC) Overview for Programming

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

Item	Description
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 **errno** global variable is set to reflect the cause of the error.

Related information:

List of RPC Programming References

TCP/IP protocols

Remote Procedure Call (RPC) Overview for Programming

rpc_broadcast Subroutine
Purpose

Invokes the remote procedure associated with the specified program and version by broadcasting the call message through all connectionless transports of the specified class.

Library

Network Services Library (**libnsl.a**)

Syntax

```
#include <rpc/rpc.h>

enum clnt_stat rpc_broadcast(progrnum, versnum, procnum, in_proc, input, out_proc, output, result, nettype)
const rpcprog_t progrnum;
const rpcvers_t versnum;
const rpcproc_t procnum;
const xdrproc_t in_proc;
caddr_t input;
const xdrproc_t out_proc;
caddr_t output;
const resultproc_t result;
const char *nettype ;
```

Description

The **rpc_broadcast** subroutine calls the remote procedure associated with the specified program and version. When calling the procedure, the subroutine broadcasts the call message through all connectionless transports of the specified class. You can specify an eXternal Data Representation (XDR) procedure that encodes the procedure parameters along with the address of the parameters. Similarly, you can specify an XDR procedure that decodes the procedure results along with the address where those results are to be placed. Every time the **rpc_broadcast** subroutine receives a response, the subroutine calls the following subroutine:

```
bool_t result(caddr_t output, const struct netbuf *addr, const struct netconfig *nconf);
```

The *output* parameter of the subroutine is the same as that of the **rpc_broadcast** subroutine. The *addr* parameter holds the address of the machine that sent the results. The *nconf* parameter specifies the transport that is used by the machine to respond. If the **result** subroutine returns a value of 0, the **rpc_broadcast** subroutine waits for more replies. Otherwise, the subroutine returns with an appropriate status.

Parameters

Item	Description
prognum	Specifies the program number of the remote program.
versnum	Specifies the version number of the remote program.
procnum	The remote procedure number.
in_proc	An XDR procedure for encoding the procedure parameters.
input	The address of the procedure arguments.
out_proc	An XDR procedure for decoding the procedure results.
output	The address where the results will be placed.
result	The subroutine that is invoked when the **rpc_broadcast** receives a response.
nettype	Defines a class of transports which can be used for a particular application.

Return Values

Item	Description
0	successful
an appropriate status	unsuccessful

You can obtain the status using the **clnt_perrno** subroutine.

Error Codes

The **rpc_broadcast** subroutine returns failure when one or more of the following codes are true.

Item	Description
RPC_UNKNOWNPROTO	• The value of the *nettype* parameter is not valid. • The value of the *nettype* parameter is set to **netpath**, and the NETPATH environment variable is set to a transport service that is not valid.
RPC_TIMEDOUT	• The timeout value has expired. • The specified version is not registered at the server. • The remote procedure is not available.
RPC_PROGVERSMISMATCH	The specified version is not registered at the server.
RPC_FAILED	An unspecified error occurred. The procedure specified by the *in_proc* or *out_proc* parameter might not be valid.
RPC_CANTDECODEARGS	The arguments or results are not valid.
RPC_SYSTEMERROR	All of the process memory is exhausted (heap).

Examples

```
#include <rpc/rpc.h>

bool_t result(caddr_t out, const struct netbuf *addr, const struct netconfig *nconf)
{
    /* result() subroutine code */
}
int main()
{
    rpcprog_t program_number = 0x3fffffffL;
    rpcvers_t version_number = 0x1L;
    rpcproc_t procedure_number = 0x1L;
    enum clnt_stat cs ;
    char *nettype = "visible";
    cs = rpc_broadcast(program_number, version_number, procedure_number,
                (xdrproc_t)xdr_void, NULL, (xdrproc_t)xdr_void,
                NULL, eachresult, nettype);
    if (cs != RPC_SUCCESS)
    {
       fprintf(stderr,"\n RPC Call failed\n");
```

```
        exit(1);
    }
    return 0;
}
```

Related reference:

"rpc_broadcast_exp Subroutine"

"rpc_broadcast_exp Subroutine"

Related information:

Transport Independent Remote Procedure Call, IPv6 concerns for Transport Independent Remote Procedure Call

eXternal Data Representation Overview for Programming

rpc_broadcast_exp Subroutine
Purpose

Invokes the remote procedure associated with specified program and version by broadcasting the call message through all connectionless transports of the specified class with initial wait time and the maximum wait-time constraints.

Library

Network Services Library (**libnsl.a**)

Syntax

```
#include <rpc/rpc.h>

enum clnt_stat rpc_broadcast_exp(progNum, versnum, procnum, in_proc, input, out_proc, output,
result, itime, wtime, nettype)
const rpcprog_t prognum;
const rpcvers_t versnum;
const rpcproc_t procnum;
const xdrproc_t in_proc;
caddr_t input;
const xdrproc_t out_proc;
caddr_t output;
const resultproc_t result;
const int itime;
const int wtime;
const char *nettype;
```

Description

The **rpc_broadcast_exp** subroutine calls the remote procedure associated with the specified program and version. When calling the procedure, the subroutine broadcasts the call message through all connectionless transports of the specified class. You can specify an eXternal Data Representation (XDR) procedure that encodes the procedure parameters along with the address of the parameters. Similarly, you can specify an XDR procedure that decodes the procedure results along with the address where those results are to be placed. Every time the **rpc_broadcast_exp** subroutine receives a response, the subroutine calls the following subroutine:

```
bool_t result(caddr_t output, const struct netbuf *addr, const struct netconfig *nconf);
```

The *output* parameter of the subroutine is the same as that of the **rpc_broadcast_exp** subroutine. The *addr* parameter holds the address of the machine that sent the results. The *nconf* parameter specifies the transport used by the machine to respond. You can specify the initial time before the request is resent in milliseconds. Similarly, after the request is resent for the first time, the retransmission interval increases

exponentially until it exceeds the maximum value that you can also specify in milliseconds. If the **result** subroutine returns a value of 0, the **rpc_broadcast_exp** subroutine waits for more replies. Otherwise, the subroutine returns with an appropriate status.

Parameters

Item	Description
prognum	Specifies the program number of the remote program.
versnum	Specifies the version number of the remote program.
procnum	Specifies the remote procedure number.
in_proc	Specifies the XDR procedure for encoding the procedure parameters.
input	Specifies the address of the procedure arguments.
out_proc	Specifies the XDR procedure for decoding the procedure results.
output	Specifies the address where the results will be placed.
result	Specifies the subroutine that is invoked when the **rpc_broadcast_exp** receives a response.
itime	Specifies the initial timeout before the request is resent.
wtime	Specifies the maximum timeout.
nettype	Defines a class of transports which can be used for a particular application.

Return Values

The **rpc_broadcast_exp** subroutine returns failure when one or more of the following codes are true.

Item	Description
RPC_UNKNOWNPROTO	• The value of the *nettype* parameter is not valid. • The value of the *nettype* parameter is set to **netpath**, and the NETPATH environment variable is set to a transport service that is not valid.
RPC_TIMEDOUT	• The timeout value has expired. • The specified version is not registered at the server. • The remote procedure is not available.
RPC_PROGVERSMISMATCH	The specified version is not registered at the server.
RPC_FAILED	An unspecified error occurred. The procedure specified by the *in_proc* or *out_proc* parameter might not be valid.
RPC_CANTDECODEARGS	The arguments or results are not valid.
RPC_SYSTEMERROR	All of the process memory is exhausted (heap).

Examples

```
#include <rpc/rpc.h>

bool_t result(caddr_t output, const struct netbuf *addr, const struct netconfig *nconf)
{
    /* result() subroutine code */
}

int main()
{
    rpcprog_t program_number = 0x3fffffffL;
    rpcvers_t version_number = 0x1L;
    rpcproc_t procedure_number = 0x1L;
    enum clnt_stat cs ;
    char *nettype = "visible";
    const int itime = 5 ;
    const int wtime = 25 ;
    cs = rpc_broadcast_exp( program_number, version_number, procedure_number,
                      (xdrproc_t)xdr_void, NULL,(xdrproc_t)xdr_void, NULL,
                       result, itime, wtime, nettype);
    if (cs != RPC_SUCCESS)
    {
       fprintf(stderr,"\n RPC Call failed\n");
```

```
        exit(1);
    }
    return 0;
}
```

Related reference:

"rpc_broadcast Subroutine" on page 304

"rpc_broadcast Subroutine" on page 304

Related information:

Transport Independent Remote Procedure Call, IPv6 concerns for Transport Independent Remote Procedure Call

eXternal Data Representation Overview for Programming

rpc_call Subroutine
Purpose

Calls the remote procedure associated with the specified program and version on a remote host.

Library

Network Services Library (**libnsl.a**)

Syntax

```
#include <rpc/rpc.h>

enum clnt_stat rpc_call(host, prognum, versnum, procnum, in_proc, input, out_proc, output, nettype)
const char *host;
const rpcprog_t prognum;
const rpcvers_t versnum;
const rpcproc_t procnum;
const xdrproc_t in_proc;
const char *input;
const xdrproc_t out_proc;
char *output;
const char *nettype;
```

Description

The **rpc_call** subroutine calls the remote procedure associated with the specified program and version. The remote procedure that is specified by the *procnum* procedure resides on a remote host. You can specify an eXternal Data Representation (XDR) procedure that encodes the procedure parameters along with the addresses of the parameters. Similarly, you can specify an XDR procedure that decodes the procedure results and address where those results are to be placed. You can specify the transport class using the *nettype* parameter. The **rpc_call** subroutine uses the first available transport. You cannot control timeout, and you cannot control authentication because the client handle is not created (as in client-creation subroutines).

Parameters

Item	Description
host	Specifies the host name where the server resides.
prognum	Specifies the program number of the remote program.
versnum	Specifies the version number of the remote program.
procnum	The remote procedure number.
in_proc	The XDR procedure for encoding the procedure parameters.
input	The address of the procedure arguments.
out_proc	The XDR procedure for decoding the procedure results.
output	The address where the results are placed.
nettype	Defines a class of transports that can be used for a particular application.

Return Values

Item	Description
RPC_SUCCESS	successful
an appropriate status	unsuccessful

You can use the **clnt_perrno** subroutine to get the status.

Error Codes

The **rpc_call** subroutine returns failure when one or more of the following codes are true.

Item	Description
RPC_UNKNOWNPROTO	• The value specified by the *nettype* parameter is not valid. • The value specified by the *nettype* parameter is set to **netpath**, and the NETPATH environment variable is set to a transport service that is not valid.
RPC_UNKNOWNHOST	The host name is not valid.
RPC_PROCUNAVAIL	The remote procedure is not available
RPC_TIMEDOUT	The timeout value has expired.
RPC_PROGVERSMISMATCH	The specified version is not registered at the server.
RPC_FAILED	An unspecified error occurred. The procedure specified by the *in_proc* or *out_proc* parameter might not be valid.
RPC_CANTDECODEARGS	The arguments or results are not valid.
RPC_SYSTEMERROR	All of the process memory is exhausted (heap).

Examples

```
#include <stdlib.h>
#include <rpc/rpc.h>
int main()
{
    char hostname[255] ;
    /* The Remote host on which server is implemented */
    rpcprog_t program_number ;
    rpcvers_t version_number ;
    rpcproc_t procedure_number ;
    enum clnt_stat cs ;
    char *nettype = "visible";

    cs = rpc_call(hostname, program_number, version_number, procedure_number,
         (xdrproc_t)xdr_void, NULL,(xdrproc_t)xdr_void, NULL, nettype);
    if (cs != RPC_SUCCESS)
    {
        fprintf(stderr,"\n RPC Call failed\n");
        exit(1);
    }

    return 0;
}
```

Related reference:

"clnt_call Macro" on page 222

Related information:

Transport Independent Remote Procedure Call, IPv6 concerns for Transport Independent Remote Procedure Call

eXternal Data Representation Overview for Programming

rpc_control Subroutine
Purpose

Changes or retrieves information of global remote procedure call (RPC) attributes for client and server applications.

Library

Network Services Library (**libnsl.a**)

Syntax

```
#include <rpc/rpc.h>

bool_t rpc_control(op, info);
int op;
void * info;
```

Description

The subroutine sets and retrieves values of global RPC attributes that apply to clients and servers. The *op* parameter indicates the operation type and the *info* parameter is a pointer to the operation-specific information. The data type specified by the *info* parameter changes according to the operation type. For example, you can set the *op* parameter with the following values:

RPC_SVC_MTMODE_SET
Sets the multithread mode.

RPC_SVC_MTMODE_GET
Get the multithread mode.

For the *op* parameter in this example, the value of the *info* parameter is of the **int *** type.

Values for the *op* Parameter	Argument Type	Function
RPC_SVC_MTMODE_SET	**int ***	Sets the multithread mode.
RPC_SVC_MTMODE_GET	**int ***	Gets the multithread mode.
RPC_SVC_THRMAX_SET	**int ***	Sets the maximum number of threads.
RPC_SVC_THRMAX_GET	**int ***	Gets the maximum number of threads.
RPC_SVC_THRTOTAL_GET	**int ***	Gets the number of active threads.
RPC_SVC_THRCREATES_GET	**int ***	Gets the number of threads created.
RPC_SVC_THRERRORS_GET	**int ***	Gets the number of threads that create errors.
RPC_SVC_USE_POLLFD	**int ***	Sets the number of file descriptors to unlimited.

Three multithread (MT) modes are listed in the following table.

Item	Description
RPC_SVC_MT_NONE	the single-threaded mode (default)
RPC_SVC_MT_AUTO	the automatic MT mode
RPC_SVC_MT_USER	the user MT mode

The default (single-threaded) mode stays unless the application sets the other two modes. When a mode is set, it cannot be changed. A server can create a maximum of 16 threads anytime. You can restrict the number of thread resources consumed by a server. If a server needs more than 16 threads, set the maximum number of threads to a desired number. Similarly, RPC servers are limited to a maximum of 1024 file descriptors or connections. Applications that use preferred interfaces of the **svc_pollfd** global variable and the **svc_getreq_poll** subroutines can use unlimited number of file descriptors. To achieve the

goal, you can point the *info* parameter to nonzero and set the *op* parameter value to **RPC_SVC_USE_POLLFD**.

Parameters

Item	Description
op	Represents the operation type.
info	Points to the information for the request type. The *info* parameter is expected to be a pointer to an appropriate structure. The nature of the structure depends on the operation type.

Return Values

Item	Description
TRUE	successful
FALSE	unsuccessful

Examples

In the following example, the **rpc_control** subroutine is used to set server program in the automatic MT mode.

```
#include <stdlib.h>
#include <rpc/rpc.h>

int main()
{
  SVCXPRT *transpnum;
  rpcprog_t prognum = 0x3fffffffL;
  rpcvers_t progver = 0x1L;

  /* Register the service for prognum & progver on tcp transport */
  transpnum = svc_create(dispatch_AUTOMT, prognum, progver, "tcp");
  if (transpnum == 0)
  {
    fprintf(stderr, "Cannot create a service.\n");
    svc_unreg(prognum,progver);
    exit(1);
  }

  /* Configure the server in AUTO_MT mode */
  mode = RPC_SVC_MT_AUTO;
  if(rpc_control(RPC_SVC_MTMODE_SET,&mode) == FALSE)
  {
    fprintf(stderr,"\nError in rpc_control!\n");
    exit(1);
  }

  svc_run();

  return 0;
}
```

Related information:

Transport Independent Remote Procedure Call, IPv6 concerns for Transport Independent Remote Procedure Call

eXternal Data Representation Overview for Programming

rpc_createerr Global Variable
Purpose

Holds the status of a client-handle-creation subroutine for a remote procedure call (RPC).

Library

Network Services Library (**libnsl.a**)

Syntax

```
#include <rpc/rpc.h>
struct rpc_createerr rpc_createerr
```

Description

Whenever a client-creation subroutine fails, the subroutine sets the value of the **rpc_createerr** global variable to an appropriate error code. The **clnt_pcreateerror** and **clnt_spcreateerror** subroutines use this global variable to display the failure reason.

Note: For multithreaded applications, each thread has its own **rpc_createerr** variable.

Examples

In the following example, the **rpc_createerr** global variable is used to display the error code.

```
#include <stdlib.h>
#include <rpc/rpc.h>

int main()
{
     CLIENT *cl;
     rpcprog_t PROGNUM = 0x3fffffffL;
     rpcvers_t PROGVER = 0x1L;
     char *nettype = "visible";
     char hostname[255] ;      /* The name of remote host */

     /*
      * make the clnt_create call with this nettype and
      * observe the result
      */
     if ((cl=clnt_create( hostname, PROGNUM, PROGVER, nettype)) == NULL)
     {
          fprintf(stdout, "The error status : %d\n" , rpc_createerr.cf_stat);
          exit(EXIT_FAILURE);
     }

     /* destroy the client handle */
     clnt_destroy(cl);


     return 0;
}
```

Related information:

Transport Independent Remote Procedure Call

eXternal Data Representation Overview for Programming

rpc_gss_get_error Subroutine
Purpose

Gets an error number on failure.

Library

Network Services Library (**libnsl.a**)

Syntax

```
#include <rpc/rpcsec_gss.h>
bool_t rpc_gss_get_error(rpc_gss_error_t *err);
```

Description

You can use the **rpc_gss_get_error** subroutine to retrieve the error code when RPCSEC_GSS subroutines fail.

Parameters

Item	Description
err	Points to an **rpc_gss_error_t** structure. This is an output parameter.

Return Values

Item	Description
TRUE	successful
FALSE	unsuccessful

If an RPCSEC_GSS subroutine fails, only the **rpc_gss_get_error** subroutine sets the error to a meaningful value.

Error Codes

Item	Value	Description
RPC_GSS_ER_SUCCESS	0	No error occurred.
RPC_GSS_ER_SYSTEMERROR	1	A system error occurred.

Examples

```
#include <stdlib.h>
#include <rpc/rpc.h>
#include <rpc/rpcsec_gss.h>
#define PROGNUM 0x3fffffffL
#define VERSNUM 0x1L

static void sample_dispatch(struct svc_req *, SVCXPRT *);
main()
{
  char *principal, *mechanism;
  u_int reqtime
  rpc_gss_error_t gss_error;

  /* Create RPC service handle and register with RPCBIND service */

  /* Initialize the required parameters */

  /* Set the principal name */
  if(rpc_gss_set_svc_name(principal, mechanism, req_time, PROGNUM,
  VERSNUM) == FALSE)
  {
    fprintf(stderr,"\nError in rpc_gss_set_svc_name:\n");
    /* Retrieve error */
    rpc_gss_get_error(&gss_error);
    fprintf(stderr,"rpc_gss_error: %d \nSystem_error: %d \n",
    gss_error.rpc_gss_error,gss_error.system_error);
    exit(EXIT_FAILURE);
  }
  svc_run();
```

```
  return 1;
}

/* following is the sample dispatch routine*/
static void sample_dispatch(struct svc_req *request, SVCXPRT *xprt)
{
  /* dispatch routine code */
}
```

Related information:

Transport Independent Remote Procedure Call

eXternal Data Representation Overview for Programming

rpc_gss_get_mech_info Subroutine

Purpose

Gets a list of quality of protections for the specified mechanism and security type.

Library

Network Services Library (**libnsl.a**)

Syntax

```
#include <rpc/rpcsec_gss.h>

char ** rpc_gss_get_mech_info(mechanism, service)
char *mechanism;
rpc_gss_service_t *service;
```

Description

The subroutine provides a list of quality of protections for the specified mechanism and security type.

Parameters

Item	Description
mechanism	Represents the supported security mechanism that is used for context creation (for example, *kerberosv5*).
service	Represents the type of service for the session that basically offers a level of protection (for example, integrity and privacy).

Return Values

Item	Description
a list of character strings terminated by a null value	successful
a null value	unsuccessful

The value of null specifies that you can use the default quality of protection.

Examples

```
#include <stdlib.h>
#include <rpc/rpc.h>
#include <rpc/rpcsec_gss.h>

int main(int argc,char *argv[])
{
    char **s;
    char *mechanism;
    int i;
```

```
    rpc_gss_service_t service;

    mechanism = "kerberosv5";
    service = 2;              /* 1: none, 2: integrity. 3: privacy */

    if((s = rpc_gss_get_mech_info(mechanism, &service)) == NULL)
    {
         fprintf(stderr,"\nrpc_gss_get_mech_info() Returned NULL, default QOP value can be used!\n");
         exit(1);
    }
    return 0;
}
```

Related information:

Transport Independent Remote Procedure Call

eXternal Data Representation Overview for Programming

rpc_gss_get_mechanisms Subroutine

Purpose

Gets a list of supported security mechanisms.

Library

Network Services Library (**libnsl.a**)

Syntax

```
#include <rpc/rpcsec_gss.h>
char ** rpc_gss_get_mechanisms();
```

Description

The **rpc_gss_get_mechanisms** subroutine returns a list of supported security mechanisms.

Return Values

Item	Description
a NULL-terminated list of character strings	successful
NULL	unsuccessful

Examples

```
#include <stdlib.h>
#include <rpc/rpc.h>
#include <rpc/rpcsec_gss.h>

int main(void)
{
    char **s;
    /* Get mechanisms */
    if((s = rpc_gss_get_mechanisms()) == NULL)
    {
        fprintf(stderr,"\nrpc_gss_get_mechanisms() failed!\n");
        exit(1);
    }
    return 0;
}
```

Related information:

Transport Independent Remote Procedure Call

rpc_gss_get_principal_name Subroutine
Purpose

Gets the principal name of a known entity at the server end.

Library

Network Services Library (**libnsl.a**)

Syntax

```
#include <rpc/rpcsec_gss.h>

bool_t rpc_gss_get_principal_name(s_principal,mech,name_u,node,secdomain)
rpc_gss_principal_t *s_principal;
char *mech;
char *name_u;
char *node;
char *secdomain;
```

Description

Sometimes, a server wants to compare principal name that it has received with that of a known entity. The **rpc_gss_get_principal_name** subroutine provides the principal name of a known entity. This subroutine has various parameters that uniquely identify the known entity on the network and creates principal name of the **rpc_gss_principal_t** type.

Parameters

Item	Description
s_principal	Represents the principal name of a client. This is an output parameter.
mech	Represents the supported security mechanism that is used (for example, `kerberosv5`).
name_u	Specifies the UNIX login name.
node	Represents the machine name.
secdomain	Represents the security domain.

Parameter values are dependent on security mechanism. For those parameters that are not applicable for a particular security mechanism, you can specify NULL.

Return Values

Item	Description
TRUE	successful
FALSE	unsuccessful

You can use the **rpc_gss_get_error** subroutine to retrieve the error number.

Examples

In the following example, the principal name is constructed for users with the `myuser` UNIX-login name, the `mynode` node, the `mydomain` domain, and the `kerberosv5` security mechanism.

```
#include <stdlib.h>
#include <rpc/rpc.h>
#include <rpc/rpcsec_gss.h>
#define PROGNUM 0x3fffffffL
#define VERSNUM 0x1L
```

```
static void sample_dispatch(struct svc_req *, SVCXPRT *);
main()
{

    /* Create RPC service handle and register with RPCBIND service */

    /* Set the principal name */

    svc_run();
    return 1;
}

/* following is the sample dispatch routine*/
static void sample_dispatch(struct svc_req *request, SVCXPRT *xprt)
{
    char *myuser,*mynode,*mydomain;
    rpc_gss_principal_t princ;

    myuser = "test01";
    mynode = "localhost";
    mydomain = "ibm.com";
    if (!rpc_gss_get_principal_name(&princ,"kerberosv5",myuser,mynode,mydomain))
    {
         fprintf(stderr,"Error in getting principal name\n");
         exit(1);
    }
    /* Compare retrieved principal name in 'princ' with received principal name */
    /* Send reply back to caller */
}
```

Related information:

Transport Independent Remote Procedure Call

eXternal Data Representation Overview for Programming

rpc_gss_get_versions Subroutine

Purpose

Gets the highest and lowest version of RPCSEC_GSS.

Library

Network Services Library (**libnsl.a**)

Syntax

```
#include <rpc/rpcsec_gss.h>
bool_t rpc_gss_get_versions(vers_hi,vers_lo)
u_int *vers_hi;
u_int *vers_lo;
```

Description

You can use this subroutine to determine the highest and the lowest version of RPCSEC_GSS that is supported.

Parameters

Item	Description
vers_hi	Points to the highest version when a subroutine returns successfully.
vers_lo	Points to the lowest version when a subroutine returns successfully.

Return Values

Item	Description
TRUE	successful
FALSE	unsuccessful

Examples

```
#include <stdlib.h>
#include <rpc/rpc.h>
#include <rpc/rpcsec_gss.h>

int main()
{
    int high_vers, low_vers;
    if (rpc_gss_get_versions(&high_vers,&low_vers))
    {
       fprintf(stderr,"\nError in rpc_gss_get_versions:\n");
       exit(EXIT_FAILURE);
    }
    return 0;
}
```

Related information:

Transport Independent Remote Procedure Call

eXternal Data Representation Overview for Programming

rpc_gss_getcred Subroutine

Purpose

Gets credentials of a caller.

Library

Network Services Library (**libnsl.a**)

Syntax

```
#include <rpc/rpcsec_gss.h>

bool_t rpc_gss_getcred(req, r_cred, u_cred, cookie)
struct svc_req *req;
rpc_gss_rawcred_t **r_cred;
rpc_gss_ucred_t **u_cred;
void **cookie;
```

Description

The **rpc_gss_getcred** subroutine is used to get credentials of a caller. You can retrieve network credentials and UNIX credentials.

Parameters

Item	Description
req	Points to a received service-request structure.
r_cred	Points to an **rpc_gss_rawcred_t** structure that is returned with raw credentials. Raw credentials include the remote procedure call (RPC) version, security mechanism, quality of protection, client principal, server principal, service type, and so on. This is an output parameter. You can specify the parameter with NULL.
u_cred	Points to an **rpc_gss_ucred_t** structure that is returned with UNIX credentials. UNIX credentials include user ID, group ID, and so on. This is an output parameter. You can specify the parameter with NULL.
cookie	Represents a 4-byte entity that an application can use in any manner. This is an output parameter.

Return Values

Item	Description
TRUE	successful
FALSE	unsuccessful

You can use the **rpc_gss_get_error** subroutine to retrieve the error number.

Examples

In the following example, credentials of the caller are retrieved in the dispatch routine of the server.

```
#include <stdlib.h>
#include <rpc/rpc.h>
#include <rpc/rpcsec_gss.h>

#define PROGNUM 0x3fffffffL
#define VERSNUM 0x1L

static void sample_dispatch(struct svc_req *, SVCXPRT *);
main()
{

  /* Create RPC service handle and register with RPCBIND service */

  /* Set the principal name */

  svc_run();
  return 1;
}

/* following is the sample dispatch routine*/
static void sample_dispatch(struct svc_req *request, SVCXPRT *xprt)
{
  rpc_gss_rawcred_t *r_cred;
  rpc_gss_ucred_t *u_cred;

  /* Get caller's credentials */
  if(rpc_gss_getcred(request, &r_cred, &u_cred, NULL) == FALSE)
  {
    fprintf(stderr,"\nError in rpc_gss_getcred:\n");
    rpc_gss_get_error(&gss_error);
    fprintf(stderr,"rpc_gss_error: %d \nSystem_error: %d \n",
    gss_error.rpc_gss_error,gss_error.system_error);
    svcerr_systemerr(xprt);
    return;
  }

  /* Send reply back to caller */
}
```

Related information:

rpc_gss_is_installed Subroutine
Purpose

Checks whether a security mechanism is installed.

Library

Network Services Library (**libnsl.a**)

Syntax

```
#include <rpc/rpcsec_gss.h>

bool_t rpc_gss_is_installed(mechanism)
char *mechanism;
```

Description

You can use the subroutine to determine whether the specified security mechanism is installed.

Parameters

Item	Description
mechanism	Specifies a security mechanism (for example, *kerberosv5*).

Return Values

Item	Description
TRUE	successful
FALSE	unsuccessful

Examples

```
#include <stdlib.h>
#include <rpc/rpc.h>
#include <rpc/rpcsec_gss.h>

int main()
{

    char *mechanism;

    mechanism = "kerberosv5";
    /* Check if mechanism is installed */
    if(rpc_gss_is_installed(mechanism) == FALSE)
    {
        fprintf(stderr,"\n%s Mechanism not installed!\n", mechanism);
        exit(1);
    }
    return 0;
}
```

Related information:

Transport Independent Remote Procedure Call

eXternal Data Representation Overview for Programming

rpc_gss_max_data_length Subroutine
Purpose

Gets the maximum length of untransformed data that is allowed by the transport (a client-side version).

Library

Network Services Library (**libnsl.a**)

Syntax

```
#include <rpc/rpcsec_gss.h>

int rpc_gss_max_data_length(a_handle, max_tp_length)
AUTH *a_handle;
int max_tp_length;
```

Description

Some transport types have restrictions on the maximum size of data that can be sent out in one data unit. After the security transformations on actual data, data length increases that depends on the selected security mechanism. Some applications need to know the actual length of untransformed data that is allowed before performing security transformations. You can get this maximum length of untransformed data using the **rpc_gss_max_data_length** subroutine.

Parameters

Item	Description
a_handle	Represents an RPC handle that is returned when security context is created.
max_tp_length	Represents the maximum length of data unit allowed by transport. This is an input parameter.

Return Values

On successful completion, the **rpc_gss_max_data_length** subroutine returns the maximum size of untransformed data that is allowed.

Examples

```
#include <stdlib.h>
#include <tiuser.h>
#include <rpc/rpc.h>
#include <rpc/rpcsec_gss.h>

main()
{
    CLIENT *client;
    int fd, untransformed_data, max_tp_len;
    struct t_info info;

    /* Create client handle */
        /* Create security context */

    /* Get associated file descriptor */
    if(clnt_control(client,CLGET_FD,(caddr_t)&fd) == FALSE)
    {
        fprintf(stderr,"\nError in clnt_control.\n");
        exit(1);
    }
    /* Get info related to transport */
    if(t_getinfo(fd,&info) !=0)
    {
        fprintf(stderr,"\nError in t_getinfo.\n");
```

```
        exit(1);
    }
    /* Get max data length allowed by transport */
    max_tp_len = info.tsdu;

    /* get max untransformed data length */
    untransformed_data = rpc_gss_max_data_length(client->cl_auth,  max_tp_len);
}
```

Related information:

Transport Independent Remote Procedure Call

eXternal Data Representation Overview for Programming

rpc_gss_mech_to_oid Subroutine

Purpose

Gets values of object-identifier structure corresponding to the specified mechanism.

Library

Network Services Library (**libnsl.a**)

Syntax

```
#include <rpc/rpcsec_gss.h>

bool_t rpc_gss_mech_to_oid(mech,oid)
char *mech;
rpc_gss_OIDc *oid;
```

Description

Kernel remote procedure call (RPC) routines use non-string values to represent mechanisms and quality of parameters. The non-string values, which an application sometimes needs, can be in the form of structures or just numbers. This subroutine provides values of an object-identifier structure that are related to the specified mechanism.

Parameters

Item	Description
mech	Represents the supported security mechanism that is used for context creation (for example, *kerberosv5*).
oid	Points to an **rpc_gss_OIDc** structure that is filled up by this subroutine.

Return Values

Item	Description
TRUE	successful
FALSE	unsuccessful

Examples

```
#include <stdlib.h>
#include <rpc/rpc.h>
#include <rpc/rpcsec_gss.h>

int main(int argc,char *argv[])
{
    rpc_gss_OID *oid;
    char *mechanism;
```

```
    mechanism = "kerberosv5";
    /* Get non-string value for mechanism */
    if(rpc_gss_mech_to_oid(mechanism,oid) == FALSE)
    {
        fprintf(stderr,"\nrpc_gss_mech_to_oid() failed!\n");
        exit(1);
    }
    return 0;
}
```

Related information:

Transport Independent Remote Procedure Call

eXternal Data Representation Overview for Programming

rpc_gss_qop_to_num Subroutine

Purpose

Gets the number that is related to specified mechanism and quality of protection.

Library

Network Services Library (**libnsl.a**)

Syntax

```
#include <rpc/rpcsec_gss.h>

bool_t rpc_gss_qop_to_num(qop,mech,num)
char *qop;
char *mech;
u_int *num;
```

Description

Kernel remote procedure call (RPC) routines use non-string values to represent mechanisms and quality of parameters. The non-string values, which an application sometimes needs, can be in the form of structures or just numbers. This subroutine provides the number that is related to the specified mechanism and quality of protection.

Parameters

Item	Description
qop	Represents the quality of protection (qop).
mech	Represents the supported security mechanism that is used for context creation (for example, *kerberosv5*).
num	Represents a non-string value for the specified mechanism-qop combination. The value is filled up by the subroutine.

Return Values

Item	Description
TRUE	successful
FALSE	unsuccessful

Examples

```
#include <stdlib.h>
#include <rpc/rpc.h>
#include <rpc/rpcsec_gss.h>

int main(int argc, char *argv[])
```

```
{
    u_int num;
    char *qop, *mechanism;

    mechanism = "kerberosv5";
    qop = "GSS_C_QOP_DEFAULT";
    /* Get non-string value for qop */
    if(rpc_gss_qop_to_num(qop, mechanism, &num) == FALSE)
    {
         fprintf(stderr,"\nrpc_gss_qop_to_num() failed!\n");
         exit(1);
    }
    return 0;
}
```

Related information:

Transport Independent Remote Procedure Call

eXternal Data Representation Overview for Programming

rpc_gss_set_svc_name Subroutine

Purpose

Sets the principal name that a server or a service represents.

Library

Network Services Library (**libnsl.a**)

Syntax

```
#include <rpc/rpcsec_gss.h>

bool_t rpc_gss_set_svc_name(s_principal, mech, r_time, prog, vers)
char *s_principal;
char *mech;
u_int r_time;
u_int prog;
u_int vers;
```

Description

When a client wants to use any service provided by a server with RPCSEC_GSS APIs, the client basically addresses server principals rather than actual services. A principal is a user or a service that uses authentication services and is identified in authentication database. The **rpc_gss_set_svc_name** subroutine sets the principal name that the server or service represents. You can use this subroutine to set more than one principal name to the same server or service.

Parameters

Item	Description
s_principal	Specifies a server principal of the form *service@host*. The *service* variable represents the service offered by a server and the *host* variable indicates the name of a machine on which the server resides (for example, `nfs@aix1.ibm.com`).
mech	Represents the supported security mechanism that is used for client-server communication (for example, `kerberosv5`).
r_time	Represents the time, in seconds, for which credentials must be valid. (The time is mechanism-dependent.)
prog	Represents the remote procedure call (RPC) program number of a service.
vers	Represents the RPC version number of a service.

Return Values

Item	Description
TRUE	successful
FALSE	unsuccessful

You can use the **rpc_gss_get_error** subroutine to retrieve the error number.

Examples

In the following example, the principal name is set for the RPC service with the program and version number that are provided by the server.

```
#include <stdlib.h>
#include <rpc/rpc.h>
#include <rpc/rpcsec_gss.h>

#define PROG 0x3fffffffL
#define VERS 0x1L

static void sample_dispatch(struct svc_req *, SVCXPRT *);
main()
{
    char *s_principal, *mech;
    u_int r_time
    rpc_gss_error_t gss_error;

    /* Create RPC service handle and register with RPCBIND service */

    /* Initialize the required parameters */
    s_principal = "myservice@aix1.ibm.com";   /* service@host */
    mech = "kerberosv5";
    r_time = 1000;

    /* Set the principal name */
    if(rpc_gss_set_svc_name(s_principal, mech, r_time, PROG, VERS) == FALSE)
    {
        fprintf(stderr,"\nError in rpc_gss_set_svc_name:\n");
        rpc_gss_get_error(&gss_error);
        fprintf(stderr,"rpc_gss_error: %d \nSystem_error: %d \n",
                gss_error.rpc_gss_error,gss_error.system_error);
        exit(EXIT_FAILURE);
     }
    svc_run();
    return 1;
}

/* following is the sample dispatch routine*/
static void sample_dispatch(struct svc_req *request, SVCXPRT *xprt)
{
    /* dispatch routine code */
}
```

Related information:

Transport Independent Remote Procedure Call

eXternal Data Representation Overview for Programming

rpc_gss_seccreate Subroutine

Purpose

Creates a security context.

Library

Network Services Library (**libnsl.a**)

Syntax

```
#include <rpc/rpcsec_gss.h>

AUTH *rpc_gss_seccreate(cl, s_principal, mech, s_type, qop, o_req, o_ret)
CLIENT *cl;
char *s_principal;
char *mech;
rpc_gss_service_t s_type;
char *qop;
rpc_gss_options_req_t *o_req;
rpc_gss_options_ret_t *o_ret;
```

Description

When making a remote procedure call using RPCSEC_GSS APIs, a security context must be created between the client and the server. The **rpc_gss_seccreate** subroutine uses the RPCSEC_GSS protocol to create a context. With the subroutine, you can specify the security mechanism that is used for context creation and thus for further client-server communication, security types and the quality of protection.

Parameters

Item	Description
cl	Represents a client handle that can be created using any of the client handle creation subroutines.
s_principal	Specifies a server principal of the form *service@host*. The *service* variable represents the service offered by a server and the *host* variable indicates the name of a machine on which the server resides (for example, `nfs@aix1.ibm.com`).
mech	Represents the supported security mechanism that is used for context creation and client-server communication (for example, `kerberosv5`).
s_type	Represents the type of service for the session that basically offers a level of protection. (for example, integrity and privacy).
qop	Represents the quality of protection. You can specify the parameter to select cryptographic algorithm.
o_req	Specifies the options that are passed to the GSS_API layer under the RPCSEC_GSS layer. If you specify the parameter with NULL, default parameters are used.
o_ret	Specifies the options that are returned by the GSS_API layer. If you do not want to see options, you can specify the parameter with NULL. The *o_ret* parameter is an output parameter.

Return Values

Item	Description
a security context handle of the **AUTH** type	successful
NULL	unsuccessful

You can use the **rpc_gss_get_error** subroutine to retrieve the error number.

Examples

In the following example, security context is created to have a secure communication between the client and the server.

```
#include <stdlib.h>
#include <rpc/rpc.h>
#include <rpc/rpcsec_gss.h>

main()
{
    CLIENT *client;
    char *s_principal;
    char *mech;
    rpc_gss_service_t s_type;
    char *qop;
```

```
    rpc_gss_options_ret_t o_ret;
    rpc_gss_error_t gss_error;

    /* Create client handle using any of the client handle creation routines*/

    /* Initialize the required parameters */
    s_principal = "myservice@aix1.ibm.com";   /* service@host */
    mech = "kerberosv5";
    s_type = 2;             /* 1: none, 2: integrity. 3: privacy */
    qop = "GSS_C_QOP_DEFAULT";
    o_ret.major_status = 0;
    o_ret.minor_status = 0;

    /* Create security context */
    client->cl_auth = rpc_gss_seccreate(client, s_principal,
                          mech, s_type, qop, NULL, &0_ret);
    if(client->cl_auth == NULL)
    {
         fprintf(stderr,"\nError in rpc_gss_seccreate:\n");
         rpc_gss_get_error(&gss_error);
         fprintf(stderr,"rpc_gss_error: %d \nSystem_error: %d \n"
                 ,gss_error.rpc_gss_error,gss_error.system_error);
         exit(EXIT_FAILURE);
     }
    /* Make a call to server */
}
```

Related information:

Transport Independent Remote Procedure Call

eXternal Data Representation Overview for Programming

rpc_gss_set_callback Subroutine

Purpose

Specifies callback routine for the context use.

Library

Network Services Library (**libnsl.a**)

Syntax

```
#include <rpc/rpcsec_gss.h>

bool_t rpc_gss_set_callback(cb)
struct rpc_gss_callback_t *cb;
```

Description

With the **rpc_gss_set_callback** subroutine, you can set a user-defined callback routine that is invoked when the context is used for the first time.

Parameters

Item	Description
cb	Points to a **rpc_gss_callback_t** structure.

The following is the definition of the **rpc_gss_callback_t** structure.

```
typedef struct {
u_int program;
u_int version;
bool_t (*callback )();
} rpc_gss_callback_t;
```

Item	Description
program	Represents the program number for which the context is established.
version	Represents version number for which context is established.

callback	Represents a user-defined callback routine that is in the following form:
	`bool_t callback ( req, deleg, gss_context, lock, cookie )`
	`struct svc_req *req;`
	`gss_cred_id_t deleg;`
	`gss_ctx_id_t gss_context;`
	`rpc_gss_lock_t *lock;`
	`void **cookie;`

The following table list the parameters of the callback routine.

Item	Description
req	Points to a received service-request structure.
deleg	Represents delegated credentials.
gss_context	Represents the Generic Security Services (GSS) context.
lock	Points to a **rpc_gss_lock_t** structure. You can use the parameter to enforce particular protection quality for that session.
cookie	Represents a 4-byte entity that an application can use in any manner.

Return Values

Item	Description
TRUE	successful
FALSE	unsuccessful

Examples

```
#include <stdlib.h>
#include <rpc/rpc.h>
#include <rpc/rpcsec_gss.h>
#define PROGNUM 0x3fffffffL
#define VERSNUM 0x1L

static void sample_dispatch(struct svc_req *, SVCXPRT *);

bool_t callback(struct svc_req *req, gss_cred_id_t deleg, gss_ctx_id_t gss_context,
                rpc_gss_lock_t *lock, void **cookie)
{
    fprintf(stdout,"\nIn callback routine!\n");
    return TRUE;
}

main()
{
    rpc_gss_callback_t cb;
    cb.program = PROG;
    cb.version = VERS;
    cb.callback = callback;
```

```
    /* Create RPC service handle and register with RPCBIND service */

    /* Set the principal name */

    if (!rpc_gss_set_callback(&cb)) {
         fprintf(stderr,"Error while setting callback\n");
         exit(1);
    }
    svc_run();
    return 1;
}

/* following is the sample dispatch routine*/
static void sample_dispatch(struct svc_req *request, SVCXPRT *xprt)
{
    /* Dispatch routine code */
}
```

Related information:

Transport Independent Remote Procedure Call

eXternal Data Representation Overview for Programming

rpc_gss_set_defaults Subroutine

Purpose

Changes the service type and quality of protection for client-server communication.

Library

Network Services Library (**libnsl.a**)

Syntax

```
#include <rpc/rpcsec_gss.h>

bool_t rpc_gss_set_defaults(auth_t, s_type, qop)
AUTH *auth_t;
rpc_gss_service_t s_type;
char *qop;
```

Description

While creating security context, you can specify the *s_type* and *qop* parameters for the transfer sessions. You can change the two parameters for next transfer sessions using the **rpc_gss_set_defaults** subroutine.

Parameters

Item	Description
auth_t	Represents an authentication handle returned by the **rpc_gss_seccreate** subroutine.
s_type	Represents the type of service for the session that basically offers a level of protection. (for example, integrity and privacy).
qop	Represents the quality of protection. You can specify the parameter to select cryptographic algorithm.

Return Values

Item	Description
TRUE	successful
FALSE	unsuccessful

You can use the **rpc_gss_get_error** subroutine to retrieve the error number.

Examples

The following example uses the **rpc_gss_set_defaults** subroutine to set service type and quality of protection after security context creation.

```
#include <stdlib.h>
#include <rpc/rpc.h>
#include <rpc/rpcsec_gss.h>

main()
{
    CLIENT *client;
    rpc_gss_service_t service_type;
    char *qop;
    rpc_gss_error_t gss_error;

    /* Create client handle using any of the client handle creation routines*/

    /* Create security context using rpc_gss_seccreate */

    /* Set service_type and quality of protection */
    if(rpc_gss_set_defaults(client->cl_auth,service_type,qop) == FALSE)
    {
        fprintf(stderr,"\nError in rpc_gss_set_defaults:\n");
        rpc_gss_get_error(&gss_error);
        fprintf(stderr,"rpc_gss_error: %d \nSystem_error: %d \n",
                gss_error.rpc_gss_error,gss_error.system_error);
        exit(EXIT_FAILURE);
    }
    /* Make a call to server */
}
```

Related information:

Transport Independent Remote Procedure Call

eXternal Data Representation Overview for Programming

rpc_gss_svc_max_data_length Subroutine

Purpose

Gets the maximum length of untransformed data allowed by the transport (a server-side version).

Library

Network Services Library (**libnsl.a**)

Syntax

```
#include <rpc/rpcsec_gss.h>

int rpc_gss_svc_max_data_length(req,max_tp_length)
struct svc_req *req;
int max_tp_length;
```

Description

Some transport types have restrictions on the maximum size of data that can be sent out in one data unit. After the security transformations on actual data, data length increases that depends on the selected security mechanism. Some applications need to know the actual length of untransformed data that is allowed before performing security transformations. You can get this maximum length of untransformed data using the **rpc_gss_svc_max_data_length** subroutine.

Parameters

Item	Description
req	Points to a received service-request structure.
max_tp_length	Represents the maximum length of data unit allowed by transport. This is an input parameter.

Return Values

On successful completion, the subroutine returns the maximum size of the allowed untransformed data.

Examples

```
#include <stdlib.h>
#include <rpc/rpc.h>
#include <tiuser.h>
#include <rpc/rpcsec_gss.h>

#define PROGNUM 0x3fffffffL
#define VERSNUM 0x1L

static void sample_dispatch(struct svc_req *, SVCXPRT *);
main()
{
    /* Create RPC service handle and register with RPCBIND service */
    /* Set the principal name */
    svc_run();
    return 1;
    }
    /* following is the sample dispatch routine*/
    static void sample_dispatch(struct svc_req *request, SVCXPRT *xprt)
    {
    int untransformed_data, max_tp_len;
    struct t_info info;

    /* Get info related to transport */
    if(t_getinfo(xprt->xp_fd,&info) !=0)
    {
         fprintf(stderr,"\nError in t_getinfo.\n");
         exit(1);
    }
    /* Get max data length allowed by transport */
    max_tp_len = info.tsdu;

    /* Get max data length allowed by transport */
    untransformed_data = rpc_gss_svc_max_data_length(request, max_tp_len);

    /* Send reply back to caller */

}
```

Related information:

Transport Independent Remote Procedure Call

eXternal Data Representation Overview for Programming

rpc_reg Subroutine
Purpose

Registers program number, version number, and procedure with the remote procedure call (RPC) service package.

Library

Network Services Library (**libnsl.a**)

Syntax

```
#include <rpc/rpc.h>

bool_t rpc_reg( prog, vers, proc, proc_name, iproc, oproc, nettype);
const rpcprog_t prog;
const rpcvers_t vers;
const rpcproc_t proc;
char *(*proc_name)(char *);
const xdrproc_t iproc;
const xdrproc_t oproc;
const char *nettype;
```

Description

The **rpc_reg** subroutine is a simplified-level API for transport-independent RPC that specify the transport type. Applications using this level do not need to explicitly create handles. The **rpc_reg** subroutine registers a program, a procedure, and a version with the RPC service package on all available transports that are specified by the *nettype* parameter. If you set the *nettype* parameter to a null value, the **rpc_reg** subroutine searches transports in **NETPATH** environment variable from left to right. If the value of the **NETPATH** environment variable is also null or unset, the **rpc_reg** subroutine searches in **netconfig** database from top to bottom. Whenever a service request from a client arrives, the program number, version number, and procedure number are mapped with registered services and respective procedure is called with appropriate parameters. The **rpc_reg** subroutine uses eXternal Data Representation (XDR) functions to encode and decode the parameters. The **rpc_reg** subroutine can register an individual procedure that can be a part of a large RPC service. A single procedure cannot be unregistered, but you can unregister the whole RPC service using the **svc_unreg** subroutine.

Parameters

Item	Description
prog	Specifies the program number.
vers	Specifies the version number.
proc	Specifies the procedure number.
proc_name	Points to a registered procedure, which returns a pointer to a static result. The parameter to the procedure is a pointer to the decoded procedure argument.
iproc	Specifies the XDR function to decode the parameters of the procedure.
oproc	Specifies the XDR function to encode the result of the procedure.
nettype	Defines a class of transports that can be used for a particular application.

Return Values

Item	Description
0	successful
-1	unsuccessful

Examples

In the following example, after the successful run of the **rpc_reg** subroutine, a service with PROG and VERS is registered with RPC service package on the tcp transport.

```
#include <stdlib.h>
#include <rpc/rpc.h>

#define PROG 0x3fffffffL
#define VERS 0x1L
#define PROC 0x1L

char * sample_proc(char *);

main()
{
  char *nettype;

  /* Specify transport type */
  nettype = "tcp";

  /* unregister the previous RPC service */
  svc_unreg(PROG,VERS);

  /* Register a single procedure at a time using rpc_reg() */
  if(rpc_reg(PROG,VERS,PROC,sample_proc,xdr_char,xdr_char,nettype) == -1)
  {
     fprintf(stderr,"\nError in rpc_reg!\n");
     svc_unreg(PROG,VERS);
     exit(EXIT_FAILURE);
  }

  /* Server waits for client's request to arrive */
  svc_run();

  return 0;
}


char * sample_proc(char *ptr)
{

  /* code for sample_proc procedure */

}
```

Related information:

Transport Independent Remote Procedure Call, IPv6 concerns for Transport Independent Remote Procedure Call

eXternal Data Representation Overview for Programming

rpcb_getaddr Subroutine

Purpose

Finds the address of a remote service.

Library

Network Services Library (**libnsl.a**)

Syntax

```
#include <rpc/rpc.h>

bool_t rpcb_getaddr (prognum, progver, nconf, svcaddr, host)
const rpcprog_t prognum;
const rpcvers_t progver;
const struct netconfig *nconf;
struct netbuf *svcaddr;
const char *host;
```

Description

The **rpcb_getaddr** subroutine is used to get the address of the remote service that is located on the host. This remote service is registered on the host with the specified program and version. The service is associated with the specified transport. On successful completion, the value of the *svcaddr* parameter is the address of the remote service.

Note: You must preallocate the *svcaddr* parameter before calling the **rpcb_getaddr** subroutine.

Parameters

Item	Description
prognum	Specifies the program number of the remote program.
progver	Specifies the version number of the remote program.
nconf	Specifies the protocol associated with the service.
svcaddr	Specifies the address of the remote service.
host	Specifies the host name on which the server resides.

Return Values

Item	Description
TRUE	successful
FALSE	unsuccessful

The subroutine returns FALSE in the following two conditions:

- The mapping does not exist on the host.
- The remote **rpcbind** service cannot be contacted.

The status of failure is set in the **rpc_createerr** global variable .

Error Codes

Item	Description
RPC_UNKNOWNHOST	The host name is not valid.
RPC_PROGNOTREGISTERED	The service is not registered with the remote **rpcbind** service.
RPC_FAILED	An unspecified error occurred.

Examples

In the following example, the **rpcb_getaddr** subroutine gets the address of remote service associated with the specified program, version, and **udp** transport.

```
#include <stdlib.h>
#include <rpcb_getaddrrpc/rpc.h>

#define ADDRBUFSIZE 255

int main()
{
```

```
    char hostname[255] ; /* The Remote host on which server is implemented */
    rpcprog_t program_number = 0x3fffffffL;
    rpcvers_t version_number = 0x1L;
    struct netbuf  nbuf;
    struct netconfig *nconf;
    char addrbuf[ADDRBUFSIZE];

    /* Get pointer to struct netconfig for udp transport */
    nconf = getnetconfigent("udp");
    if (nconf == (struct netconfig *) NULL) {
          fprintf(stdout, "\nerror in getnconfigent!\n");
          exit(1);
    }

    nbuf.len = 0;
    nbuf.maxlen = ADDRBUFSIZE;
    nbuf.buf = addrbuf;

    if (!rpcb_getaddr(program_number, version_number, nconf, &nbuf, hostname)) {
        fprintf(stdout, "\nerror in getnconfigent!\n");
          exit(1);
      }

    return 0;
}
```

Related information:

Transport Independent Remote Procedure Call, IPv6 concerns for Transport Independent Remote Procedure Call

eXternal Data Representation Overview for Programming

rpcb_getmaps Subroutine

Purpose

Returns program-to-address mappings.

Library

Network Services Library (**libnsl.a**)

Syntax

```
#include <rpc/rpc.h>

struct rpcblist *rpcb_getmaps(nconf, host)
const struct netconfig *nconf;
const char *host;
```

Description

The **rpcb_getmaps** subroutine returns a list of remote procedure call (RPC) program-to-address mappings for on a remote host. The *host* parameter represents the host from which the list of mappings is returned. The remote **rpcbind** service on the host is contacted by the transport specified by the *nconf* parameter. The subroutine returns a null value if the remote **rpcbind** service cannot be contacted.

Parameters

Item	Description
nconf	Specifies the protocol associated with the service.
host	Specifies the host name on which the server resides.

Return Values

Item	Description
a pointer to the **rpcblist** structure	successful
FALSE	unsuccessful

Error Codes

Item	Description
RPC_UNKNOWNHOST	The host name is not valid.
RPC_N2AXLATEFAILURE	The value of the *nconf* argument is not valid.

Examples

```
#include <stdlib.h>
#include <rpc/rpc.h>

int main()
{
    /* The Remote host on which server is implemented */
    char hostname[255] ;
    struct netconfig *nconf;
    struct rpcblist *rpclist = NULL;

    /* Get pointer to struct netconfig for udp transport */
    nconf = getnetconfigent("udp");
    if (nconf == (struct netconfig *) NULL) {
         fprintf(stdout, "\nerror in getnconfigent!\n");
         exit(1);
    }
    rpclist = (struct rpcblist *)rpcb_getmaps(nconf, hostname);
    if (rpclist == NULL) {
          fprintf(stderr,"could not get the rpcblist on remote host\n");
          clnt_pcreateerror("rpcb_getmap:");
          exit(1);
    }
    return 0;
}
```

Related information:

Transport Independent Remote Procedure Call, IPv6 concerns for Transport Independent Remote Procedure Call

eXternal Data Representation Overview for Programming

rpcb_gettime Subroutine

Purpose

Returns the time on a remote host.

Library

Network Services Library (**libnsl.a**)

Syntax

```
#include <rpc/rpc.h>
```

```
bool_t rpcb_gettime(host, time_p)
const char *host;
time_t *time_p;
```

Description

You can obtain the time on a remote host using the **rpcb_gettime** subroutine. The time is returned by the *time_p* parameter. You must preallocate the *time_p* parameter before calling this subroutine. If the host is specified with a null value, this subroutine returns the time on the local machine from which the subroutine is called. Generally, the **rpcb_gettime** subroutine is used to synchronize the time between clients and servers. This subroutine is particularly needed for secure remote procedure call (RPC) applications in which clients and servers must be synchronized.

Parameters

Item	Description
host	Specifies the host name on which the server resides.
time_p	Specifies the time on the host.

Return Values

Item	Description
TRUE	successful
FALSE	unsuccessful

Examples

```
#include <stdlib.h>
#include <rpc/rpc.h>

int main()
{
    char hostname[255] ; /* The Remote host */
    time_t        time_p = 0;

    if( rpcb_gettime(hostname, &time_p)== FALSE ) {
          fprintf(stderr,"rpcb_gettime failed");
          exit(1);
     }
     return 0;
}
```

Related information:

Transport Independent Remote Procedure Call, IPv6 concerns for Transport Independent Remote Procedure Call

eXternal Data Representation Overview for Programming

rpcb_rmtcall Subroutine

Purpose

Instructs the **rpcbind** service on a remote host to make a remote procedure call (RPC) on behalf of the caller to a procedure on that host.

Library

Network Services Library (**libnsl.a**)

Syntax

```
#include <rpc/rpc.h>
```

```
enum clnt_stat rpcb_rmtcall(nconf, host, prognum, progver, procnum, in_proc, input, out_proc, output,
 t_out, svcaddr)
const struct netconfig *nconf
const char *host
const rpcprog_t prognum;
const rpcvers_t versnum;
const rpcproc_t procnum;
const xdrproc_t in_proc;
const caddr_t input;
const xdrproc_t out_proc;
const caddr_t output;
const struct timeval t_out;
struct netbuf *svcaddr
```

Description

The **rpcb_rmtcall** subroutine is an interface to the **rpcbind** service. The subroutine instructs the **rpcbind** service on the remote host to make an RPC call on behalf of the caller to a procedure on that host. The **netconfig** structure must correspond to a connectionless transport. You can use this subroutine for a **ping** program because the subroutine performs the lookup and call in one step.

Parameters

Item	Description
host	Specifies the host name on which the server resides.
prognum	Specifies the program number of the remote host.
progver	Specifies the version number of the remote host.
procnum	Specifies the procedure number on the remote host.
in_proc	Specifies the eXternal Data Representation (XDR) procedure to encode the input parameters.
out_proc	Specifies the XDR procedure to decode the output results.
output	Specifies the address of output parameters.
t_out	Specifies the timeout value.
input	Specifies the address of input parameters.
nconf	Specifies the protocol associated with the service.
svcaddr	Specifies the address of the remote service when the procedure succeeds.

Return Values

Item	Description
RPC_SUCCESS	successful
nonzero	unsuccessful

Error Codes

Item	Description
RPC_TIMEDOUT	• The **netconfig** structure corresponds to a connection-oriented transport. • The version number is not valid. • The program number is not valid. • The procedure number is not valid.

Examples

```
#include <rpc/rpc.h>

int main()
{
    struct netconfig *nconf;
    rpcprog_t PROGNUM = 0x3fffffffL;
    rpcvers_t PROGVER = 0x1L;
```

```
    rpcproc_t PROCNUM = 0x1L;
    struct timeval timeout = {25,0};
    int req , resp;
    enum clnt_stat cs;
    char host[255]; /* Remote host name */

    req = 5 ; /* initialise input parameter to a valid value */

    /* Set the netconfig structure for tcp transport */
    if ((nconf = getnetconfigent("tcp")) ==(struct netconfig *) NULL) {
        printf("getnetconfigent() failed");
        exit(1);
    }

    /* Call the remote procedure using rpcb_rmtcall() */
    cs = rpcb_rmtcall(nconf,host,PROGNUM,PROGVER,PROCNUM,(xdrproc_t)xdr_int,
        (caddr_t)&req, (xdrproc_t)xdr_int, (caddr_t)&resp, timeout, nbuf);

    /* Check for the return status */
    if(cs != RPC_SUCCESS)
    {
        printf("rpcb_rmtcall() failed");
        exit(1);
    }

    return 0;
}
```

Related information:

Transport Independent Remote Procedure Call, IPv6 concerns for Transport Independent Remote Procedure Call

eXternal Data Representation Overview for Programming

rpcb_set Subroutine

Purpose

Establishes a mapping between the program, version, **netconfig** structure and the service address.

Library

Network Services Library (**libnsl.a**)

Syntax

```
#include <rpc/rpc.h>

bool_t rpcb_set(prognum, progver, nconf, svcaddr)
const rpcprog_t prognum;
const rpcvers_t progver;
const struct netconfig *nconf;
struct netbuf *svcaddr
```

Description

The **rpcb_set** subroutine is used to establish a mapping of triplet (the program number, version number and the **nc_netid** field of the *nconf* argument) to the service on a remote host. The mapping is established on the **rpcbind** service of the machine. The *svcaddr* parameter specifies the address of the remote service. The **nc_netid** field of the **netconfig** structure identifies the network identifier defined by the **netconfig** database.

Note: This subroutine fails if it tries to create a mapping that exists on the machine.

Parameters

Item	Description
prognum	Specifies the program number of the remote program.
progver	Specifies the version number of the remote program.
nconf	Specifies the protocol associated with the service.
svcaddr	Specifies the address of the remote service.

Return Values

Item	Description
TRUE	successful
FALSE	unsuccessful

Error Codes

The **rpcb_set** subroutine returns failure if one or more of the following codes are true.

Item	Description
RPC_UNKNOWNPROTO	The value of the *netconfig* argument is not valid.
RPC_UNKNOWNADDR	The remote service address is not valid.

Examples

```
#include <stdlib.h>
#include <rpc/rpc.h>

int main()
{
    rpcprog_t PROGNUM ;
    rpcvers_t PROGVER ;
    struct netconfig *nconf ;
    struct netbuf *nbuf;
    struct t_bind *bind_addr = NULL;

    /* Get netconfig structure corresponding to tcp transport */
    if ((nconf = getnetconfigent("tcp")) == (struct netconfig *) NULL)
    {
        fprintf(stderr, "getnetconfigent failed");
        exit(1);
    }
    /*
     * Code to open and bind file descriptor to bind_addr address
     */
    nbuf = &bind_addr->addr;
    if( rpcb_set(PROGNUM, PROGVER, nconf, nbuf) == FALSE ) {
        fprintf(stderr,"rpcb_set() failed");
        exit(1);
    }
    svc_run();
    return 0;
}
```

Related information:

Transport Independent Remote Procedure Call, IPv6 concerns for Transport Independent Remote Procedure Call

eXternal Data Representation Overview for Programming

rpcb_unset Subroutine

Purpose

Destroys the mapping between the program, version, **netconfig** structure and the service address.

Library

Network Services Library (**libnsl.a**)

Syntax

```
#include <rpc/rpc.h>

bool_t rpcb_unset(progNum, progver, nconf)
const rpcprog_t progNum;
const rpcvers_t progver;
const struct netconfig *nconf;
```

Description

The **rpcb_unset** subroutine destroys a mapping of triplet (program number, version number, and the **nc_netid** field of the *nconf* argument) to the address of a remote service. The mapping is destroyed from the **rpcbind** service of the machine. The **nc_netid** field of the **netconfig** structure identifies the network identifier defined by the **netconfig** database.

Parameters

Item	Description
progNum	Specifies the program number of the remote program.
progver	Specifies the version number of the remote program.
nconf	Specifies the protocol associated with the service.

Return Values

Item	Description
TRUE	successful
FALSE	unsuccessful

Error Codes

The **rpcb_unset** subroutine returns failure if one or more of the following codes are true.

Item	Description
RPC_UNKNOWNPROTO	The value of the *netconfig* argument is not valid.
RPC_UNKNOWNADDR	The remote service address is not valid.

Examples

```
#include <rpc/rpc.h>
#include <stdio.h>

int main()
{
    rpcprog_t PROGNUM ;
    rpcvers_t PROGVER ;
    struct netconfig *nconf ;
    struct netbuf *nbuf;
    struct t_bind *bind_addr = NULL;

    /* Get netconfig structure corresponding to tcp transport */
    if ((nconf = getnetconfigent("tcp")) == (struct netconfig *) NULL) {
        fprintf(stderr, "getnetconfigent failed");
        exit(1);
    }
    /*
     * Code to open and bind file descriptor to bind_addr address
```

```
     */
    nbuf = &bind_addr->addr;
    if( rpcb_set(PROGNUM, PROGVER, nconf, nbuf) == FALSE ) {
        fprintf(stderr,"rpcb_set() failed");
        exit(1);
    }
    rpcb_unset(PROGNUM, PROGVER, nconf);
    svc_run();
    return 0;
}
```

Related information:

Transport Independent Remote Procedure Call, IPv6 concerns for Transport Independent Remote Procedure Call

eXternal Data Representation Overview for Programming

S

The following RPC subroutines begin with the letter s.

svc_auth_reg Subroutine

Purpose

Registers an authentication routine with the dispatch mechanism.

Library

Network Services Library (**libnsl.a**)

Syntax

```
#include <rpc/rpc.h>

int svc_auth_reg(cr_flavor, auth_handler);
const int cr_flavor;
enum auth_stat(*auth_handler)(struct svc_req *, struct rpc_msg *);
```

Description

The **svc_auth_reg** subroutine registers an authentication routine with the dispatch mechanism so that service requests from clients can be authenticated with the specified authentication type. With the subroutine, you can add new authentication types to applications without changing the existing library. Call the **svc_auth_reg** subroutine after the service registration and before calling the **svc_run** subroutine. When remote procedure call (RPC) credentials are checked, the corresponding authentication handler is invoked. When registered, the authentication handler cannot be changed or deleted.

Parameters

Item	Description
cr_flavor	Specifies the authentication type.
auth_handler	Specifies the authentication handler that has two parameters and returns a valid value of the **auth_stat** type.

Return Values

Item	Description
0	The subroutine completed successfully.
1	An authentication handler has already registered for the specified authentication type.
-1	An error occurred.

Examples

```
#include <stdlib.h>
#include <rpc/rpc.h>

#define PROG 0x3fffffffL
#define VERS 0x1L

static void sample_dispatch();
enum auth_stat auth_handler(struct  svc_req  *rqst, struct  rpc_msg  *msg);

main()
{
      char *nettype;
      int no_of_handles,cr_flavor;

      /* Specify transport type */
      nettype = "tcp";


      /* Create RPC service handle and register with RPCBIND service */
      if((no_of_handles = svc_create(sample_dispatch, PROG, VERS,nettype)) == 0)
      {
              fprintf(stdout,"Error in svc_create!");
              exit(EXIT_FAILURE);
      }

      /* select desired cr_flavor */
      cr_flavor = AUTH_NONE;

      /* Register an authentication routine with the dispatch mechanism */
      if(svc_auth_reg(cr_flavor, auth_handler) == -1)
      {
              fprintf(stdout,"Error in svc_auth_reg!");
              exit(EXIT_FAILURE);
       }

       svc_run();
       return 0;
}

       /* following is the sample dispatch routine*/
       static void sample_dispatch(struct svc_req *request, SVCXPRT *xprt)
       {

              /*  dispatch routine code  */

       }

/*following is the sample authentication handler */
enum auth_stat auth_handler(struct  svc_req  *rqst, struct  rpc_msg  *msg)
{
       fprintf(stdout, "Entering authentication handler\n");
       return AUTH_OK;  /* auth_stat value */
}
```

Related information:

Transport Independent Remote Procedure Call, IPv6 concerns for Transport Independent Remote Procedure Call

eXternal Data Representation Overview for Programming

svc_control Subroutine
Purpose

Retrieves information about a client call (a server-side routine).

Library

Network Services Library (**libnsl.a**)

Syntax

```
#include <rpc/rpc.h>

bool_t svc_control(svc, rq, in);
SVCXPRT *svc;
u_int rq;
void *in;
```

Description

The **svc_control** subroutine is a top-level subroutine for transport-independent remote procedure calls (TI_PRC), giving you greater control over communication parameters. The subroutine retrieves the transaction ID of a client call. With the RPC service handle and the operation type specified by the *svc* and *rq* parameters, the subroutine retrieves information with a pointer specified by the *in* parameter.

Parameters

Item	Description
svc	Specifies the RPC service handle of a registered service.
rq	Represents an operation type. You can specify the **SVCGET_XID** operation type: **SVCGET_XID** This operation returns transaction ID of a client call. The transaction ID uniquely identifies a client request with the version, program number, procedure number and client. The transaction ID is retrieved from the RPC service handle. Only the RPC service handle of a connection-oriented or connectionless transport has transaction ID. For other handles, the function returns false.
in	Points to information that can be retrieved.

Return Values

Item	Description
TRUE	successful
FALSE	unsuccessful

Examples

```
#include <stdlib.h>
#include <rpc/rpc.h>
#include <netconfig.h>

#define PROG 0x3fffffffL
#define VERS 0x1L

static void sample_dispatch();

main()
{
    int in;
    struct netconfig *nconf;
    SVCXPRT *svc_handle;
```

```
    svc_unreg(PROG, VERS);

    /* Get transport type*/

    nconf = getnetconfigent("tcp");
    if (nconf == (struct netconfig *) NULL)
    {
           fprintf(stderr, "getnetconfigent failed!\n");
           exit(EXIT_FAILURE);
    }

    /* Create RPC service handle and register with RPCBIND service */

    if((svc_handle = svc_tp_create(sample_dispatch, PROG, VERS, nconf)) == (SVCXPRT *)NULL)
    {
           fprintf(stderr,"Error in svc_tp_create!");
           exit(EXIT_FAILURE);
     }

     svc_run();
     return 0;
}
/* following is the sample dispatch routine*/
static void sample_dispatch(struct svc_req *request, SVCXPRT *xprt)
{
     int in=0;
     /*   dispatch routine code  */
     /* Retrieve the information about the registered service */
     if(svc_control(xprt,SVCGET_XID,(void *)&in) == FALSE)
     {
            fprintf(stderr,"Error in svc_control!");
            exit(EXIT_FAILURE);
     }

}
```

Related information:

Transport Independent Remote Procedure Call, IPv6 concerns for Transport Independent Remote Procedure Call

eXternal Data Representation Overview for Programming

svc_create Subroutine

Purpose

Creates remote procedure call (RPC) service handles for all specified transports.

Library

Network Services Library (**libnsl.a**)

Syntax

```
#include <rpc/rpc.h>

int svc_create(dispatch, prog, vers, nettype);
void (*dispatch) (const struct svc_req*, const SVCXPRT*);
const rpcprog_t prog;
const rpcvers_t vers;
const char *nettype;
```

Description

This subroutine is a top-level API for transport-independent remote procedure calls (TI_PRC), giving you greater control over communication parameters. This subroutine creates RPC service handles and

registers with RPC service package with the specified program and version for all transports of the class specified by the *nettype* parameter. If the value of the *nettype* parameter is NULL, the subroutine searches transports in the NETPATH environment variable from left to right. If the value of the NETPATH variable is also NULL or unset, the subroutine searches in the **netconfig** database from top to bottom. After creating the handle, call the **svc_run** subroutine that waits for a service request to arrive. When a service request from the client for the specified program and version arrives, a dispatch subroutine is called.

Parameters

Item	Description
dispatch	Specifies a subroutine that is called when a service request arrives.
prog	Represents the program number.
vers	Represents the version number.
nettype	Defines a class of transports that can be used for a particular application.

Return Values

Item	Description
the number of RPC service handles that are created	successful
0	unsuccessful

Examples

```
#include <stdlib.h>
#include <rpc/rpc.h>

#define PROG 0x3fffffffL
#define VERS 0x1L


static void sample_dispatch();

main()
{
      char *nettype;
      int no_of_handles;

      /* initialize transport type */
      nettype = "tcp";

      /* Create RPC service handle and register with RPCBIND service */

      if((no_of_handles = svc_create(sample_dispatch, PROG, VERS,nettype)) == 0)
      {
            fprintf(stdout,"Error in svc_create!");
            exit(EXIT_FAILURE);
      }

      svc_run();

      return 0;
}

/* following is the sample dispatch routine*/
static void sample_dispatch(struct svc_req *request, SVCXPRT *xprt)
{

     /*   dispatch routine code  */

       }
```

Related information:

Transport Independent Remote Procedure Call, IPv6 concerns for Transport Independent Remote Procedure Call

eXternal Data Representation Overview for Programming

svc_destroy Macro

Important: The macro is exported from both the **libc** and the **libnsl** libraries.

svc_destroy Macro Exported from the libc Library

Purpose

Destroys a Remote Procedure Call (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 an 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.

Parameters

Item	Description
xprt	Points to the RPC service transport handle.

svc_destroy Macro Exported from the libnsl Library

Purpose

Destroys a remote procedure call (RPC) service handle.

Library

Network Services Library (**libnsl.a**)

Syntax

```
#include <rpc/rpc.h>
void svc_destroy(xprt)
SVCXPRT *xprt;
```

Description

This subroutine is a top-level API for transport-independent remote procedure calls (TI_PRC), giving you greater control over communication parameters. This subroutine destroys the RPC service handle that is

created when registering the service. This subroutine deallocates all of the private data structures that are allocated when the handle is created and the handle itself.

Parameters

Item	Description
xprt	Points to the RPC service handle.

Examples

```
#include <stdlib.h>
#include <rpc/rpc.h>
#include <netconfig.h>

#define PROG 0x3fffffffL
#define VERS 0x1L

static void sample_dispatch();

main()
{

     SVCXPRT *svc_handle;
     int fd;
     struct netconfig *nconf;

     /* Get required file descriptor, and transport type */

     /* Create RPC service handle */
     if((svc_handle = svc_tli_create(fd, nconf, 0, 0, 0)) == (SVCXPRT *)NULL)
     {
          fprintf(stdout,"Error in svc_tli_create!");
          exit(EXIT_FAILURE);
     }

     /* Register RPC service using RPC service handle with RPCBIND package*/
     if(svc_reg(svc_handle, PROG, VERS, sample_dispatch, nconf) == 0)
     {
          fprintf(stdout,"Error in svc_reg!");

          /*Destroy the RPC service handle as service is not registered*/
          svc_destroy(svc_handle);
          exit(EXIT_FAILURE);
      }

      svc_run();
      return 0;
      }
/* following is the sample dispatch routine*/
static void sample_dispatch(struct svc_req *request, SVCXPRT *xprt)
{
         /*   dispatch routine code  */

}
```

svc_dg_create Subroutine
Purpose

Creates a remote procedure call (RPC) service handle for a connectionless transport.

Library

Network Services Library (**libnsl.a**)

Syntax

```
#include <rpc/rpc.h>

SVCXPRT *svc_dg_create(fd, sendsize, recvsize)
int fd;
const uint_t sendsize;
const uint_t recvsize;
```

Description

The **svc_dg_create** subroutine is a bottom-level API for transport-independent remote procedure calls (TI_PRC). Bottom-level APIs provide a full control over the transport options. The subroutine creates an RPC service handle for a connectionless transport. You can use this handle for procedures that accept a small number of arguments or return small values, because connectionless messages can hold limited amount of data. This subroutine does not register a server with RPC services because the program number and version number are not specified.

Parameters

Item	Description
fd	Indicates an open file descriptor that is bound.
sendsize	Specify the send buffer size. If the value is set to 0, the default size for that transport is used.
recvsize	Specify the receive buffer size. If the value is set to 0, the default size for that transport is used.

Return Values

Item	Description
an RPC service handle	successful
NULL	unsuccessful

Examples

```
#include <stdlib.h>
#include <rpc/rpc.h>

main()
{
      SVCXPRT *svc_handle;      /* server handle */
      int fd;

      /* Get proper file descriptor */

      /* sendsize and recvsize are 0, thus default size will be chosen */

      if((svc_handle = svc_dg_create(fd, 0, 0))==(SVCXPRT *)NULL)
      {
                fprintf(stdout,"Error in svc_dg_create!");
                exit(EXIT_FAILURE);
      }

      /* Register RPC service */

      svc_run();
      return 0;
}
```

Related information:

Transport Independent Remote Procedure Call, IPv6 concerns for Transport Independent Remote Procedure Call

eXternal Data Representation Overview for Programming

svc_dg_enablecache Subroutine
Purpose

Allocates the duplicate request cache for the service endpoint.

Library

Network Services Library (**libnsl.a**)

Syntax

```
#include <rpc/rpc.h>

int svc_dg_enablecache(xprt, cachesz )
SVCXPRT *xprt;
const uint_t cachesz;
```

Description

This subroutine allocates the duplicate request cache for the RPC service handle specified by the *xprt* parameter that can hold cache entries whose number are specified by the *cachesz* parameter. Request caching is useful for operations that cannot be performed twice with the same result. When the cache mechanism is enabled, the mechanism cannot be disabled.

Parameters

Item	Description
xprt	Indicates the RPC service handle.
cachesz	Indicates the number of cache entries.

Return Values

Item	Description
1	successful
0	unsuccessful

Examples

```
#include <stdlib.h>
#include <rpc/rpc.h>

#define PROG 0x3fffffffL
#define VERS 0x1L

static void sample_dispatch();

main()
{

     SVCXPRT *svc_handle;
     struct netconfig *nconf;

     svc_unreg(PROG,VERS);

     /* Get desired transport type */
     nconf = getnetconfigent("tcp");
     if(nconf == NULL)
     {
          fprintf(stderr,"\nError in getnetconfigent!\n");
          exit(EXIT_FAILURE);
     }
```

```
     /* Create svc_handle */
     svc_handle = svc_tli_create(RPC_ANYFD, nconf, NULL, 0, 0);
     if(svc_handle == (SVCXPRT *)NULL)
     {
           fprintf(stdout,"Error in svc_tli_create!");
           exit(EXIT_FAILURE);
     }

     /* Register dispatch routine for prog and vers with RPCBIND service */

     if(svc_reg(svc_handle, PROG, VERS, sample_dispatch,nconf) == 0)
     {
           fprintf(stdout,"Error in svc_reg!");
           exit(EXIT_FAILURE);
     }

     /* This subroutine allocates duplicate cache */
     if( svc_dg_enablecache(svc_handle,5) == 0)
     {
           fprintf(stdout,"Error in svc_dg_enablecache!");
           exit(EXIT_FAILURE);
     }

     svc_run();

     fprintf(stderr,"\nError in svc_run!\n");
     exit(EXIT_FAILURE);
}

/* following is the sample dispatch routine*/
static void sample_dispatch(struct svc_req *request, SVCXPRT *xprt)
{
     /*   dispatch routine code  */

}
```

Related information:

Transport Independent Remote Procedure Call, IPv6 concerns for Transport Independent Remote Procedure Call

eXternal Data Representation Overview for Programming

svc_done Subroutine

Note: You can set the server in multithreaded mode using the **rpc_control** subroutine.

Purpose

Frees the resources allocated to service a client request.

Library

Network Services Library (**libnsl.a**)

Syntax

```
#include <rpc/rpc.h>
int svc_done(xprt )
SVCXPRT *xprt;
```

Description

The **svc_done** subroutine frees the resources allocated to service a client request. The subroutine is used when the server is in the user-multithreaded mode. If used in the single-threaded mode or in the

AUTO-MT mode, the subroutine has no effect. This subroutine is normally called in a service procedure before the return, when the remote procedure call (RPC) request has been serviced or when any abnormal condition occurs.

Parameters

Item	Description
xprt	Identifies the service handle.

Return Values

Examples

```
#include <stdlib.h>
#include <rpc/rpc.h>
#include <pthread.h>

rpcprog_t prognum;
rpcvers_t progver;

/* create threads to serv multiple client requests */
void * thread_func(void * xprt)
{
    int         result = 3;

    if(svc_sendreply((SVCXPRT *) xprt, (xdrproc_t) xdr_int,
                          (char *) &result) == FALSE)
    svcerr_systemerr(xprt);


    /* call to svc_done which frees resources allocated */
    svc_done((SVCXPRT *)xprt);

    pthread_exit(0);
    }

    /* dispatch routine */
    static void  dispatch(struct svc_req * request, SVCXPRT * xprt)
    {
         int  ret;
         pthread_t  tid;

         if((ret = pthread_create(&tid,NULL,thread_func,
                                  (void *)xprt)) != 0)
         {
              fprintf(stderr,"\nError in pthread_create.\n");
              exit(2);
         }
    }

int main()
{
    int num,mode;

    prognum = 0x3fffffffL;
    progver = 0x1L;

    svc_unreg(prognum,progver);

    /* register RPC service */
    num = svc_create(dispatch, prognum, progver, "tcp");
    if (num == 0)
    {
         fprintf(stderr, "Error in svc_create.\n");
         exit(EXIT_FAILURE);
```

```
    }

    /* server in USER-MT mode */

    mode = RPC_SVC_MT_USER;
    if(rpc_control(RPC_SVC_MTMODE_SET,&mode) == FALSE)
    {
         fprintf(stderr,"\nError in rpc_control!\n");
         exit(EXIT_FAILURE);
    }

    svc_run();
    exit(1);
}
```

Related information:

Transport Independent Remote Procedure Call, IPv6 concerns for Transport Independent Remote Procedure Call

eXternal Data Representation Overview for Programming

svc_exit Subroutine

Important: The subroutine is exported from both the **libc** and the **libnsl** libraries.

svc_exit Subroutine Exported from the libc Library

Purpose

Causes the **svc_run** service loop to terminate and return.

Library

Network Services Library (**libnsl.a**)

Syntax

```
#include <rpc/rpc.h>

void svc_exit (void);
```

Description

The **svc_exit** subroutine causes the **svc_run** loop to terminate and return to the caller. This subroutine can be called by a service procedure. The call causes all service threads to exit and destroys all server services. Callers must reestablish all services if they wish to resume server activity.

Related Information

The "svc_run Subroutine" on page 373.

svc_exit Subroutine Exported from the libnsl Library

Purpose

Destroys all remote procedure call services registered by the server.

Library

Network Services Library (**libnsl.a**)

Syntax

```
#include <rpc/rpc.h>

void svc_exit (void);
```

Description

The **svc_exit** subroutine destroys all RPC services registered by server program and forces the **svc_run** subroutine to return. This subroutine has a global scope and thus all server activities are stopped. To restart the RPC server activities, you must reregister RPC services.

Examples

```
#include <stdlib.h>
#include <rpc/rpc.h>
#include <pthread.h>

#define PROG 0x3fffffffL
#define VERS 0x1L

static void sample_dispatch();

main()
{

    SVCXPRT *svc_handle;
    struct netconfig *nconf;

    /* Create svc_handle using server handle creation routines and get transport type */

    /* Register dispatch routine for program number and version number with RPCBIND service */

    svc_run();
    fprintf(stdout,"\nAfter svc_run()!\n");

     return 0;
}

/* following is the sample dispatch routine*/
static void sample_dispatch(struct svc_req *request, SVCXPRT *xprt)
{

    /*  code for dispatch routine */
    svc_exit();

}
```

svc_fd_create Subroutine
Purpose

Creates a remote procedure call (RPC) service handle on an open and bound file descriptor.

Library

Network Services Library (**libnsl.a**)

Syntax

```
#include <rpc/rpc.h>
SVCXPRT *svc_fd_create(fd, sendsize, recvsize)
int fd;
const uint_t sendsize;
const uint_t recvsize;
```

Description

The **svc_fd_create** subroutine is a bottom-level API for transport-independent remote procedure calls (TI_PRC). Bottom-level APIs provide a full control over the transport options. This subroutine creates a service handle over a given file descriptor. The file descriptor must be open and bound that is connected to a connection-oriented transport. This subroutine does not register a server with RPC services because the program number and version number are not specified. The size of the send and receive buffers can be specified by the *sendsize* and *recvsize* parameters. If the values of the *sendsize* and *recvsize* parameters are set to 0, the default size is used for buffers.

Parameters

Item	Description
fd	Indicates an open file descriptor that is bound.
sendsize	Specify the send buffer size. If the value is set to 0, the default size for that transport is used.
recvsize	Specify the receive buffer size. If the value is set to 0, the default size for that transport is used.

Return Values

Item	Description
an PRC service handle	successful
NULL	unsuccessful

Examples

```
#include <stdlib.h>
#include <rpc/rpc.h>

main()
{
        SVCXPRT *svc_handle;      /* server handle */
        int fd;                   /* file descriptor */
        /* Get proper file descriptor */

        /* sendsize and recvsize are 0, thus default size will be chosen */

        if((svc_handle = svc_fd_create(fd, 0, 0))==(SVCXPRT *)NULL)
        {
                    fprintf(stdout,"Error in svc_fd_create!");
                    exit(EXIT_FAILURE);
        }

        /* Register RPC service */

        svc_run();
        return 0;
}
```

Related information:

Transport Independent Remote Procedure Call, IPv6 concerns for Transport Independent Remote Procedure Call

eXternal Data Representation Overview for Programming

svc_fdset Global Variable

Note: Do not pass the address of this variable to any of the select subroutines. Instead pass a copy of the address.

Purpose

Indicates the read-file descriptor bit mask of the remote procedure call (RPC) server.

Library

Network Services Library (**libnsl.a**)

Syntax

```
#include <rpc/rpc.h>
fd_set svc_fdset
```

Description

The **svc_fdset** global variable is a read-only global variable, indicating the read-file descriptor bit mask of the remote procedure call (RPC) server. The variable is normally used when the server handles RPC requests asynchronously (the **svc_run** subroutine is not used). The value of the **svc_fdset** global variable might change after calls to the **svc_getreqset**, **svc_getreq_poll** or other RPC service handle creation subroutines. The value of the **svc_fdset** global variable is limited to 1024. Servers running in MT mode cannot read this variable. Instead they can create auxiliary threads to handle asynchronous requests.

Related information:

Transport Independent Remote Procedure Call

eXternal Data Representation Overview for Programming

svc_freeargs Macro

Important: The macro is exported from both the **libc** and the **libnsl** libraries.

svc_freeargs Macro Exported from the libc Library

Purpose

Frees data allocated by the Remote Procedure Call/eXternal Data Representation (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 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

Item	Description
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.

svc_freeargs Macro Exported from the libnsl Library

Purpose

Frees data allocated by the Remote Procedure Call/eXternal Data Representation (RPC/XDR) system.

Library

Network Services Library (**libnsl.a**)

Syntax

```
#include <rpc/rpc.h>

svc_freeargs ( xprt,  inproc, in)
SVCXPRT *xprt;
xdrproc_t inproc;
caddr_tin;
```

Description

The **svc_freeargs** macro frees data allocated by the 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

Item	Description
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.

Examples

```
#include <stdlib.h>
#include <rpc/rpc.h>

#define PROG 0x3fffffffL
#define VERS 0x1L

static void sample_dispatch();

main()
{
    char *nettype = "tcp";
    int no_of_handles;

    /* Create RPC service handle and register with RPCBIND service */

    if((no_of_handles = svc_create(sample_dispatch, PROG, VERS,nettype)) == 0)
    {
         fprintf(stdout,"Error in svc_create!");
         exit(EXIT_FAILURE);
    }

    svc_run();
    return 0;
}
```

```
/* following is the sample dispatch routine*/
static void sample_dispatch(struct svc_req *request, SVCXPRT *xprt)
{
    int args,result;

    /* decode argument from client using xdr_int() */
    if (svc_getargs(xprt, (xdrproc_t)xdr_int,(caddr_t)&args) == FALSE)
    {
        svcerr_decode(xprt);
        fprintf(stdout,"Error in svc_getargs!");
        return;
    }

    /* call service procedure */

    /* free allocated  data */
    if (svc_freeargs(xprt, (xdrproc_t)xdr_int,(caddr_t)&args) == FALSE)
    {
        fprintf(stdout,"Error in svc_freeargs!");
        return;
    }
    /* send reply to client */
    if(!svc_sendreply(xprt,(xdrproc_t)xdr_int,(caddr_t)&result))
    {
        fprintf(stdout,"Error in svc_sendreply!");
        svcerr_systemerr(xprt);
    }
}
```

svc_getargs Macro

Important: The macro is exported from both the **libc** and the **libnsl** libraries.

svc_getargs Macro Exported from the libc Library

Purpose

Decodes the arguments of a Remote Procedure Call (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 an RPC request associated with the RPC service transport handle.

Parameters

Item	Description
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.

svc_getargs Macro Exported from the libnsl Library

Purpose

Decodes the arguments of a Remote Procedure Call (RPC) request.

Library

Network Services Library (**libnsl.a**)

Syntax

```
#include <rpc/rpc.h>

svc_getargs (xprt, inproc, in)
SVCXPRT *xprt;
xdrproc_t inproc;
caddr_t in;
```

Description

The **svc_getargs** macro decodes the encoded arguments of an RPC request associated with the RPC service transport handle. The arguments can then be passed to the service procedure for further processing.

Parameters

Item	Description
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

Item	Description
TRUE	successful
FALSE	unsuccessful

Examples

```
#include <stdlib.h>
#include <rpc/rpc.h>

#define PROG 0x3fffffffL
#define VERS 0x1L

static void sample_dispatch();

main()
{
```

```
  char *nettype = "tcp";
  int no_of_handles;

  /* Create RPC service handle and register with RPCBIND service */

  if((no_of_handles = svc_create(sample_dispatch, PROG, VERS,nettype)) == 0)
  {
    fprintf(stdout,"Error in svc_create!");
    exit(EXIT_FAILURE);
  }

  svc_run();
  return 0;
}

/* following is the sample dispatch routine*/
static void sample_dispatch(struct svc_req *request, SVCXPRT *xprt)
{
  int args,result;

  /* decode argument from client using xdr_int() */
  if (svc_getargs(xprt, (xdrproc_t)xdr_int,(caddr_t)&args) == FALSE)
  {
    svcerr_decode(xprt);
    fprintf(stdout,"Error in svc_getargs!");
    return;
  }
  /* call service procedure */

  /* free allocated  data */
  if (svc_freeargs(xprt, (xdrproc_t)xdr_int,(caddr_t)&args) == FALSE)
  {
    fprintf(stdout,"Error in svc_freeargs!");
    return;
  }

  /* send reply to client */
  if(!svc_sendreply(xprt,(xdrproc_t)xdr_int,(caddr_t)&result))
  {
    fprintf(stdout,"Error in svc_sendreply!");
    svcerr_systemerr(xprt);
  }
}
```

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

Item	Description
xprt	Points to the RPC service transport handle.

Related reference:

"svc_register Subroutine" on page 370

"svc_run Subroutine" on page 373

Related information:

List of RPC Programming References

Remote Procedure Call (RPC) Overview for Programming

svc_getreq_common Subroutine
Purpose

Handles remote procedure call (RPC) service requests on a given file descriptor.

Library

Network Services Library (**libnsl.a**)

Syntax

```
#include <rpc/rpc.h>

void svc_getreq_common( fd )
int fd;
```

Description

The **svc_getreq_common** subroutine is used to handle the RPC service request on a specified file descriptor. You can use the subroutine after calling the **poll** or **select** subroutine. The **svc_getreq_common** subroutine is generally used when a server wants to handle RPC service requests asynchronously (the **svc_run** subroutine is not used).

Parameters

Item	Description
fd	Specifies a file descriptor on which the RPC service request arrives.

Examples

```
#include <stdlib.h>
#include <rpc/rpc.h>
#include <poll.h>

void my_svc_getreq_poll(struct pollfd * poll_fd, int retval)
{
    int  i;
    int   ds;

    for (i = fds = 0; fds < retval; i++) {

        /* for all file descriptors check if input is pending
```

```
        and handle the request on that file descriptor */

        svc_getreq_common(pll_fd[i]);
    }
}
main()
{

    int no_of_fds;
    int i;
    struct pollfd pollfd_set[1024];

    /* Register RPC Service */

    /* serve client's requests asynchronously */
    while(1)
    {
        /* initialize the pollfd_set array and
        get no of file descriptors in "no_of_fds"*/

        /* Keep polling on file descriptors */
        switch (i = poll(pollfd_set, no_of_fds, -1))
        {
             case -1:
             case  0:
             continue;
             default:
             /* Handle RPC request on each file descriptor */
             my_svc_getreq_poll(pollfd_set, i);
        }
    }
}
```

Related information:

Transport Independent Remote Procedure Call, IPv6 concerns for Transport Independent Remote Procedure Call

eXternal Data Representation Overview for Programming

svc_getreq_poll Subroutine

Purpose

Handles remote procedure call (RPC) requests on an RPC file descriptor asynchronously.

Library

Network Services Library (**libnsl.a**)

Syntax

```
#include <rpc/rpc.h>

void  svc_getreq_poll(poll_fd, retval)
struct pollfd *poll_fd;
int retval;
```

Description

This subroutine handles RPC requests on an RPC file descriptor asynchronously (when the **svc_run** subroutine is not used). Call the subroutine after calling the **poll** subroutine, which determines that RPC service requests have arrived in RPC file descriptors.

Parameters

Item	Description
poll_fd	Represents the array of the **pollfd** structures on which the polling operation is done.
retval	Represents the value returned by the **poll** subroutine.

Examples

```
#include <stdlib.h>
#include <rpc/rpc.h>
#include <poll.h>

main()
{
    int no_of_fds;
    int i;
    struct pollfd pollfd_set[1024];

    /* Register RPC Service */

    /* serve client's requests asynchronously */
    while(1)
    {
        /* initialize the pollfd_set array and
         get no of file descriptors in "no_of_fds"*/

        /* Keep polling on file descriptors */
        switch (i = poll(pollfd_set, no_of_fds, -1))
        {
            case -1:
            case  0:
            continue;
            default:
            /* Handle RPC request on each file descriptor */
            svc_getreq_poll(pollfd_set, i);
        }
    }
}
```

Related information:

Transport Independent Remote Procedure Call, IPv6 concerns for Transport Independent Remote Procedure Call

eXternal Data Representation Overview for Programming

svc_getreqset Subroutine

Important: The subroutine is exported from both the **libc** and the **libnsl** libraries.

svc_getreqset Subroutine Exported from the libc Library

Purpose

Services a Remote Procedure Call (RPC) request.

Library

C Library (**libc.a**)

Syntax

```
#include <sys/types.h>
#include <sys/select.h>
#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 an 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

Item	Description
rdfds	Specifies the resultant read-file descriptor bit mask.

Restrictions

The maximum number of open file descriptors that an RPC server can use has been set to 32767 so that compatibility can be maintained with RPC-server applications built on earlier releases of AIX.

The **fd_set** type passed into the **svc_getreqset** subroutine must be compiled with **FD_SETSIZE** set to 32767 or larger. Passing in a smaller **fd_set** argument can cause the **svc_getreqset** subroutine to overrun the passed-in buffer.

svc_getreqset Subroutine Exported from the libnsl Library

Purpose

Services a Remote Procedure Call (RPC) request.

Library

Network Services Library (**libnsl.a**)

Syntax

```
#include <rpc/rpc.h>

void svc_getreqset ( rdfds)
fd_set *rdfds;
```

Description

The **svc_getreqset** subroutine is used only when RPC service requests are handled asynchronously (the **svc_run** subroutine is not used). The subroutine is called after a call to the **select** subroutine that determines that an RPC request has arrived on RPC file descriptors. The **svc_getreqset** subroutine returns when all file descriptors specified by the *rdfds* parameter have been serviced.

Parameters

Item	Description
rdfds	Specifies the resultant read-file descriptor bit mask.

Examples

```
#include <stdlib.h>
#include <rpc/rpc.h>
#include <sys/types.h>
#include <sys/select.h>

main()
{
  fd_set rfds;

  /* Register RPC Service */

  /* serve client's requests asynchronously */
  while(1)
  {
  rfds = svc_fdset;

  /* get max value that newly created file descriptor can have in "tb_size" */

    switch (select(tb_size, &rfds, NULL,NULL,NULL))
    {
      case -1:
      case 0 :
         break;
      default:
         /* Handle RPC request on each file descriptor */
         svc_getreqset(&rfds);
    }
  }
}
```

In the example, the **svc_run** subroutine is replaced by a while loop that handles RPC requests asynchronously.

svc_getrpccaller Subroutine

Purpose

Gives the network address of a caller of a procedure (a server-side subroutine).

Library

Network Services Library (**libnsl.a**)

Syntax

```
#include <rpc/rpc.h>
struct netbuf *svc_getrpccaller( xprt );
SVCXPRT *xprt;
```

Description

The **svc_getrpccaller** subroutine gives network address of a caller of a procedure associated with the remote procedure call (RPC) service handle.

Parameters

Item	Description
xprt	Represents the RPC service handle.

Return Values

On successful completion, the **svc_getrpccaller** subroutine returns network address of a caller.

Examples

```
#include <stdlib.h>
#include <rpc/rpc.h>

#define PROG 0x3fffffffL
#define VERS 0x1L

static void sample_dispatch();
main()
{
    char *nettype;
    int no_of_handles;

    nettype = "tcp";

    /* Create RPC service handle and register with RPCBIND service */
    if((no_of_handles = svc_create(sample_dispatch, PROG, VERS,nettype)) == 0)
    {
        fprintf(stdout,"Error in svc_create!");
        exit(EXIT_FAILURE);
    }
    svc_run();
    return 0;
}

/* following is the sample dispatch routine*/
static void sample_dispatch(struct svc_req *request, SVCXPRT *xprt)
{
    int result;
    struct netbuf *clntaddr;

    /* Get client's name and address */
    clntaddr = svc_getrpccaller(xprt);

    /* send reply back to client */
}
```

Related information:

Transport Independent Remote Procedure Call, IPv6 concerns for Transport Independent Remote Procedure Call

eXternal Data Representation Overview for Programming

svc_max_pollfd Global Variable
Purpose

Indicates the maximum length of the **svc_pollfd** array.

Library

Network Services Library (**libnsl.a**)

Syntax

#include <rpc/rpc.h>

pollfd_t ***svc_pollfd**;

Description

The **svc_max_pollfd** global variable is a read-only global variable indicating the maximum length of the **svc_pollfd** array. This variable is generally used when the **svc_run** subroutine is not used and a server wants to handle remote procedure call (RPC) service requests asynchronously. Its value might change when the **svc_getreq_poll** subroutine is called or when other RPC service handle-creation subroutines are called.

Related information:

Transport Independent Remote Procedure Call

eXternal Data Representation Overview for Programming

svc_pollfd Global Variable
Purpose

Points to an array of the **pollfd_t** structures representing the read-file descriptor of a remote procedure call (RPC) server.

Library

Network Services Library (**libnsl.a**)

Syntax

```
#include <rpc/rpc.h>
pollfd_t *svc_pollfd
```

Description

The **svc_pollfd** global variable is a read-only global variable pointing to an array of the **pollfd_t** structures representing the read-file descriptor of an RPC server. Use this variable only when the **svc_run** subroutine is not used and the server wants to handle RPC service requests asynchronously. The value of the variable might change when the **svc_getreq_poll** subroutine is called or when other RPC service handle-creation subroutines are called. By default, the **svc_pollfd** global variable contains 1024 entries. You can change the value using the **rpc_control** subroutine.

Note: Do not pass the address of this variable to any of the select subroutines. Pass a copy of the address instead.

Related information:

Transport Independent Remote Procedure Call

eXternal Data Representation Overview for Programming

svc_raw_create Subroutine
Purpose

Creates a remote procedure call (RPC) handle for raw interfaces.

Library

Network Services Library (**libnsl.a**)

Syntax

```
#include <rpc/rpc.h>
SVCXPRT *svc_raw_create (void)
```

Description

The **svc_raw_create** subroutine is a bottom-level API for transport-independent remote procedure calls (TI_PRC). Bottom-level APIs provide a full control over the transport options. This subroutine creates an RPC service handle for raw interfaces. The transport type is a buffer in the address space of the process. Thus the client must be in the same address space. This subroutine provides simulation of the RPC service with all RPC overheads without actually using any network interface. This subroutine does not register a server with an RPC service package because the program number and version number are not specified.

Note: Do not use the **svc_run** subroutine over raw interfaces.

Return Values

Item	Description
an PRC service handle	successful
NULL	unsuccessful

Examples

```
#include <stdlib.h>
#include <rpc/rpc.h>
#include <netconfig.h>

main()
{
    SVCXPRT *svc_handle;      /* server handle */

    /* create server handle using svc_raw_create */
    if((svc_handle=svc_raw_create())==(SVCXPRT *)NULL)
    {
          fprintf(stdout,"Error in svc_raw_create!");
          exit(EXIT_FAILURE);
    }

    /* Note that transport type passed to svc_reg() is NULL */
    if(svc_reg(svc_handle, PROG, VERS, sample_dispatch, NULL) == 0)
    {
          fprintf(stdout,"Error in svc_reg!");
          exit(EXIT_FAILURE);
    }

    /* note that here svc_run() is not called */
    /* To call registered procedure, client should be in same address space */

    return 0;
}

/* following is the sample dispatch routine*/
static void sample_dispatch(struct svc_req *request, SVCXPRT *xprt)
{
       /*   dispatch routine code  */
}
```

Related information:

Transport Independent Remote Procedure Call, IPv6 concerns for Transport Independent Remote Procedure Call

eXternal Data Representation Overview for Programming

svc_reg Subroutine
Purpose

Associates the program number and version number with a service-dispatch subroutine.

Library

Network Services Library (**libnsl.a**)

Syntax

```
#include <rpc/rpc.h>

int svc_reg( xprt, prog, vers, dispatch, nconf)
const SVCXPRT *xprt;
const rpcprog_t prog;
const rpcvers_t vers;
void (*dispatch)(struct svc_req *, SVCXPRT *);
const struct netconfig *nconf;
```

Description

The **svc_reg** subroutine is an expert-level API for transport-independent remote procedure calls (TI_PRC). This subroutine registers the program number and version with the RPC service package on the transport that is specified by the *nconf* parameter. This subroutine associates the program number and version number with a service-dispatch subroutine. If you set the *nconf* parameter to a null value, the service is not registered with the **rpcbind** service. Otherwise, the service is registered with the **rpcbind** service.

Parameters

Item	Description
xprt	Specifies an RPC service handle.
prog	Specifies the program number.
vers	Specifies the version number.
dispatch	Specifies the subroutine that is called when a service request arrives.
nconf	Defines a **netconfig** structure that specifies the type of transport.

Return Values

Item	Description
1	successful
0	unsuccessful

Examples

In the following example, before the **svc_reg** subroutine is called, an RPC service handle must be created with the **svc_dg_create**, **svc_vc_create**, **svc_fd_create** subroutines and so on.

```
#include <stdlib.h>
#include <rpc/rpc.h>
#include <netconfig.h>

#define PROG 0x3fffffffL
#define VERS 0x1L

static void sample_dispatch();

main()
{

  SVCXPRT *svc_handle;
```

```
  struct netconfig *nconf;

  /* Get transport type */
  nconf = getnetconfigent("tcp");
  if(nconf == NULL)
  {
    fprintf(stderr,"\nError in getnetconfigent!\n");
    exit(EXIT_FAILURE);
  }

  /* Create RPC service handle */
  svc_handle = svc_tli_create(RPC_ANYFD, nconf, NULL, 0, 0);
  if(svc_handle == (SVCXPRT *)NULL)
  {
    fprintf(stdout,"Error in svc_tli_create!");
    exit(EXIT_FAILURE);
  }


  /* Register dispatch routine for prog and vers with RPCBIND service */

  if(svc_reg(svc_handle, PROG, VERS, sample_dispatch,nconf) == 0)
  {
    fprintf(stdout,"Error in svc_reg!");
    exit(EXIT_FAILURE);
  }

  svc_run();
  return 0;
}

/* following is the sample dispatch routine*/
static void sample_dispatch(struct svc_req *request, SVCXPRT *xprt)
{

  /*   dispatch routine code  */

}
```

Related information:

Transport Independent Remote Procedure Call, IPv6 concerns for Transport Independent Remote Procedure Call

eXternal Data Representation Overview for Programming

svc_register Subroutine

Important:

- The subroutine is exported from both the **libc** and the **libnsl** libraries.

svc_register Subroutine Exported from the libc Library

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

Item	Description
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.

svc_register Subroutine Exported from the libnsl Library

Purpose

Associates the program and version number with the dispatch subroutine.

Library

Network Services Library (**libnsl.a**)

Syntax

```
#include <rpc/rpc.h>

bool_t svc_register (xprt, prognum, versnum, dispatch, protocol)
SVCXPRT *xprt;
rpcprog_t prognum;
rpcvers_t versnum;
void (* dispatch)();
int protocol;
```

Description

The **svc_register** subroutine maps a program and version to a service dispatch procedure pointed to by the *dispatch* parameter. If the value of the *protocol* parameter is 0, the service is not registered with the

portmap daemon. If the value of the *protocol* parameter is not 0 (or it is **IPPROTO_UDP** or **IPPROTO_TCP**), the remote procedure triplet (the program, the version, and the protocol) is mapped to the **xprt->xp_port** port.

The **svc_register** subroutine is obsolete. Use the **svc_reg** subroutine instead.

The dispatch procedure has the following form:

```
dispatch (request, xprt)
struct svc_req *request;
SVCXPRT *xprt;
```

Parameters

Item	Description
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

Item	Description
1	successful
0	unsuccessful

Examples

```
#include <rpc/rpc.h>

static void dispatch(rqstp, transp) /* remote procedure */
struct svc_req *rqstp;
SVCXPRT *transp;
{
  /* Dispatch Routine Code */
}

int main()
{
  SVCXPRT *svc = NULL;
  uint_t sendsz, recvsz;
  int protocol = IPPROTO_TCP;

  /* Set send and recieve buffer sizes to 0 so that they are set to default values
   * when svctcp_create() is called
   */
  sendsz = 0;
  recvsz = 0;

  /* Create service handle for tcp transport */
  svc = (SVCXPRT *) svctcp_create(RPC_ANYSOCK, sendsz, recvsz);
  if (svc == NULL) {
    fprintf(stderr, "\nsvcudp_create failed\n");
    exit(1);
  }

  if(svc_register(svc, prognum, versnum, dispatch, protocol)==0);
  {
    fprintf(stderr,"svc_register() failed");
    exit(1);
  }
```

```
  /* Accept the client requests */
  svc_run();

  return 0;
}
```

Related reference:
"svc_getcaller Macro" on page 360

svc_run Subroutine

Important: The subroutine is exported from both the **libc** and the **libnsl** libraries.

svc_run Subroutine Exported from the libc Library

Purpose

Waits for a Remote Procedure Call service request to arrive.

Library

C Library (**libc.a**)

Syntax

```
#include <rpc/rpc.h>
void svc_run (void);
```

Description

The **svc_run** subroutine waits for a Remote Procedure Call (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.

Restrictions

The maximum number of open file descriptors that an RPC server can use has been set to 32767 so that compatibility can be maintained with RPC-server applications built on earlier releases of AIX.

svc_run Subroutine Exported from the libnsl Library

Purpose

Waits for a Remote Procedure Call service request to arrive.

Library

Network Services Library (**libnsl.a**)

Syntax

```
#include <rpc/rpc.h>
void svc_run (void);
```

Description

The **svc_run** subroutine waits for a remote procedure call (RPC) service request to arrive and never returns. When a server is configured in single-threaded mode and if a request arrives, the **svc_run** subroutine calls the appropriate dispatch subroutine. In the Automatic-MT or User-MT mode, this

subroutine must be called exactly once. In the Automatic-MT mode, a new thread is created for each new RPC request. In the User-MT mode, the **svc_run** subroutine does not create threads, and you must create threads to serve RPC requests.

Examples

```
#include <rpc/rpc.h>
#include <stdlib.h>

#define PROG 0x3fffffffL
#define VERS 0x1L

static void sample_dispatch();

main()
{
  char *nettype;
  int no_of_handles;
  nettype = "tcp";
 /* Create RPC service handle and register with RPCBIND service */
  if((no_of_handles = svc_create(sample_dispatch, PROG, VERS, nettype)) == 0)
  {
    fprintf(stdout,"Error in svc_create!");
    exit(EXIT_FAILURE);
  }
  svc_run();
  return 0;
}

/* following is the sample dispatch routine*/
static void sample_dispatch(struct svc_req *request, SVCXPRT *xprt)
{
  /* dispatch routine code  */
}
```

Related reference:

"svc_getcaller Macro" on page 360

svc_sendreply Subroutine

Important: The subroutine is exported from both the **libc** and the **libnsl** libraries.

svc_sendreply Subroutine Exported from the libc Library

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

Item	Description
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.

svc_sendreply Subroutine Exported from the libnsl Library

Purpose

Sends back the results of a remote procedure call.

Library

Network Services Library (**libnsl.a**)

Syntax

```
#include <rpc/rpc.h>

bool_t svc_sendreply(xprt, oproc, output)
const SVCXPRT *xprt;
const xdrproc_t oproc;
caddr_t output;
```

Description

The **svc_sendreply** subroutine sends back the results of a remote procedure call (RPC). This subroutine is called by an RPC service dispatch subroutine. This subroutine encodes the result from a service procedure into the eXternal Data Representation (XDR) format with a specified XDR procedure, and then sends the procedure back to a client.

Parameters

Item	Description
xprt	Points to the RPC service transport handle of the caller.
oproc	Specifies the eXternal Data Representation (XDR) routine that encodes the results.
output	Points to the address where results are placed.

Return Values

Item	Description
TRUE	successful
FALSE	unsuccessful

Examples

```
#include <rpc/rpc.h>
#include <stdlib.h>
#define PROG 0x3fffffffL
#define VERS 0x1L

static void sample_dispatch();
main()
{
  char *nettype;
  int no_of_handles;

  nettype = "tcp";
  /* Create RPC service handle and register with RPCBIND service */
  if((no_of_handles = svc_create(sample_dispatch, PROG, VERS,nettype)) == 0)
  {
    fprintf(stdout, " Error in svc_create ");
    exit(EXIT_FAILURE);
  }
  svc_run();
  return 0;
}

/* following is the sample dispatch routine*/
static void sample_dispatch(struct svc_req *request, SVCXPRT *xprt)
{
  int result;

  /*  Some code to call appropriate service procedure.
   * Procedure will return its result.
  */
  /* Send the result back to client */
  if(!svc_sendreply(xprt,(xdrproc_t)xdr_int,(caddr_t)&result))
  {
    fprintf(stdout, " Error in svc_create ");
    svcerr_systemerr(transp);
  }
}
```

svc_tli_create Subroutine
Purpose

Creates a remote procedure call (RPC) service handle for the specified transport.

Library

Network Services Library (**libnsl.a**)

Syntax

```
#include <rpc/rpc.h>

SVCXPRT *svc_tli_create(fd, nconf, bind_addr, sendsize, recvsize)
int fd;
const struct netconfig *nconf;
const struct  t_bind *bind_addr;
const uint_t sendsize;
const uint_t recvsize ;
```

Description

The subroutine is an expert-level API for transport-independent remote procedure calls (TI_PRC). This subroutine creates an RPC service handle on a given file descriptor and returns a pointer to it. The server is not registered with an RPC service package because the program and version numbers are not specified. If you specify the *fd* parameter with the **RPC_ANYFD** value, the file descriptor on the specified transport is opened. If the file descriptor is open but unbound, and you specify the *bind_addr* parameter with a valid address, the file descriptor is bound by the given address. If the file descriptor is open and unbound, and the value of the *bind_addr* parameter is NULL, the default address for the transport is used and the number of connections for the connection-oriented transport is set to 8.

Parameters

Item	Description
fd	Indicates an open file descriptor that is bound or unbound.
nconf	Indicates type of transport.
bind_addr	Indicates address to which the file descriptor is to be bound. The value can be a valid address or NULL.
sendsize	Specify the send buffer size. If the value is set to 0, the default size for is used.
recvsize	Specify the receive buffer size. If the value is set to 0, the default size is used.

Return Values

Item	Description
a service handle	successful
NULL	unsuccessful

Examples

```
#include <stdlib.h>
#include <rpc/rpc.h>
#include <netconfig.h>

main()
{
      SVCXPRT *svc_handle;       /* server handle */
      struct netconfig *nconf;

      /* get proper file descriptor */

      /* get transport type */
      nconf = getnetconfigent("tcp");
      if (nconf == (struct netconfig *) NULL)
      {
            fprintf(stderr, "getnetconfigent failed\n");
            exit(EXIT_FAILURE);
      }

      /* sendsize and recvsize are 0, thus default size will be chosen */
      if((svc_handle=svc_tli_create(fd, nconf, 0, 0, 0))==(SVCXPRT *)NULL)
      {
            fprintf(stdout,"Error in svc_tli_create!");
            exit(EXIT_FAILURE);
       }

      /* Register RPC service using RPC service handle with RPCBIND package */
      svc_run();
      return 0;
}
```

Related information:

Transport Independent Remote Procedure Call, IPv6 concerns for Transport Independent Remote Procedure Call

eXternal Data Representation Overview for Programming

svc_tp_create Subroutine

Purpose

Creates a server handle for the specified transport.

Library

Network Services Library (**libnsl.a**)

Syntax

```
#include <rpc/rpc.h>

SVCXPRT *svc_tp_create(dispatch, prog, vers, nconf);
void (*dispatch)(struct svc_req*, SVCXPRT*);
const rpcprog_t prog;
const rpcvers_t vers;
const struct netconfig *nconf;
```

Description

The subroutine is an intermediate-level API for transport-independent remote procedure calls (TI_PRC). This subroutine creates and returns a service handle for the transport specified by *nconf* parameter. The subroutine also registers a server with the RPCBIND service. When a request arrives for the specified program and version, a subroutine specified by the *dispatch* parameter is called. Call the **svc_run** subroutine so that the server can listen to the requests from clients.

Parameters

Item	Description
dispatch	Specifies the subroutine that is called when a service request arrives.
prog	Specifies the program number.
vers	Specifies the version number.
nconf	Indicates a type of transport.

Return Values

Item	Description
a service handle	successful
NULL	unsuccessful

Examples

```
#include <stdlib.h>
#include <rpc/rpc.h>
#include <netconfig.h>

#define PROG 0x3fffffffL
#define VERS 0x1L

static void sample_dispatch();

main()
{
    SVCXPRT *svc_handle; /* server handle */
    struct netconfig *nconf; /* transport type */
```

```
    /* get transport type */
    nconf = getnetconfigent("tcp");
    if (nconf == (struct netconfig *) NULL)
    {
         fprintf(stderr, "getnetconfigent failed.\n");
         exit(EXIT_FAILURE);
    }

    /* create service handle and register with RPCBIND service */

    if((svc_handle=svc_tp_create(sample_dispatch, PROG, VERS, nconf))==(SVCXPRT *)NULL)
    {
         fprintf(stdout,"Error in svc_tp_create!");
         exit(EXIT_FAILURE);
    }

    svc_run();
    return 0;
}

/* following is the sample dispatch routine*/
static void sample_dispatch(struct svc_req *request, SVCXPRT *xprt)
{

     /* code for dispatch routine  */

}
```

Related information:

Transport Independent Remote Procedure Call, IPv6 concerns for Transport Independent Remote Procedure Call

eXternal Data Representation Overview for Programming

svc_unreg Subroutine

Purpose

Unregisters the program number and version number with the remote procedure call (RPC) service package.

Library

Network Services Library (**libnsl.a**)

Syntax

```
#include <rpc/rpc.h>

void  svc_unreg(prog ,vers)
const rpcprog_t prog;
const rpcvers_t vers;
```

Description

This subroutine is an expert-level API for transport-independent remote procedure calls (TI_PRC). This subroutine removes the mapping to network address from the RPC service package. The subroutine also unregisters the program number and version number from the dispatch subroutine with the RPC service package. When the subroutine is called, the whole service that is identified by the program and version gets unregistered.

Parameters

Item	Description
prog	Specifies the program number.
vers	Specifies the version number.

Examples

```
#include <stdlib.h>
#include <rpc/rpc.h>
#include <netconfig.h>

#define PROG 0x3fffffffL
#define VERS 0x1L

main()
{

  SVCXPRT *svc_handle;    /* service handle */
  struct netconfig *nconf;

  /* Get transport type and create RPC service handle. */

  /* Register dispatch routine for prog and vers with RPCBIND service */

  If(svc_reg(svc_handle, PROG, VERS, sample_dispatch,nconf) == 0)
  {
    sprintf(stdout,"Error in svc_reg!");
    exit(EXIT_FAILURE);
  }

  /* Unregister the service with given prog and vers */
  svc_unreg(PROG, VERS);

  return 0;
}
```

Related information:

Transport Independent Remote Procedure Call, IPv6 concerns for Transport Independent Remote Procedure Call

eXternal Data Representation Overview for Programming

svc_unregister Subroutine

Important:

- The subroutine is exported from both the **libc** and the **libnsl** libraries.

svc_unregister Subroutine Exported from the libc Library

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

Item	Description
prognum	Specifies the program number of the remote program.
versnum	Specifies the version number of the remote program.

svc_unregister Subroutine Exported from the libnsl Library

Purpose

Removes mappings between a service procedure and dispatch subroutines.

Library

Network Services Library (**libnsl.a**)

Syntax

```
#include <rpc/rpc.h>

void svc_unregister (prognum, versnum)
rpcprog_t prognum;
rpcvers_t 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 triple (the program, the version, the protocol) from the **portmap** daemon.

The **svc_unregister** subroutine is obsolete. Use the **svc_unreg** subroutine instead.

Parameters

Item	Description
prognum	Specifies the program number of the remote program.
versnum	Specifies the version number of the remote program.

Examples

```
#include <rpc/rpc.h>

rpcprog_t prognum = 0x3fffffffL;
rpcvers_t versnum = 0x1L;

static void dispatch(rqstp, transp) /* remote procedure */
struct svc_req *rqstp;
SVCXPRT *transp;
{
  /* Dispatch Routine Code */
}
```

```
void main_exit_handler()
{
  svc_unregister(prognum,versnum);
}

int main()
{
  SVCXPRT   *svc = NULL;
  uint_t     sendsz, recvsz;
  int  protocol = IPPROTO_TCP;

  /* register exit handler which on exit of server program calls svc_unregister */
  atexit(main_exit_handler)

  /* Set send and recieve buffer sizes to 0 so that they are set to default values
   * when svctcp_create() is called
   */
  sendsz = 0;
  recvsz = 0;

  /* Create service handle for tcp transport */
  svc = (SVCXPRT *) svctcp_create(RPC_ANYSOCK, sendsz, recvsz);
  if (svc == NULL) {
    fprintf(stderr, "\nsvcudp_create failed\n");
    exit(1);
  }

  if(svc_register(svc, prognum, versnum, dispatch, protocol)==0);
  {
    fprintf(stderr,"svc_register() failed");
    exit(1);
  }

  /* Accept client requests */
  svc_run();

  return 0;
}
```

svc_vc_create Subroutine
Purpose

Creates a remote procedure call (RPC) service handle for connection-oriented transport.

Library

Network Services Library (**libnsl.a**)

Syntax

```
#include <rpc/rpc.h>

SVCXPRT *svc_vc_create(fd, sendsize, recvsize)
int fd;
const uint_t sendsize;
const uint_t recvsize;
```

Description

The **svc_vc_create** subroutine is a bottom-level API for transport-independent remote procedure calls (TI_PRC). Bottom-level APIs provide a full control over the transport options. This subroutine creates an RPC service handle for connection-oriented transport. This subroutine does not register a server with an RPC service package because the program number and version number are not specified.

Parameters

Item	Description
fd	Indicates an open file descriptor that is bound.
sendsize	Specify the send buffer size. If the value is set to 0, the default size for that transport is used.
recvsize	Specify the receive buffer size. If the value is set to 0, the default size for that transport is used.

Return Values

Item	Description
an PRC service handle	successful
NULL	unsuccessful

Examples

```
#include <stdlib.h>
#include <rpc/rpc.h>

main()
{
       SVCXPRT *svc_handle;      /* server handle */
       int fd;                   /* file descriptor */

       /* Get proper file descriptor */

       /* sendsize and recvsize are 0, thus default size will be chosen */

       if((svc_handle = svc_vc_create(fd, 0, 0))==(SVCXPRT *)NULL)
       {
                 fprintf(stdout,"Error in svc_vc_create!");
                 exit(EXIT_FAILURE);
       }

       /* Register RPC service */

       svc_run();
       return 0;
}
```

Related information:

Transport Independent Remote Procedure Call, IPv6 concerns for Transport Independent Remote Procedure Call

eXternal Data Representation Overview for Programming

svcerr_auth Subroutine

Important: The subroutine is exported from both the **libcrpc** and the **libnsl** libraries.

svcerr_auth Subroutine Exported from the libcrpc Library

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 (RPC) because of an authentication error. This subroutine sets the status of the RPC reply message to **AUTH_ERROR**.

Parameters

Item	Description
xprt	Points to the RPC service transport handle.
why	Specifies the authentication error.

svcerr_auth Subroutine Exported from the libnsl Library

Purpose

Indicates that the service dispatch routine cannot complete a remote procedure call due to an authentication error.

Library

Network Services Library (**libnsl.a**)

Syntax

```
#include <rpc/rpc.h>

void svcerr_auth ( xprt, why)
const SVCXPRT *xprt;
const enum auth_stat why;
```

Description

The **svcerr_auth** subroutine is called by a service dispatch subroutine when an authentication error occurs. This subroutine sets the status of the remote procedure call (RPC) reply message to **RPC_AUTHERROR**.

Parameters

Item	Description
xprt	Points to the RPC service transport handle.
why	Specifies the authentication error.

Examples

In the following example, a dispatch subroutine sends reply back to the client with the reason indicating why an authentication error occurred.

```
#include <rpc/rpc.h>
#include <stdlib.h>
#define PROG 0x3fffffffL
#define VERS 0x1L
```

```
static void sample_dispatch();
main()
{
  char *nettype;
  int no_of_handles;

  nettype = "tcp";
  /* Create RPC service handle and register with RPCBIND service */
  if((no_of_handles = svc_create(sample_dispatch, PROG, VERS,nettype)) == 0)
  {
    fprintf(stdout, " Error in svc_create ");
    exit(EXIT_FAILURE);
  }
  svc_run();
  return 0;
}

/* following is the sample dispatch routine*/
static void sample_dispatch(struct svc_req *request, SVCXPRT *xprt)
{
  int result;
  enum auth_stat why;

  /* Check for appropriate authentication */
  /* set reason for authentication error */
  why = AUTH_BADCRED;

  /* Send reply to client */
  svcerr_auth(xprt,why);
}
```

svcerr_decode Subroutine

Important: The subroutine is exported from both the **libc** and the **libnsl** libraries.

svcerr_decode Subroutine Exported from the libc Library

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 Remote Procedure Call (RPC) reply message to the **GARBAGE_ARGS** condition.

Parameters

Item	Description
xprt	Points to the RPC service transport handle.

svcerr_decode Subroutine Exported from the libnsl Library

Purpose

Indicates that the service dispatch routine cannot decode the parameters of a request.

Library

Network Services Library (**libnsl.a**)

Syntax

```
#include <rpc/rpc.h>
void svcerr_decode ( xprt)
const 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 remote procedure call (RPC) reply message to the **RPC_CANTDECODEARGS** condition.

Parameters

Item	Description
xprt	Points to the RPC service transport handle.

Examples

```
#include <rpc/rpc.h>
#include <stdlib.h>
#define PROG 0x3fffffffL
#define VERS 0x1L

static void sample_dispatch();
main()
{
  char *nettype = "tcp";
  int no_of_handles;

  /* Create RPC service handle and register with RPCBIND service */
  if((no_of_handles = svc_create(sample_dispatch, PROG, VERS,nettype)) == 0)
  {
    fprintf(stdout, " Error in svc_create ");
    exit(EXIT_FAILURE);
  }
  svc_run();
  return 0;
}

/* following is the sample dispatch routine*/
static void sample_dispatch(struct svc_req *request, SVCXPRT *xprt)
{
  int args;

  /* decode argument from client using xdr_int() */
  if (svc_getargs(xprt, (xdrproc_t)xdr_int,(caddr_t)&args) == FALSE)
  {
```

```
 svcerr_decode(xprt);
 fprintf(stdout, " Error in svc_create ");
 return;
  }
}
```

svcerr_noproc Subroutine

Important: The subroutine is exported from both the **libc** and the **libnsl** libraries.

svcerr_noproc Subroutine Exported from the libc Library

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 Remote Procedure Call (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.

Parameters

Item	Description
xprt	Points to the RPC service transport handle.

svcerr_noproc Subroutine Exported from the libnsl Library

Purpose

Indicates that the service dispatch routine cannot complete a remote procedure call because the program cannot support the requested procedure.

Library

Network Services Library (**libnsl.a**)

Syntax

```
#include <rpc/rpc.h>

void svcerr_noproc ( xprt)
const SVCXPRT *xprt;
```

Description

The **svcerr_noproc** subroutine is called by a service dispatch subroutine when the procedure number that is requested by a caller is not implemented. This subroutine sets the status of the remote procedure call (RPC) reply message to the **RPC_PROCUNAVAIL** condition, which indicates that the program cannot support the requested procedure.

Parameters

Item	Description
xprt	Points to the RPC service transport handle.

Examples

In the following example, the **svcerr_noproc** subroutine is called from a dispatch subroutine when a requested procedure is not supported.

```
#include <rpc/rpc.h>
#include <stdlib.h>
#define PROG 0x3fffffffL
#define VERS 0x1L

static void sample_dispatch();
main()
{
  char *nettype = "tcp";
 int no_of_handles;

  /* Create RPC service handle and register with RPCBIND service */
  if((no_of_handles = svc_create(sample_dispatch, PROG, VERS,nettype)) == 0)
  {
    fprintf(stdout, " Error in svc_create ");
    exit(EXIT_FAILURE);
  }
  svc_run();
  return 0;
}
/* following is the sample dispatch routine*/
static void sample_dispatch(struct svc_req *request, SVCXPRT *xprt)
{
int args;

switch(request->rq_proc)
{
  case 0:
  svc_sendreply(xprt, (xdrproc_t) xdr_void, (caddr_t) NULL);
  return;
  case 1:
  case 2:
  . . .
  case n:
  /* Call appropriate procedure */
  default:
  svcerr_noproc(xprt);
  return;
  }
}
```

svcerr_noprog Subroutine

Important: The subroutine is exported from both the **libc** and the **libnsl** libraries.

svcerr_noprog Subroutine Exported from the libc Library

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.

Parameters

Item	Description
xprt	Points to the RPC service transport handle.

svcerr_noprog Subroutine Exported from the libnsl Library

Purpose

Indicates that the service dispatch routine cannot complete a remote procedure call because the requested program is not registered.

Library

Network Services Library (**libnsl.a**)

Syntax

```
#include <rpc/rpc.h>
void svcerr_noprog ( xprt)
const 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 **RPC_PROGUNAVAIL** condition, which indicates that the remote server has not exported the program.

Parameters

Item	Description
xprt	Points to the RPC service transport handle.

Examples

```
#include <rpc/rpc.h>
#include <stdlib.h>
#define PROG 0x3fffffffL
#define VERS 0x1L

static void sample_dispatch();
main()
{
  char *nettype = "tcp";
  int no_of_handles;

  /* Create RPC service handle and register with RPCBIND service */
  if((no_of_handles = svc_create(sample_dispatch, PROG, VERS,nettype)) == 0)
  {
    fprintf(stdout, " Error in svc_create ");
    exit(EXIT_FAILURE);
  }
  svc_run();
  return 0;
}

/* following is the sample dispatch routine*/
static void sample_dispatch(struct svc_req *request, SVCXPRT *xprt)
{
  int args;
  /* Dispatch routine code */
  /* If requested program is not registered. */
  svcerr_noprog(xprt);
}
```

svcerr_progvers Subroutine

Important: The subroutine is exported from both the **libc** and the **libnsl** libraries.

svcerr_progvers Subroutine Exported from the libc Library

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.

Parameters

Item	Description
xprt	Points to the RPC service transport handle.

svcerr_progvers Subroutine Exported from the libnsl Library

Purpose

Indicates that the service dispatch routine cannot complete the remote procedure call because the requested program version is not registered.

Library

Network Services Library (**libnsl.a**)

Syntax

```
#include <rpc/rpc.h>

void svcerr_progvers( xprt, low, high)
const SVCXPRT *xprt;
const rpcvers_t low;
const rpcvers_t high;
```

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 **RPC_PROGVERSMISMATCH** condition, which indicates that the remote server cannot support the client's version number.

Note: Service implementors do not usually need this subroutine.

Parameters

Item	Description
xprt	Points to the RPC service transport handle.
low	Indicates the lowest version number.
high	Indicates the highest version number.

Examples

```
#include <rpc/rpc.h>
#include <stdlib.h>

#define PROG 0x3fffffffL
#define VERS 0x1L

static void sample_dispatch();
main()
{
  char *nettype = "tcp";
```

```
  int no_of_handles;

  /* Create RPC service handle and register with RPCBIND service */
  if((no_of_handles = svc_create(sample_dispatch, PROG, VERS,nettype)) == 0)
  {
    fprintf(stdout,"Error in svc_create!");
    exit(EXIT_FAILURE);
  }
  svc_run();
  return 0;
}

/* following is the sample dispatch routine*/
static void sample_dispatch(struct svc_req *request, SVCXPRT *xprt)
{
  int args,high,low;

  /* Dispatch routine code */
  /* If requested version of a program is not registered. */
  svcerr_progvers(xprt,low,high);
}
```

svcerr_systemerr Subroutine

Important: The subroutine is exported from both the **libc** and the **libnsl** libraries.

svcerr_systemerr Subroutine Exported from the libc Library

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 Remote Procedure Call (RPC) reply message to the **SYSTEM_ERR** condition.

Parameters

Item	Description
xprt	Points to the RPC service transport handle.

svcerr_systemerr Subroutine Exported from the libnsl Library

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

Network Services Library (**libnsl.a**)

Syntax

```
#include <rpc/rpc.h>
void svcerr_systemerr ( xprt)
const SVCXPRT *xprt;
```

Description

The **svcerr_systemerr** subroutine is called by a service dispatch subroutine when a system error occurs that can not be covered by any particular protocol. The subroutine sets the status of the remote procedure call (RPC) reply message to the **RPC_SYSTEMERROR** condition.

Parameters

Item	Description
xprt	Points to the RPC service transport handle.

Examples

```
#include <rpc/rpc.h>
#include <stdlib.h>

#define PROG 0x3fffffffL
#define VERS 0x1L

static void sample_dispatch();
main()
{
  char *nettype;
  int no_of_handles;

  nettype = "tcp";
  /* Create RPC service handle and register with RPCBIND service */
  if((no_of_handles = svc_create(sample_dispatch, PROG, VERS,nettype)) == 0)
  {
    fprintf(stdout,"Error in svc_create!");
    exit(EXIT_FAILURE);
  }
  svc_run();
  return 0;
}

/* following is the sample dispatch routine*/
static void sample_dispatch(struct svc_req *request, SVCXPRT *xprt)
{
  int result;
```

```
  /*  Some code to call appropriate service procedure.
    Procedure will return its result.
  */
  if(!svc_sendreply(xprt,(xdrproc_t)xdr_int,(caddr_t)&result))
  {
    fprintf(stdout,"Error in svc_sendreply!");
    svcerr_systemerr(xprt);
  }
}
```

svcerr_weakauth Subroutine

Important: The subroutine is exported from both the **libc** and the **libnsl** libraries.

svcerr_weakauth Subroutine Exported from the libc Library

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 (RPC) because the supplied authentication parameters are insufficient for security reasons.

The **svcerr_weakauth** subroutine calls the **svcerr_auth** subroutine with the correct 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**).

Parameters

Item	Description
xprt	Points to the RPC service transport handle.

svcerr_weakauth Subroutine Exported from the libnsl Library

Purpose

Indicates that the service dispatch routine cannot complete the remote procedure call due to insufficient authentication security parameters.

Library

Network Services Library (**libnsl.a**)

Syntax

```
#include <rpc/rpc.h>
void svcerr_weakauth ( xprt)
const SVCXPRT *xprt;
```

Description

The **svcerr_weakauth** subroutine is called by a service dispatch routine that cannot make the remote procedure call (RPC) because the supplied authentication parameters are insufficient for security reasons.

The **svcerr_weakauth** subroutine calls the **svcerr_auth** subroutine with the correct 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 (**RPC_AUTHERROR**).

Parameters

Item	Description
xprt	Points to the RPC service transport handle.

Examples

```
#include <rpc/rpc.h>
#include <stdlib.h>
#define PROG 0x3fffffffL
#define VERS 0x1L

static void sample_dispatch();
main()
{
  char *nettype;
  int no_of_handles;

  nettype = "tcp";
  /* Create RPC service handle and register with RPCBIND service */
  if((no_of_handles = svc_create(sample_dispatch, PROG, VERS,nettype)) == 0)
  {
    fprintf(stdout,"Error in svc_create!");
    exit(EXIT_FAILURE);
  }
  svc_run();
  return 0;
}

/* following is the sample dispatch routine*/
static void sample_dispatch(struct svc_req *request, SVCXPRT *xprt)
{
  int result;
  enum auth_stat why;

  /* Check for appropriate authentication */
  /* If insufficient authentication security parameters then send reply to client */
  svcerr_weakauth(xprt);
}
```

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).

By default, the RPC server uses nonblocking I/O with TCP. This behavior can be changed by setting the environment variable **RPC_TCP_MODE** to **USEBLOCKING**, which causes the TCP RPC server to use blocking I/O.

Note: Using blocking I/O leaves the server vulnerable to disruption by malicious or misconfigured clients.

Parameters

Item	Description
fd	Identifies the descriptor.
sendsize	Specifies the size of the send buffer.
recvsize	Specifies the size of the receive buffer.

Restrictions

The maximum number of open file descriptors that an RPC server can use has been set to 32767 so that compatibility can be maintained with RPC-server applications built on earlier releases of AIX.

Return Values

Upon successful completion, this subroutine returns a TCP-based transport handle. If unsuccessful, it returns a value of null.

Related information:

List of RPC Programming References

TCP/IP protocols

Remote Procedure Call (RPC) Overview for Programming

Sockets Overview

svcraw_create Subroutine
Purpose

Creates a toy Remote Procedure Call (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 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.

Return Values

Upon successful completion, this subroutine returns a pointer to a valid RPC transport handle. If unsuccessful, it returns a value of null**.**

Related reference:

"clntraw_create Subroutine" on page 266

Related information:

List of RPC Programming References

Remote Procedure Call (RPC) Overview for Programming

svctcp_create Subroutine

Important: The subroutine is exported from both the **libc** and the **libnsl** libraries.

svctcp_create Subroutine Exported from the libc Library

Purpose

Creates a Transmission Control Protocol/Internet Protocol (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 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.

By default, the RPC server uses nonblocking I/O with TCP. This behavior can be changed by setting the environment variable **RPC_TCP_MODE** to **USEBLOCKING**, which causes the TCP RPC server to use blocking I/O.

Note: Using blocking I/O leaves the server vulnerable to disruption by malicious or misconfigured clients.

Parameters

Item	Description
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.

Restrictions

The maximum number of open file descriptors that an RPC server can use has been set to 32767 so that compatibility can be maintained with RPC-server applications built on earlier releases of AIX.

Return Values

Upon successful completion, this subroutine returns a valid RPC service transport handle. If unsuccessful, it returns a value of null.

svctcp_create Subroutine Exported from the libnsl Library

Purpose

Creates a Transmission Control Protocol/Internet Protocol (TCP/IP) service transport handle.

Library

Network Services Library (**libnsl.a**)

Syntax

```
#include <rpc/rpc.h>

SVCXPRT *svctcp_create (fd, sendsz, recvsz)
int fd;
uint_t sendsz;
uint_t recvsz;
```

Description

The **svctcp_create** subroutine creates a Remote Procedure Call (RPC) service transport handle based on TCP/IP and returns a pointer to it.

Because TCP/IP remote procedure calls use buffered I/O, you 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. The *fd* parameter specifies a file descriptor on a TCP transport. You can set the value of the *fd* parameter to **RPC_ANYSOCK**, so that the **svctcp_create** subroutine creates a new file descriptor on the TCP transport and binds the file descriptor to a port.

Use the **svc_create** subroutine instead of the **svctcp_create** subroutine. The **svctcp_create** subroutine is compatible only with earlier versions of AIX.

Parameters

Item	Description
fd	Specifies the file descriptor that is associated with the TCP transport.
sendsz	Specifies the size of the send buffer.
recvsz	Specifies the size of the receive buffer.

Return Values

Item	Description
a valid RPC service transport handle	successful
a null value	unsuccessful

Examples

```
#include <rpc/rpc.h>

static void dispatch(struct svc_req *, SVCXPRT *);

int main()
{
  SVCXPRT    *svc=NULL;
  int     fd;
  uint_t     sendsz, recvsz;
  int     protocol = IPPROTO_TCP;

  /* Set send and recieve buffer sizes to 0 so that they are set to
   * default values when svctcp_create() is called
   */
  sendsz = 0;
  recvsz = 0;

  /* Set the file descriptor to RPC_ANYFD */
  fd = RPC_ANYSOCK;

  svc = (SVCXPRT *) svctcp_create(fd, sendsz, recvsz);
  if(svc==NULL)
  {
    fprintf(stderr,"svctcp_create() failed");
    exit(1);
  }

  /* create association between program & version number and dispatch routine */
  if(svc_register(svc, prognum, versnum, dispatch, protocol) == 0)
  {
    fprintf(stderr,"svc_register() failed");
    exit(1);
  }

  /* Accept client requests */
  svc_run();

  return 1;
}

static void dispatch(rqstp, transp)    /* remote procedure */
  struct svc_req *rqstp;
  SVCXPRT        *transp;
{
  /* Dispatch Routine Code */
}
```

svcudp_create Subroutine

Important: The subroutine is exported from both the **libc** and the **libnsl** libraries.

svcudp_create Subroutine Exported from the libc Library

Purpose

Creates a User Datagram Protocol/Internet Protocol (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 UDP/IP and returns a pointer to it.

The UDP/IP service transport handle is used only for procedures that take up to 8KB of encoded arguments or results.

Parameters

Item	Description
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**.

Restrictions

The maximum number of open file descriptors that an RPC server can use has been set to 32767 so that compatibility can be maintained with RPC-server applications built on earlier releases of AIX.

Return Values

Upon successful completion, this subroutine returns a valid RPC service transport. If unsuccessful, it returns a value of null.

svcudp_create Subroutine Exported from the libnsl Library

Purpose

Creates a User Datagram Protocol/Internet Protocol (UDP/IP) service transport handle.

Library

Network Services Library (**libnsl.a**)

Syntax

```
#include <rpc/rpc.h>

SVCXPRT *svcudp_create (fd)
int fd;
```

Description

The **svcudp_create** subroutine creates a remote procedure call (RPC) service transport handle. The *fd* parameter specifies a file descriptor on the UDP transport. You can set the value of the *fd* parameter to **RPC_ANYSOCK**, so that the **svcudp_create** subroutine creates a new file descriptor on UDP transport and binds the file descriptor to a port.

The UDP/IP service transport handle is used only for procedures that take up to 8KB of encoded arguments or results.

Use the **svc_create** subroutine instead of the **svcudp_create** subroutine. The **svcudp_create** subroutine is compatible only with earlier versions of AIX.

Parameters

Item	Description
fd	Specifies the file descriptor associated with the udp transport.

Return Values

Item	Description
a valid RPC service transport handle	successful
a null value	unsuccessful

Examples

```
#include <rpc/rpc.h>

static void dispatch(struct svc_req *, SVCXPRT *);

int main()
{
  SVCXPRT *svc=NULL;
  int fd;
  int protocol = IPPROTO_UDP;

  /* Set the file descriptor to RPC_ANYFD */
  fd = RPC_ANYSOCK;

  svc = (SVCXPRT *) svcudp_create(fd);
  if(svc==NULL)
  {
    fprintf(stderr,"svcudp_create() failed");
    exit(1);
  }

  /* create association between program & version number and dispatch routine */
  if(svc_register(svc, prognum, versnum, dispatch, protocol) == 0)
  {
    fprintf(stderr,"svc_register() failed");
    exit(1);
  }

  /* Accept client requests */
  svc_run();

  return 1;
}

static void dispatch(rqstp, transp)    /* remote procedure */
  struct svc_req *rqstp;
```

```
  SVCXPRT       *transp;
{
  /* Dispatch Routine Code */
}
```

Related reference:
"registerrpc Subroutine" on page 302

user2netname Subroutine

Important: The subroutine is exported from both the **libc** and the **libnsl** libraries.

user2netname Subroutine Exported from the libc Library

Purpose

Converts from a domain-specific user ID to a network name that is independent from the operating system.

Library

C Library (**libc.a**)

Syntax

```
#include <rpc/rpc.h>

int user2netname ( name,  uid,  domain)
char *name;
int uid;
char *domain;
```

Description

The **user2netname** subroutine converts from a domain-specific user ID to a network name that is independent from the operating system.

This subroutine is the inverse of the **netname2user** subroutine.

Parameters

Item	Description
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.

user2netname Subroutine Exported from the libnsl Library

Purpose

Converts a domain-specific user name to an operating-system-independent network name.

Library

Network Services Library (**libnsl.a**)

Syntax

```
#include <rpc/rpc.h>

int user2netname( name, uid, domain)
char *name;
const uid_t uid;
const char *domain;
```

Description

The **user2netname** subroutine, which belongs to the secure remote procedure call (RPC) category, is used in applications which use the **AUTH_DES** authentication flavor. This subroutine is used on client side to generate a network name (or a netname) of the user.

This subroutine is the inverse of the **netname2user** subroutine.

Parameters

Item	Description
name	Represents the network name of the user after successful completion.
uid	Specifies a domain-specific user name.
domain	Specifies the domain.

Return Values

Item	Description
1	successful
0	unsuccessful

Examples

```
#include <rpc/rpc.h>
int main()
{
  char name[255]; /* contains netname of owner of server process */
  char rhost[255]; /* Remote host name on which server resides */
  char domain[255];
  rpcprog_t PROGNUM = 0x3fffffffL;
  rpcvers_t PROGVER = 0x1L;

  /* obtain the domainname of the host */
  if (getdomainname(domain, 255)) {
    fprintf(stderr, "\ngetdomainname() failed\n");
    exit(2);
  }

  /* Obtain network name of remote host */
  if (!user2netname(name, getuid() , domain))
  {
    fprintf(stderr, "\nhost2netname() failed\n");
    exit(EXIT_FAILURE);
  }

  /* Create a client handle for remote host rhost for PROGNUM & PROGVER on tcp transport */
  clnt = clnt_create(rhost, PROGNUM, PROGVER, "tcp");
  if (clnt == (CLIENT *) NULL) {
    fprintf(stderr,"client_create() error\n");
    exit(1);
  }
```

```
  clnt->cl_auth = auhdes_seccreate(name, 80, rhost, (des_block *)NULL);

  /*
   * Make a call to clnt_call() subroutine
   */

  /* Destroy the authentication handle */
  auth_destroy(clnt->cl_auth);

  /* Destroy the client handle in the end */
  clnt_destroy(clnt);

  return 0;
}
```

X

The following RPC subroutines begin with the letter x.

xprt_register Subroutine

Important: The subroutine is exported from both the **libc** and **libnsl** libraries.

xprt_register Subroutine Exported from the libc Library

Purpose

Registers a Remote Procedure Call (RPC) service transport handle.

Library

C Library (**libc.a**)

Syntax

```
#include <rpc/svc.h>
void xprt_register ( xprt)
SVCXPRT *xprt;
```

Description

The **xprt_register** subroutine registers an RPC service transport handle with the RPC program after the transport has been created. This subroutine modifies the **svc_fdset** global variable. The **svc_fdset** global variable indicates read file descriptor bit mask of the RPC server, which is generally required if you call the **svc_exit** subroutine. After calling the **svc_exit** subroutine, you can use the **xprt_register** subroutine to reregister RPC services.

Note: Service implementors do not usually need this subroutine.

Parameters

Item	Description
xprt	Points to the newly created RPC service transport handle.

xprt_register Subroutine Exported from the libnsl Library

Purpose

Registers a Remote Procedure Call (RPC) service transport handle.

Library

Network Services Library (**libnsl.a**)

Syntax

```
#include <rpc/rpc.h>
void xprt_register ( xprt )
const SVCXPRT *xprt;
```

Description

The **xprt_register** subroutine registers an RPC service transport handle with the RPC program after the transport has been created. This subroutine modifies the **svc_fdset** global variable. The **svc_fdset** global variable indicates read file descriptor bit mask of the RPC server, which is generally required if you call the **svc_exit** subroutine. After calling the **svc_exit** subroutine, you can use the **xprt_register** subroutine to reregister RPC services.

Note: Service implementors do not usually need this subroutine.

Parameters

Item	Description
xprt	Points to the newly created RPC service transport handle.

Examples

```
#include <stdlib.h>
#include <rpc/rpc.h>
#include <netconfig.h>


#define PROG 0x3fffffffL
#define VERS 0x1L

static void sample_dispatch();

main()
{

    SVCXPRT *svc_handle;
    struct netconfig *nconf;

    /* Get transport type and create RPC service handle */

    /* Register dispatch routine for program number and version number with RPCBIND service */

    If(svc_reg(svc_handle, PROG, VERS, sample_dispatch,nconf) == 0)
    {
        fprintf(stdout,"Error in svc_reg!");
        exit(EXIT_FAILURE);
```

```
    }

    svc_run();

    /* execution control will come here after svc_exit() is called.
    Thus when client request comes, control goes in dispatch routine where svc_exit() is called. */

    /* code to get new svc_handle */

    /* register with xprt_register */
    xprt_register(svc_handle);

    /* verify if svc_fdset is modified */

    return 0;
}

/* following is the sample dispatch routine*/
static void sample_dispatch(struct svc_req *request, SVCXPRT *xprt)
{

/*   some code  */
svc_exit();

 }
```

xprt_unregister Subroutine

Important: The subroutine is exported from both the **libc** and the **libnsl** libraries.

xprt_unregister Subroutine Exported from the libc Library

Purpose

Removes a Remote Procedure Call (RPC) service transport handle.

Library

C Library (**libc.a**)

Syntax

```
void xprt_unregister ( xprt)
SVCXPRT *xprt;
```

Description

The **xprt_unregister** subroutine removes an 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.

Parameters

Item	Description
xprt	Points to the RPC service transport handle to be destroyed.

xprt_unregister Subroutine Exported from the libnsl Library

Purpose

Removes a Remote Procedure Call (RPC) service transport handle.

Library

Network Services Library (**libnsl.a**)

Syntax

```
#include <rpc/rpc.h>
void xprt_unregister ( xprt)
const SVCXPRT *xprt;
```

Description

The **xprt_unregister** subroutine removes an RPC service transport handle from the RPC service program before the transport handle can be destroyed. This subroutine modifies the **svc_fds** global variable. The **svc_fdset** global variable indicates read file descriptor bit mask of the RPC server, which is generally required if you do not call the **svc_run** subroutine.

Note: Service implementors do not usually need this subroutine.

Parameters

Item	Description
xprt	Points to the RPC service transport handle to be destroyed.

Examples

```
#include <stdlib.h>
#include <rpc/rpc.h>
#include <netconfig.h>

#define PROG 0x3fffffffL
#define VERS 0x1L

main()
{

    SVCXPRT *svc_handle;   /* server handle */
    struct netconfig *nconf;
    int fd;

    /* Get proper file descriptor */

    /* Get transport type */

    /* Get RPC service handle */

    /* sendsize and recvsize are 0, thus default size will be chosen */

    if((svc_handle=svc_tli_create(fd, nconf, 0, 0, 0))==(SVCXPRT *)NULL)
    {
         fprintf(stdout,"Error in svc_tli_create!");
         exit(EXIT_FAILURE);
```

```
    }

    /* register it */
    xprt_register(svc_handle);

    /* unregister it */
    xprt_unregister(svc_handle);

    /* destroy the RPC service handle */
    svc_destroy(svc_handle);

    /* check if svc_fdset is modified */

      return 0;
}
```

Notices

This information was developed for products and services offered in the US.

IBM may not offer the products, services, or features discussed in this document in other countries. Consult your local IBM representative for information on the products and services currently available in your area. Any reference to an IBM product, program, or service is not intended to state or imply that only that IBM product, program, or service may be used. Any functionally equivalent product, program, or service that does not infringe any IBM intellectual property right may be used instead. However, it is the user's responsibility to evaluate and verify the operation of any non-IBM product, program, or service.

IBM may have patents or pending patent applications covering subject matter described in this document. The furnishing of this document does not grant you any license to these patents. You can send license inquiries, in writing, to:

IBM Director of Licensing
IBM Corporation
North Castle Drive, MD-NC119
Armonk, NY 10504-1785
US

For license inquiries regarding double-byte character set (DBCS) information, contact the IBM Intellectual Property Department in your country or send inquiries, in writing, to:

Intellectual Property Licensing
Legal and Intellectual Property Law
IBM Japan Ltd.
19-21, Nihonbashi-Hakozakicho, Chuo-ku
Tokyo 103-8510, Japan

INTERNATIONAL BUSINESS MACHINES CORPORATION PROVIDES THIS PUBLICATION "AS IS" WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESS OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF NON-INFRINGEMENT, MERCHANTABILITY OR FITNESS FOR A PARTICULAR PURPOSE. Some jurisdictions do not allow disclaimer of express or implied warranties in certain transactions, therefore, this statement may not apply to you.

This information could include technical inaccuracies or typographical errors. Changes are periodically made to the information herein; these changes will be incorporated in new editions of the publication. IBM may make improvements and/or changes in the product(s) and/or the program(s) described in this publication at any time without notice.

Any references in this information to non-IBM websites are provided for convenience only and do not in any manner serve as an endorsement of those websites. The materials at those websites are not part of the materials for this IBM product and use of those websites is at your own risk.

IBM may use or distribute any of the information you provide in any way it believes appropriate without incurring any obligation to you.

Licensees of this program who wish to have information about it for the purpose of enabling: (i) the exchange of information between independently created programs and other programs (including this one) and (ii) the mutual use of the information which has been exchanged, should contact:

© Copyright IBM Corp. 2015, 2016

IBM Director of Licensing
IBM Corporation
North Castle Drive, MD-NC119
Armonk, NY 10504-1785
US

Such information may be available, subject to appropriate terms and conditions, including in some cases, payment of a fee.

The licensed program described in this document and all licensed material available for it are provided by IBM under terms of the IBM Customer Agreement, IBM International Program License Agreement or any equivalent agreement between us.

The performance data and client examples cited are presented for illustrative purposes only. Actual performance results may vary depending on specific configurations and operating conditions.

Information concerning non-IBM products was obtained from the suppliers of those products, their published announcements or other publicly available sources. IBM has not tested those products and cannot confirm the accuracy of performance, compatibility or any other claims related to non-IBM products. Questions on the capabilities of non-IBM products should be addressed to the suppliers of those products.

Statements regarding IBM's future direction or intent are subject to change or withdrawal without notice, and represent goals and objectives only.

All IBM prices shown are IBM's suggested retail prices, are current and are subject to change without notice. Dealer prices may vary.

This information is for planning purposes only. The information herein is subject to change before the products described become available.

This information contains examples of data and reports used in daily business operations. To illustrate them as completely as possible, the examples include the names of individuals, companies, brands, and products. All of these names are fictitious and any similarity to actual people or business enterprises is entirely coincidental.

COPYRIGHT LICENSE:

This information contains sample application programs in source language, which illustrate programming techniques on various operating platforms. You may copy, modify, and distribute these sample programs in any form without payment to IBM, for the purposes of developing, using, marketing or distributing application programs conforming to the application programming interface for the operating platform for which the sample programs are written. These examples have not been thoroughly tested under all conditions. IBM, therefore, cannot guarantee or imply reliability, serviceability, or function of these programs. The sample programs are provided "AS IS", without warranty of any kind. IBM shall not be liable for any damages arising out of your use of the sample programs.

Each copy or any portion of these sample programs or any derivative work must include a copyright notice as follows:

© (your company name) (year).

Portions of this code are derived from IBM Corp. Sample Programs.

© Copyright IBM Corp. _enter the year or years_.

Privacy policy considerations

IBM Software products, including software as a service solutions, (“Software Offerings”) may use cookies or other technologies to collect product usage information, to help improve the end user experience, to tailor interactions with the end user or for other purposes. In many cases no personally identifiable information is collected by the Software Offerings. Some of our Software Offerings can help enable you to collect personally identifiable information. If this Software Offering uses cookies to collect personally identifiable information, specific information about this offering’s use of cookies is set forth below.

This Software Offering does not use cookies or other technologies to collect personally identifiable information.

If the configurations deployed for this Software Offering provide you as the customer the ability to collect personally identifiable information from end users via cookies and other technologies, you should seek your own legal advice about any laws applicable to such data collection, including any requirements for notice and consent.

For more information about the use of various technologies, including cookies, for these purposes, see IBM’s Privacy Policy at http://www.ibm.com/privacy and IBM’s Online Privacy Statement at http://www.ibm.com/privacy/details the section entitled “Cookies, Web Beacons and Other Technologies” and the “IBM Software Products and Software-as-a-Service Privacy Statement” at http://www.ibm.com/software/info/product-privacy.

Trademarks

IBM, the IBM logo, and ibm.com are trademarks or registered trademarks of International Business Machines Corp., registered in many jurisdictions worldwide. Other product and service names might be trademarks of IBM or other companies. A current list of IBM trademarks is available on the web at Copyright and trademark information at www.ibm.com/legal/copytrade.shtml.

UNIX is a registered trademark of The Open Group in the United States and other countries.

Index

A

B

C

D

© Copyright IBM Corp. 2015, 2016

E

F

G

H

I

K

L

M

N

O

P

R

S

T

U

W

X

Y

Printed in USA