Bull

Technical Reference Communications

Volume 2/2

AIX

ORDER REFERENCE
86 A2 84AP 04

Bull

Bull

Technical Reference Communications

Volume 2/2

AIX

Software

February 1999

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

ORDER REFERENCE
86 A2 84AP 04

Printed in France

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

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

Trademarks and Acknowledgements

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

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

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

Year 2000

The product documented in this manual is Year 2000 Ready.

Contents

About This Book

Communications Technical Reference, Volumes 1 and 2 provide information on application programming interfaces to the Advanced Interactive Executive Operating System (referred to in this text as AIX) for use on system units.

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

- *Base Operating System and Extensions, Volumes 1 and 2* provide information on system calls, subroutines, functions, macros, and statements associated with AIX base operating system runtime services.
- *Communications, Volumes 1 and 2* provide information on entry points, functions, system calls, subroutines, and operations related to communications services.
- *Kernel and Subsystems, Volumes 1 and 2* provide information about kernel services, device driver operations, file system operations, subroutines, the configuration subsystem, the communications subsystem, the low function terminal (LFT) subsystem, the logical volume subsystem, the M–audio capture and playback adapter subsystem, the printer subsystem, the SCSI subsystem, and the serial DASD subsystem.

Who Should Use This Book

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

How to Use This Book

Overview of Contents

Communications consists of two parts (*Communications Technical Reference, Volumes 1 and 2*), each containing system calls (called subroutines), subroutines, functions, macros, and statements alphabetically arranged per topic.

Communications Technical Reference, Volume 1 contains the following topics:

- Data Link Controls (DLC)
- Data Link Provider Interface (DLPI)
- eXternal Data Representation
- AIX 3270 Host Connection Program (HCON)
- Network Computing System (NCS)
- Network Information Service (NIS)
- Remote Procedure Calls (RPC)

Communications Technical Reference, Volume 2 contains the following topics:

- Simple Network Management Protocol (SNMP)
- Sockets
- Streams
- X.25 Application

Highlighting

The following highlighting conventions are used in this book:

Bold	Identifies commands, keywords, files, directories, and other items whose names are predefined by the system.
Italics	Identifies parameters whose actual names or values are to be supplied by the user.
`Monospace`	Identifies examples of specific data values, examples of text similar to what you might see displayed, examples of program code similar to what you might write as a programmer, messages from the system, or information you actually type.

ISO 9000

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

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

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

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

To determine the proper way to develop a UNIX95–portable application, you may need to refer to the X/Open UNIX95 Specification, which can be obtained on a CD–ROM by ordering the printed copy of *AIX Commands Reference*, order number 86 A2 38JX to 86 A2 43JX, or by ordering *Go Solo: How to Implement and Go Solo with the Single Unix Specification*, order number SR28–5705, a book which includes the X/Open UNIX95 Specification on a CD–ROM.

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

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

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

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

Related Publications

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

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

- *AIX 4.3 System Management Guide: Communications and Networks*, Order Number 86 A2 31JX.
- *AIX Kernel Extensions and Device Support Programming Concepts*, Order Number 86 A2 36JX.
- *AIX Files Reference*, Order Number 86 A2 79AP.
- *AIX Version 4.3 Problem Solving Guide and Reference*, Order Number 86 A2 32JX.
- *VM/SP–Entry System Programmer's Guide,* Order Number SC23–0432.
- *MVS/XA System Macros and Facilities, Volumes 1 and 2*, Order Number GBOF–1014.
- *MVS/XA Supervisor Services and Macros,* Order Number GC28–1154.

Ordering Publications

You can order this book separately from Bull Electronics Angers S.A. CEDOC. See address in the Ordering Form at the end of this book.

If you received a printed copy of *Documentation Overview* with your system, use that book for information on related publications and for instructions on ordering them.

To order additional copies of this book, use the following order number: 86 A2 84AP.

To order additional copies of the six–volume set, order *AIX Technical Reference*, Order Number 86 A2 81AP to 86 A2 91AP.

Chapter 9. Simple Network Management Protocol (SNMP)

getsmuxEntrybyname or getsmuxEntrybyidentity Subroutine

Purpose

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

Library

SNMP Library (**libsnmp.a**)

Syntax

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

struct smuxEntry *getsmuxEntrybyname (name)
char *name;

struct smuxEntry *getsmuxEntrybyidentity (identity)
OID identity;
```

Description

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

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

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

Parameters

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

Return Values

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

Implementation Specifics

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

Files

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

Related Information

List of Network Manager Programming References.

SNMP Overview for Programmers in *AIX Version 4 Communications Programming Concepts*.

isodetailor Subroutine

Purpose

Initializes variables for various logging facilities.

Library

ISODE Library (**libisode.a**)

Syntax

```
#include <isode/tailor.h>

void isodetailor (myname, wantuser)
char *myname;
int wantuser;
```

Description

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

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

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

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

Parameters

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

Implementation Specifics

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

Files

/etc/isodetailor	Location of user's copy of the **/usr/lpp/snmpd/smux/isodetailor** file.
/usr/lpp/snmpd/smux/isodetailor	Contains a complete list of all the logging parameters.

Related Information

The **ll_hdinit**, **ll_dbinit**, **_ll_log**, or **ll_log** subroutine.

SNMP Overview for Programmers in *AIX Communications Programming Concepts*.

ll_hdinit, ll_dbinit, _ll_log, or ll_log Subroutine

Purpose

Reports errors to log files.

Library

ISODE Library (**libisode.a**)

Syntax

```
#include <isode/logger.h>

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

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

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

int ll_log (va_alist)
va_dcl
```

Description

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

The **LLog** structure contains the following fields:

```
typedef struct ll_struct
{
char      *ll_file;    /* path name to logging file         */
char      *ll_hdr;     /* text to put in opening line       */
char      *ll_dhdr;    /* dynamic header - changes          */
int       ll_events;   /* loggable events                   */
int       ll_syslog;   /* loggable events to send to syslog */
                       /* takes same values as ll_events    */
int       ll_msize;    /* max size for log, in Kbytes       */
                       /* If ll_msize < 0, then no checking */
int       ll_stat;     /* assorted switches                 */
int       ll_fd;       /* file descriptor                   */
} LLog;
```

The possible values for the `ll_events` and `ll_syslog` fields are:

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

The possible values for the `ll_stat` field are:

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

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

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

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

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

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

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

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

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

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

where:

`mm/dd`	Specifies the date.
`hh:mm:ss`	Specifies the time.
`ll_hdr`	Specifies the value of the `ll_hdr` field of the **LLog** structure.
`ll_dhdr`	Specifies the value of the `ll_dhdr` field of the **LLog** structure.
`string_format`	Specifies the string format passed to the **ll_log** subroutine, with the extra variables filled in.
`what`	Specifies the variable that tells what has occurred. The `what` variable often contains the reason for the failure. For example if the memory device, /**dev**/**mem**, fails, the `what` variable contains the name of the /**dev**/**mem** device.
`system_error`	Contains the string for the **errno** value, if it exists.

Parameters

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

Return Values

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

Implementation Specifics

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

Related Information

The **isodetailor** subroutine.

Examples of SMUX Error Logging Routines, and SNMP Overview for Programmers in *AIX Communications Programming Concepts*.

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

Purpose

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

Library

SNMP Library (**libsnmp.a**)

Syntax

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

int o_number (oi, v, number)
OI oi;
register struct type_SNMP_VarBind *v;
int number;

#define o_integer (oi, v, number) o_number ((oi), (v), (number))

int o_string (oi, v, base, len)
OI oi;
register struct type_SNMP_VarBind *v;
char *base;
int len;

int o_igeneric (oi, v, offset)
OI oi;
register struct type_SNMP_VarBind *v;
int offset;

int o_generic (oi, v, offset)
OI oi;
register struct type_SNMP_VarBind *v;
int offset;

int o_specific (oi, v, value)
OI oi;
register struct type_SNMP_VarBind *v;
caddr_t value;

int o_ipaddr (oi, v, netaddr)
OI oi;
register struct type_SNMP_VarBind *v;
struct sockaddr_in *netaddr;
```

Description

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

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

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

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

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

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

Parameters

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

Return Values

The return values for these subroutines are:

int_SNMP_error__status_genErr	Indicates an error occurred when setting the *v* parameter value.
int_SNMP_error__status_noErr	Indicates no errors found.

Implementation Specifics

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

Related Information

List of Network Manager Programming References.

SNMP Overview for Programmers, and Working with Management Information Base (MIB) Variables in *AIX Communications Programming Concepts*.

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

Purpose

Manipulates the object identifier data structure.

Library

ISODE Library (**libisode.a**)

Syntax

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

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

OID oid_cpy (oid)
OID oid;

void oid_free (oid)
OID oid;

char *sprintoid (oid)
OID oid;

OID str2oid (s)
char *s;

OID ode2oid (descriptor)
char *descriptor;

char *oid2ode (oid)
OID oid;

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

OID prim2oid (pe)
PE pe;

PE oid2prim (oid)
OID oid;
```

Description

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

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

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

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

The **oid2ode** subroutine is identical to the **sprintoid** subroutine except that the **oid2ode** subroutine checks whether the *oid* parameter is in the **isobjects** database. The **oid2ode** subroutine is implemented as a macro call to the **oid2ode_aux** subroutine. The

oid2ode_aux subroutine is similar to the **oid2ode** subroutine except for an additional integer parameter that specifies whether the string should be enclosed by quotes. The **oid2ode** subroutine always encloses the string in quotes.

The **ode2oid** subroutine retrieves an object identifier from the **isobjects** database.

Parameters

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

Return Values

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

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

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

Implementation Specifics

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

Related Information

The **oid_extend** subroutine, **oid_normalize** subroutine.

List of Network Manager Programming References.

SNMP Overview for Programmers in *AIX Communications Programming Concepts*.

oid_extend or oid_normalize Subroutine

Purpose

Extends the base ISODE library subroutines.

Library

SNMP Library (**libsnmp.a**)

Syntax

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

OID oid_extend (q, howmuch)
OID q;
integer howmuch;

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

Description

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

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

These subroutines do not free the *q* parameter.

Parameters

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

Return Values

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

Implementation Specifics

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

Related Information

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

SNMP Overview for Programmers in *AIX Communications Programming Concepts*.

readobjects Subroutine

Purpose

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

Library

SNMP Library (**libsnmp.a**)

Syntax

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

int
readobjects (file)
char *file;
```

Description

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

```
<MIB directory> <MIB position>

<MIB name> <MIB position> <MIB type> <MIB access> <MIB required?>
<MIB name> <MIB position> <MIB type> <MIB access> <MIB required?>
...
```

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

Parameters

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

Return Values

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

Implementation Specifics

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

Related Information

The **text2oid** subroutine.

RFC 1155 describes the basic MIB structure.

SNMP Overview for Programmers in *AIX Communications Programming Concepts*.

s_generic Subroutine

Purpose

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

Library

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

Syntax

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

int s_generic
(oi, v, offset)
OI oi;
register struct type_SNMP_VarBind *v;
int offset;
```

Description

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

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

Parameters

oi	Designates the **OI** structure representing the MIB variable to be set.
v	Specifies the variable binding that contains the value to be set.
offset	Contains the stage of the set. The possible values for the *offset* parameter are **type_SNMP_PDUs_commit**, **type_SNMP_PDUs_rollback**, or **type_SNMP_PDUs_set__request**.

Return Values

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

Implementation Specifics

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

Related Information

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

SNMP Overview for Programmers, and Understanding SNMP Daemon Processing in *AIX Communications Programming Concepts*.

smux_close Subroutine

Purpose

Ends communications with the SNMP agent.

Library

SNMP Library (**libsnmp.a**)

Syntax

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

int smux_close (reason)
int reason;
```

Description

The **smux_close** subroutine closes the transmission control protocol (TCP) connection from the SNMP multiplexing (SMUX) peer. The **smux_close** subroutine sends the close protocol data unit (PDU) with the error code set to the *reason* value. The subroutine closes the TCP connection and frees the socket. This subroutine also frees information it was maintaining for the connection.

Parameters

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

Return Values

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

Error Codes

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

invalidOperation	Indicates that the **smux_init** subroutine has not been executed successfully.
congestion	Indicates that memory could not be allocated for the close PDU. The TCP connection is closed.
youLoseBig	Indicates that the SNMP code has a problem. The TCP connection is closed.

Implementation Specifics

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

Related Information

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

RFC 1227, SNMP MUX Protocol and MIB.

SNMP Overview for Programmers in *AIX Communications Programming Concepts*.

smux_error Subroutine

Purpose

Creates a readable string from the **smux_errno** global variable value.

Library

SNMP Library (**libsnmp.a**)

Syntax

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

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

Description

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

Parameters

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

Return Values

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

Implementation Specifics

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

Related Information

The **smux_close** subroutine, **smux_init** subroutine, **smux_register** subroutine, **smux_response** subroutine, **smux_simple_open** subroutine, **smux_trap** subroutine, **smux_wait** subroutine.

RFC 1227, SNMP MUX Protocol and MIB.

SNMP Overview for Programmers in *AIX Communications Programming Concepts*.

smux_free_tree Subroutine

Purpose

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

Library

SNMP Library (**libsnmp.a**)

Syntax

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

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

Description

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

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

This subroutine does not return values or error codes.

Parameters

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

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

Implementation Specifics

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

Related Information

The **snmpd** command.

The **readobjects** subroutine, **smux_register** subroutine.

SNMP Overview for Programmers in *AIX Communications Programming Concepts*.

smux_init Subroutine

Purpose

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

Library

SNMP Library (**libsnmp.a**)

Syntax

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

int smux_init (debug)
int debug;
```

Description

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

Parameters

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

Return Values

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

Error Codes

Possible values for the **smux_errno** global variable are:

congestion	Indicates memory allocation problems
youLoseBig	Signifies problem with SNMP library code
systemError	Indicates TCP connection failure.

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

Implementation Specifics

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

Related Information

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

RFC 1227, SNMP MUX Protocol and MIB.

SNMP Overview for Programmers in *AIX Communications Programming Concepts*.

smux_register Subroutine

Purpose

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

Library

SNMP Library (**libsnmp.a**)

Syntax

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

int smux_register (subtree, priority, operation)

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

Description

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

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

Parameters

subtree	Indicates an object identifier that contains the root of the MIB tree to be registered.
priority	Indicates the level of responsibility that the SMUX peer has on the MIB tree. The priority levels range from 0 to (2^31 – 2). The lower the priority number, the higher the priority. A priority of –1 tells the SNMP daemon to assign the highest priority currently available.
operation	Specifies the operation for the SNMP agent to apply to the MIB tree. Possible values are **delete**, **readOnly**, or **readWrite**. The **delete** operation removes the MIB tree from the SMUX peers in the eyes of the SNMP agent. The other two values specify the operations allowed by the SMUX peer on the MIB tree that is being registered with the SNMP agent.

Return Values

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

Error Codes

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

parameterMissing	Indicates a parameter was null. When the parameter is fixed, the **smux_register** subroutine can be reissued.
invalidOperation	Indicates that the **smux_register** subroutine is trying to perform this operation before a **smux_init** operation has successfully completed. Start over with a new **smux_init** subroutine call.
congestion	Indicates a memory problem occurred. The TCP connection is closed. Start over with a new **smux_init** subroutine call.
youLoseBig	Indicates an SNMP code problem has occurred. The TCP connection is closed. Start over with a new **smux_init** subroutine call.

Implementation Specifics

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

Related Information

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

RFC1227, SNMP MUX Protocol and MIB.

SNMP Overview for Programmers in *AIX Communications Programming Concepts*.

smux_response Subroutine

Purpose

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

Library

SNMP Library (**libsnmp.a**)

Syntax

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

int smux_response (event)
struct type_SNMP_GetResponse__PDU *event;
```

Description

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

Parameters

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

Return Values

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

Error Codes

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

parameterMissing	Indicates the parameter was null. When the parameter is fixed, the subroutine can be reissued.
invalidOperation	Indicates the subroutine was attempted before the **smux_init** subroutine successfully completed. Start over with the **smux_init** subroutine.
youLoseBig	Indicates a SNMP code problem has occurred and the TCP connection is closed. Start over with the **smux_init** subroutine.

Implementation Specifics

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

Related Information

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

RFC 1227, SNMP MUX Protocol and MIB.

SNMP Overview for Programmers in *AIX Communications Programming Concepts*.

smux_simple_open Subroutine

Purpose

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

Library

SNMP Library (**libsnmp.a**)

Syntax

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

int smux_simple_open (identity, description, commname, commlen)
OID identity;
char *description;
char *commname;
int commlen;
```

Description

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

Parameters

identity	Specifies an object identifier that describes the SMUX peer.
description	Contains a string of characters that describes the SMUX peer. The *description* parameter value cannot be longer than 254 characters.
commname	Contains the password to be sent to the SNMP agent. Can be a null value.
commlen	Indicates the length of the community name (*commname* parameter) to be sent to the SNMP agent. The value for this parameter must be at least 0.

Return Values

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

Error Codes

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

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

Implementation Specifics

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

Related Information

The **smux_close** subroutine, **smux_error** subroutine, **smux_init** subroutine, **smux_register** subroutine, **smux_response** subroutine, **smux_trap** subroutine, **smux_wait** subroutine.

List of Network Manager Programming References.

RFC 1227, SNMP MUX Protocol and MIB.

SNMP Overview for Programmers in *AIX Communications Programming Concepts*.

smux_trap Subroutine

Purpose

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

Library

SNMP Library (**libsnmp.a**)

Syntax

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

int smux_trap (generic, specific, bindings)

int generic;
int specific;
struct type_SNMP_VarBindList *bindings;
```

Description

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

Parameters

generic

Contains an integer specifying the generic trap type. The value must be one of the following:

0	Specifies a cold start.
1	Specifies a warm start.
2	Specifies a link down.
3	Specifies a link up.
4	Specifies an authentication failure.
5	Specifies an EGP neighbor loss.
6	Specifies an enterprise–specific trap type.

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

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

Return Values

The subroutine returns **NOTOK** on failure and **OK** on success.

Error Codes

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

invalidOperation	Indicates the Transmission Control Protocol (TCP) connection was not completed.
congestion	Indicates memory is not available. The TCP connection was closed.
youLoseBig	Indicates an error occurred in the SNMP code. The TCP connection was closed.

Implementation Specifics

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

Related Information

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

RFC 1227, SNMP MUX Protocol and MIB.

SNMP Overview for Programmers in *AIX Communications Programming Concepts*.

smux_wait Subroutine

Purpose

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

Library

SNMP Library (**libsnmp.a**)

Syntax

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

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

Description

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

Parameters

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

Return Values

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

Error Codes

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

parameterMissing	Indicates that the *event* parameter value was null.
inProgress	Indicates that there was nothing for the subroutine to receive.
invalidOperation	Indicates that the **smux_init** subroutine was not called or failed to operate.
youLoseBig	Indicates an error occurred in the SNMP code. The TCP connection was closed.

Implementation Specifics

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

Related Information

The **smux_close** subroutine, **smux_error** subroutine, **smux_init** subroutine, **smux_register** subroutine, **smux_response** subroutine, **smux_simple_open** subroutine, **smux_trap** subroutine.

RFC1227, SNMP MUX Protocol and MIB.

SNMP Overview for Programmers in *AIX Communications Programming Concepts*.

text2inst, name2inst, next2inst, or nextot2inst Subroutine

Purpose

Retrieves instances of variables from various forms of data.

Library

SNMP Library (**libsnmp.a**)

Syntax

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

OI text2inst (text)
char *text;

OI name2inst (oid)
OID oid;

OI next2inst (oid)
OID oid;

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

Description

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

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

The **name2inst** subroutine uses an object identifier structure specified in the *oid* parameter to specify which object is desired. If the object cannot be found, a **NULLOI** value is returned.

The **next2inst** and **nextot2inst** subroutines find the next object in the database given an object identifier. The **next2inst** subroutine starts at the root of the tree, while the **nextot2inst** subroutine starts at the object given in the *ot* parameter. If another object cannot be found, the **NULLOI** value will be returned.

Parameters

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

Return Values

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

Implementation Specifics

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

Related Information

The **text2oid** subroutine, **text2obj** subroutine.

SNMP Overview for Programmers in *AIX Communications Programming Concepts*.

text2oid or text2obj Subroutine

Purpose

Converts a text string into some other value.

Library

SNMP Library (**libsnmp.a**)

Syntax

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

OID text2oid (text)
char *text;

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

Description

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

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

Parameters

text	Contains a text string used to specify the object identifier or object to be returned.

Return Values

On a successful execution, these subroutines return completed data structures. If a failure occurs, the **text2oid** subroutine returns a **NULLOID** value and the **text2obj** returns a **NULLOT** value.

Implementation Specifics

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

Related Information

The **malloc** subroutine, **oid_free** subroutine, **text2inst** subroutine.

SNMP Overview for Programmers in *AIX Communications Programming Concepts*.

Chapter 10. Sockets

accept Subroutine

Purpose

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

Library

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

Syntax

```
#include <sys/types.h>
#include <sys/socket.h>
#include <sys/atmsock.h> /*Needed for SOCK_CONN_DGRAM sockets
type only*/

int accept (Socket, Address, AddressLength)
int Socket;
struct sockaddr *Address;
size_t *AddressLength;
```

Description

The **accept** subroutine extracts the first connection on the queue of pending connections, creates a new socket with the same properties as the specified socket, and allocates a new file descriptor for that socket.

If the **listen** queue is empty of connection requests, the **accept** subroutine:

- Blocks a calling socket of the blocking type until a connection is present.
- Returns an **EWOULDBLOCK** error code for sockets marked nonblocking.

The accepted socket cannot accept more connections. The original socket remains open and can accept more connections.

The **accept** subroutine is used with **SOCK_STREAM** and **SOCK_CONN_DGRAM** socket types.

For **SOCK_CONN_DGRAM** socket type and **ATM** protocol, a socket is not ready to transmit/receive data until **SO_ATM_ACCEPT** socket option is called. This allows notification of an incoming connection to the application, followed by modification of appropriate parameters and then indicate that a connection can become fully operational.

Parameters

Socket	Specifies a socket created with the **socket** subroutine that is bound to an address with the **bind** subroutine and has issued a successful call to the **listen** subroutine.
Address	Specifies a result parameter that is filled in with the address of the connecting entity as known to the communications layer. The exact format of the *Address* parameter is determined by the domain in which the communication occurs.
AddressLength	Specifies a parameter that initially contains the amount of space pointed to by the *Address* parameter. Upon return, the parameter contains the actual length (in bytes) of the address returned. The **accept** subroutine is used with **SOCK_STREAM** socket types.

Return Values

Upon successful completion, the **accept** subroutine returns the nonnegative socket descriptor of the accepted socket.

If the **accept** subroutine is unsuccessful, the subroutine handler performs the following functions:

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

Error Codes

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

EBADF	The *Socket* parameter is not valid.
ENOTSOCK	The *Socket* parameter refers to a file, not a socket.
EOPNOTSUPP	The referenced socket is not of type **SOCK_STREAM**.
EFAULT	The *Address* parameter is not in a writable part of the user address space.
EWOULDBLOCK	The socket is marked as nonblocking, and no connections are present to be accepted.
ENETDOWN	The network with which the socket is associated is down.
ENOTCONN	The socket is not in the connected state.

Examples

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

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

Implementation Specifics

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

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

Related Information

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

Accepting UNIX Stream Connections Example Program, Binding Names to Sockets, Sockets Overview, Understanding Socket Connections, and Understanding Socket Creation in *AIX Communications Programming Concepts*.

bind Subroutine

Purpose

Binds a name to a socket.

Library

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

Syntax

```
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <sys/ndd_var.h>  /*Needed for AF_NDD address family
only*/
#include <sys/atmsock.h> /*Needed for SOCK_CONN_DGRAM socket type
only*/

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

Description

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

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

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

Parameters

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

Return Values

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

If the **bind** subroutine is unsuccessful, the subroutine handler performs the following actions:

- Returns a value of –1 to the calling program.
- Moves an error code, indicating the specific error, into the **errno** global variable. For further explanation of the **errno** variable see ”Error Notification Object Class” in *AIX Communications Programming Concepts*.

Error Codes

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

EBADF	The *Socket* parameter is not valid.
ENOTSOCK	The *Socket* parameter refers to a file, not a socket.
EADDRNOTAVAIL	The specified address is not available from the local machine.
EADDRINUSE	The specified address is already in use.
EINVAL	The socket is already bound to an address.
EACCES	The requested address is protected, and the current user does not have permission to access it.
EFAULT	The *Address* parameter is not in a writable part of the *UserAddress* space.
ENODEV	The specified device does not exist.

Examples

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

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

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

Implementation Specifics

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

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

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

Related Information

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

Binding Names to Sockets, Reading UNIX Datagrams Example Program, Sockets Overview, Understanding Socket Connections, and Understanding Socket Creation in *AIX Communications Programming Concepts*.

connect Subroutine

Purpose

Connects two sockets.

Library

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

Syntax

```
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <sys/ndd_var.h> /*Needed for AF_NDD address family
only*/
#include <sys/atmsock.h> /*Needed for SOCK_CONN_DGRAM socket type
only*/

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

Description

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

If a **connect** subroutine is issued on an unbound socket, the system automatically binds the socket.

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

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

Parameters

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

Return Values

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

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

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

Error Codes

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

EBADF	The *Socket* parameter is not valid.
ENOTSOCK	The *Socket* parameter refers to a file, not a socket.
EADDRNOTAVAIL	The specified address is not available from the local machine.
EAFNOSUPPORT	The addresses in the specified address family cannot be used with this socket.
EISCONN	The socket is already connected.
ETIMEDOUT	The establishment of a connection timed out before a connection was made.
ECONNREFUSED	The attempt to connect was rejected.
ENETUNREACH	No route to the network or host is present.
EADDRINUSE	The specified address is already in use.
EFAULT	The *Address* parameter is not in a writable part of the user address space.
EINPROGRESS	The socket is marked as nonblocking. The connection cannot be immediately completed. The application program can select the socket for writing during the connection process.
EINVAL	The specified path name contains a character with the high–order bit set.
ENETDOWN	The specified physical network is down.
ENOSPC	There is no space left on a device or system table.
ENOTCONN	The socket could not be connected.

Examples

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

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

Implementation Specifics

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

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

Related Information

The **accept** subroutine, **bind** subroutine, **getsockname** subroutine, **send** subroutine, **socket** subroutine.

Initiating UNIX Stream Connections Example Program, Sockets Overview, and Understanding Socket Connections in *AIX Communications Programming Concepts*.

dn_comp Subroutine

Purpose

Compresses a domain name.

Library

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

Syntax

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


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

Description

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

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

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

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

Parameters

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

Return Values

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

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

Implementation Specifics

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

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

Files

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

Related Information

The **named** daemon.

The **dn_expand** subroutine, **_getlong** subroutine, **_getshort** subroutine, **putlong** subroutine, **putshort** subroutine, **res_init** subroutine, **res_mkquery** subroutine, **res_query** subroutine, **res_search** subroutine, **res_send** subroutine.

TCP/IP Name Resolution in *AIX 4.3 System Management Guide: Communications and Networks*.

Sockets Overview, and Understanding Domain Name Resolution in *AIX Communications Programming Concepts*

dn_expand Subroutine

Purpose

Expands a compressed domain name.

Library

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

Syntax

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


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

Description

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

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

Parameters

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

Return Values

Upon successful completion, the **dn_expand** subroutine returns the size of the expanded domain name.

If unsuccessful, the **dn_expand** subroutine returns a value of –1 to the calling program.

Implementation Specifics

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

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

Files

/etc/resolv.conf

Defines name server and domain name constants, structures, and values.

Related Information

The **dn_comp** subroutine, **_getlong** subroutine, **getshort** subroutine, **putlong** subroutine, **putshort** subroutine, **res_init** subroutine, **res_mkquery** subroutine, **res_query** subroutine, **res_search** subroutine, **res_send** subroutine.

TCP/IP Name Resolution in *AIX 4.3 System Management Guide: Communications and Networks*.

Sockets Overview, and Understanding Domain Name Resolution in *AIX Communications Programming Concepts*.

endhostent Subroutine

Purpose

Closes the /**etc**/**hosts** file.

Library

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

Syntax

```
#include <netdb.h>

endhostent ()
```

Description

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

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

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

Implementation Specifics

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

Files

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

Related Information

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

Sockets Overview and Network Address Translation in *AIX Communications Programming Concepts*.

endnetent Subroutine

Purpose

Closes the /**etc**/**networks** file.

Library

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

Syntax

```
#include <netdb.h>

void endnetent ( )
```

Description

Attention: Do not use the **endnetent** subroutine in a multithreaded environment. See the multithread alternative in the **endnetent_r** subroutine article.

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

Return Values

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

Examples

To close the /**etc**/**networks** file:

```
endnetent();
```

Implementation Specifics

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

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

Files

/etc/networks

Contains official network names.

Related Information

The **exit** subroutine, **getnetbyaddr** subroutine, **getnetbyname** subroutine, **getnetent** subroutine, **setnetent** subroutine.

Sockets Overview, Understanding Network Address Translation, and List of Socket Programming References in *AIX Communications Programming Concepts*.

endprotoent Subroutine

Purpose

Closes the /**etc**/**protocols** file.

Library

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

Syntax

#include <netdb.h>

void endprotoent ( )

Description

Attention: Do not use the **endprotoent** subroutine in a multithreaded environment. See the multithread alternative in the **endprotoent_r** subroutine article.

The **endprotoent** subroutine closes the /**etc**/**protocols** file.

Calls made to the **getprotoent** subroutine, **getprotobyname** subroutine, or **getprotobynumber** subroutine open the /**etc**/**protocols** file. An application program can use the **endprotoent** subroutine to close the /**etc**/**protocols** file.

Return Values

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

Examples

To close the /**etc**/**protocols** file:

```
endprotoent();
```

Implementation Specifics

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

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

Files

/etc/protocols

Contains protocol names.

Related Information

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

Sockets Overview, and Understanding Network Address Translation in *AIX Communications Programming Concepts*.

endservent Subroutine

Purpose

Closes the /**etc**/**services** file.

Library

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

Syntax

```
#include <netdb.h>

void endservent ( )
```

Description

Attention: Do not use the **endservent** subroutine in a multithreaded environment. See the multithread alternative in the **endservent_r** subroutine article.

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

Return Values

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

Examples

To close the /**etc**/**services** file:

```
endservent ();
```

Implementation Specifics

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

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

Files

/etc/services

Contains service names.

Related Information

The **endprotoent** subroutine, **exit** subroutine, **getprotobyname** subroutine, **getprotobynumber** subroutine, **getprotoent** subroutine, **getservbyname** subroutine, **getservbyport** subroutine, **getservent** subroutine, **setprotoent** subroutine, **setservent** subroutine.

Sockets Overview, Understanding Network Address Translation, and List of Socket Programming References in *AIX Communications Programming Concepts*.

ether_ntoa, ether_aton, ether_ntohost, ether_hostton, or ether_line Subroutine

Purpose

Maps 48–bit Ethernet numbers.

Library

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

Syntax

```
#include <sys/types.h>
#include <sys/socket.h>
#include <net/if.h>
#include <netinet/in.h>
#include <netinet/if_ether.h>

char *ether_ntoa (EthernetNumber)
struct ether_addr *EthernetNumber;

struct ether_addr *other_aton(String);
char *string

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

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

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

Description

Attention: Do not use the **ether_ntoa** or **ether_aton** subroutine in a multithreaded environment. See the multithread alternatives in the **ether_ntoa_r** or **ether_aton_r** subroutine article.

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

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

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

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

Parameters

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

Return Values

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

Implementation Specifics

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

Files

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

Related Information

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

getdomainname Subroutine

Purpose

Gets the name of the current domain.

Library

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

Syntax

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

Description

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

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

Note: Domain names are restricted to 64 characters.

Parameters

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

Return Values

If the call succeeds, a value of 0 is returned. If the call is unsuccessful, a value of –1 is returned and an error code is placed in the **errno** global variable.

Error Codes

The following error may be returned by this subroutine:

EFAULT	The *Name* parameter gave an invalid address.

Implementation Specifics

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

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

Related Information

The **gethostname** subroutine, **setdomainname** subroutine, **sethostname** subroutine.

Sockets Overview in *AIX Communications Programming Concepts*.

gethostbyaddr Subroutine

Purpose

Gets network host entry by address.

Library

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

Syntax

```
#include <netdb.h>

struct hostent *gethostbyaddr (Address, Length, Type)
char *Address;
int Length, Type;
```

Description

Attention: Do not use the **gethostbyaddr** subroutine in a multithreaded environment. See the multithread alternative in the **gethostbyaddr_r** subroutine article.

Attention: Do not use the **gethostbyaddr** subroutine in a multithreaded environment.

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

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

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

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

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

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

Parameters

Address	Specifies a host address. The host address is passed as a pointer to the binary format address.
Length	Specifies the length of host address.
Type	Specifies the domain type of the host address. This currently works only on the address family **AF_INET**.

Return Values

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

Note: The return value points to static data that is overwritten by subsequent calls so it must be copied if it is to be saved.

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

Error Codes

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

HOST_NOT_FOUND	The host specified by the *Name* parameter is not found.
TRY_AGAIN	The local server does not receive a response from an authoritative server. Try again later.
NO_RECOVERY	This error code indicates an unrecoverable error.
NO_ADDRESS	The requested *Address* parameter is valid but does not have a name at the name server.
SERVICE_UNAVAILABLE	None of the name services specified are running or available.

Implementation Specifics

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

Files

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

Related Information

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

Sockets Overview, and Network Address Translation in *AIX Communications Programming Concepts*.

gethostbyname Subroutine

Purpose

Gets network host entry by name.

Library

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

Syntax

```
#include <netdb.h>

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

Description

Note: The **gethostbyname** subroutine is threadsafe in AIX Version 4.3.

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

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

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

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

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

Parameters

Name	Points to the host name.

Return Values

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

Note: The return value points to static data that is overwritten by subsequent calls so it must be copied it it is to be saved.

If the parameter *Name* passed to **gethostbyname** is actually an IP address, **gethostbyname** will return a non–NULL hostent structure with an IP address as the hostname without actually doing a lookup. Remember to call **inet_addr** subroutine to make

sure *Name* is not an IP address before calling **gethostbyname**. To resolve an IP address call **gethostbyaddr** instead.

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

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

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

Error Codes

The **gethostbyname** subroutine is unsuccessful if any of the following errors occurs:

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

Examples

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

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

Implementation Specifics

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

Files

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

Related Information

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

Sockets Overview and Network Address Translation in *AIX Communications Programming Concepts*.

gethostent Subroutine

Purpose

Retrieves a network host entry.

Library

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

Syntax

```
#include <netdb.h>

struct hostent *gethostent ()
```

Description

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

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

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

Return Values

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

Note: The return value points to static data that is overwritten by subsequent calls, so it must be copied to be saved.

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

Implementation Specifics

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

Files

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

Related Information

The **gethostbyaddr** subroutine, **gethostbyname** subroutine, **sethostent** subroutine **endhostent** subroutine.

Sockets Overview and Network Address Translation in *AIX Communications Programming Concepts*.

gethostid Subroutine

Purpose

Gets the unique identifier of the current host.

Library

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

Syntax

```
#include <unistd.h>

int gethostid ( )
```

Description

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

Return Values

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

If the **gethostid** subroutine is unsuccessful, the system handler performs the following functions:

- Returns a value of –1 to the calling program.
- Moves an error code, indicating the specific error, into the **errno** global variable. For further explanation of the **errno** variable see ”Error Notification Object Class” in *AIX Communications Programming Concepts*.

Implementation Specifics

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

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

Related Information

The **gethostname** subroutine, **sethostid** subroutine, **sethostname** subroutine.

Sockets Overview in *AIX Communications Programming Concepts*.

gethostname Subroutine

Purpose

Gets the name of the local host.

Library

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

Syntax

```
#include <unistd.h>

int gethostname (Name, NameLength)
char *Name;
int NameLength;
```

Description

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

The **gethostname** subroutine allows a calling process to determine the internal host name for a machine on a network.

Parameters

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

Return Values

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

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

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

Error Codes

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

EFAULT	The *Name* parameter or *NameLength* parameter gives an invalid address.

Implementation Specifics

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

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

Related Information

The **gethostid** subroutine, **sethostid** subroutine, **sethostname** subroutine.

Sockets Overview in *AIX Communications Programming Concepts*.

_getlong Subroutine

Purpose

Retrieves long byte quantities.

Library

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

Syntax

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

unsigned long _getlong (MessagePtr)
u_char *MessagePtr;
```

Description

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

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

Parameters

MessagePtr Specifies a pointer into the byte stream.

Return Values

The **_getlong** subroutine returns an unsigned long (32–bit) value.

Implementation Specifics

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

All applications containing the **_getlong** subroutine must be compiled with **_BSD** set to a specific value. Acceptable values are 43 and 44. In addition, all socket applications must include the BSD **libbsd.a** library or, in versions 4.2.1 and later, the Berkeley Thread Safe Library (**libbsd_r.a**).

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

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

Files

/etc/resolv.conf

Lists name server and domain names.

Related Information

The **dn_comp** subroutine, **dn_expand** subroutine, **_getshort** subroutine, **putlong** subroutine, **putshort** subroutine, **res_init** subroutine, **res_mkquery** subroutine, **res_query** subroutine, **res_search** subroutine, **res_send** subroutine.

Sockets Overview, and Understanding Domain Name Resolution in *AIX Communications Programming Concepts*.

getnetbyaddr Subroutine

Purpose

Gets network entry by address.

Library

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

Syntax

```
#include <netdb.h>

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

Description

Attention: Do not use the **getnetbyaddr** subroutine in a multithreaded environment. See the multithread alternative in the **getnetbyaddr_r** subroutine article.

Attention: Do not use the **getnetbyaddr** subroutine in a multithreaded environment.

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

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

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

Parameters

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

Return Values

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

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

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

Implementation Specifics

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

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

Files

/etc/networks

Contains official network names.

Related Information

The **endnetent** subroutine, **getnetbyname** subroutine, **getnetent** subroutine, **setnetent** subroutine.

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

getnetbyname Subroutine

Purpose

Gets network entry by name.

Library

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

Syntax

```
#include <netdb.h>

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

Description

Attention: Do not use the **getnetbyname** subroutine in a multithreaded environment. See the multithread alternative in the **getnetbyname_r** subroutine article.

Attention: Do not use the **getnetbyname** subroutine in a multithreaded environment.

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

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

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

Parameters

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

Return Values

Upon successful completion, the **getnetbyname** subroutine returns a pointer to a **netent** structure.

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

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

Implementation Specifics

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

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

Files

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

Related Information

The **endnetent** subroutine, **getnetbyaddr** subroutine, **getnetent** subroutine, **setnetent** subroutine.

Sockets Overview in *AIX Communications Programming Concepts*.

getnetent Subroutine

Purpose

Gets network entry.

Library

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

Syntax

```
#include <netdb.h>

struct netent *getnetent ( )
```

Description

Attention: Do not use the **getnetent** subroutine in a multithreaded environment. See the multithread alternative in the **getnetent_r** subroutine article.

Attention: Do not use the **getnetent** subroutine in a multithreaded environment.

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

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

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

Return Values

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

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

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

Implementation Specifics

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

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

Files

/etc/networks

Contains official network names.

Related Information

The **endnetent** subroutine, **getnetbyaddr** subroutine, **getnetbyname** subroutine, **setnetent** subroutine.

Sockets Overview in *AIX Communications Programming Concepts*.

getpeername Subroutine

Purpose

Gets the name of the peer socket.

Library

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

Syntax

```
#include <sys/types.h>
#include <sys/socket.h>
#include <sys/ndd_var.h> /*Needed for AF_NDD address family
only*/
#include <sys/atmsock.h> /*Needed for SOCK_CONN_DGRAM socket type
only*/

int getpeername (Socket, Name, NameLength)
int Socket;
struct sockaddr *Name;
size_t *NameLength;
```

Description

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

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

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

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

Parameters

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

Return Values

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

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

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

Error Codes

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

EBADF	The *Socket* parameter is not valid.
ENOTSOCK	The *Socket* parameter refers to a file, not a socket.
ENOTCONN	The socket is not connected.
ENOBUFS	Insufficient resources were available in the system to complete the call.
EFAULT	The *Address* parameter is not in a writable part of the user address space.

Examples

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

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

Implementation Specifics

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

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

Related Information

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

Sockets Overview in *AIX Communications Programming Concepts*.

getprotobyname Subroutine

Purpose

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

Library

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

Syntax

```
#include <netdb.h>

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

Description

Attention: Do not use the **getprotobyname** subroutine in a multithreaded environment. See the multithread alternative in the **getprotobyname_r** subroutine article.

Attention: Do not use the **getprotobyname** subroutine in a multithreaded environment.

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

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

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

Parameters

Name	Specifies the protocol name.

Return Values

Upon successful completion, the **getprotobyname** subroutine returns a pointer to a **protoent** structure.

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

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

Implementation Specifics

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

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

Related Information

The **endprotoent** subroutine, **getprotobynumber** subroutine, **getprotoent** subroutine, **setprotoent** subroutine.

Sockets Overview in *AIX Communications Programming Concepts*.

getprotobynumber Subroutine

Purpose

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

Library

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

Syntax

```
#include <netdb.h>

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

Description

Attention: Do not use the **getprotobynumber** subroutine in a multithreaded environment. See the multithread alternative in the **getprotobynumber_r** subroutine article.

Attention: Do not use the **getprotobynumber** subroutine in a multithreaded environment.

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

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

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

Parameters

Protocol	Specifies the protocol number.

Return Values

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

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

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

Implementation Specifics

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

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

Files

/etc/protocols

Contains protocol information.

Related Information

The **endprotoent** subroutine, **getprotobyname** subroutine, **getprotoent** subroutine, **setprotoent** subroutine.

Sockets Overview in *AIX Communications Programming Concepts*.

getprotoent Subroutine

Purpose

Gets protocol entry from the /**etc**/**protocols** file.

Library

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

Syntax

```
#include <netdb.h>

struct protoent *getprotoent ( )
```

Description

Attention: Do not use the **getprotoent** subroutine in a multithreaded environment. See the multithread alternative in the **getprotoent_r** subroutine article.

Attention: Do not use the **getprotoent** subroutine in a multithreaded environment.

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

The **getprotoent** subroutine opens and performs a sequential read of the /**etc**/**protocols** file. The **getprotoent** subroutine returns a pointer to a **protoent** structure, which contains the fields for a line of information in the /**etc**/**protocols** file. The **netdb.h** file defines the **protoent** structure.

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

Return Values

Upon successful completion, the **getprotoent** subroutine returns a pointer to a **protoent** structure.

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

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

Implementation Specifics

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

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

Files

/etc/protocols

Contains protocol information.

Related Information

The **endprotoent** subroutine, **getprotobyname** subroutine, **getprotobynumber** subroutine, **setprotoent** subroutine.

Sockets Overview in *AIX Communications Programming Concepts*.

getservbyname Subroutine

Purpose

Gets service entry by name.

Library

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

Syntax

```
#include <netdb.h>

struct servent *getservbyname (Name, Protocol)
char *Name, *Protocol;
```

Description

Attention: Do not use the **getservbyname** subroutine in a multithreaded environment. See the multithread alternative in the **getservbyname_r** subroutine article.

Attention: Do not use the **getservbyname** subroutine in a multithreaded environment.

The **getservbyname** subroutine retrieves an entry from the /**etc**/**services** file using the service name as a search key.

An application program can use the **getservbyname** subroutine to access a service, service aliases, the protocol for the service, and a protocol port number for the service.

The **getservbyname** subroutine searches the /**etc**/**services** file sequentially from the start of the file until it finds one of the following:

- Matching name and protocol number
- Matching name when the *Protocol* parameter is set to 0
- End of the file

Upon locating a matching name and protocol, the **getservbyname** subroutine returns a pointer to the **servent** structure, which contains fields for a line of information from the /**etc**/**services** file. The **netdb.h** file defines the **servent** structure and structure fields.

Use the **endservent** subroutine to close the /**etc**/**hosts** file.

Parameters

Name	Specifies the name of a service.
Protocol	Specifies a protocol for use with the specified service.

Return Values

The **getservbyname** subroutine returns a pointer to a **servent** structure when a successful match occurs. Entries in this structure are in network byte order.

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

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

Implementation Specifics

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

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

Files

/etc/services

Contains service names.

Related Information

The **endprotoent** subroutine, **endservent** subroutine, **getprotobyname** subroutine, **getprotobynumber** subroutine, **getprotoent** subroutine, **getservbyport** subroutine, **getservent** subroutine, **setprotoent** subroutine, **setservent** subroutine.

Sockets Overview, and Understanding Network Address Translation in *AIX Communications Programming Concepts*.

getservbyport Subroutine

Purpose

Gets service entry by port.

Library

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

Syntax

```
#include <netdb.h>

struct servent *getservbyport (Port, Protocol)
int Port;
char *Protocol;
```

Description

Attention: Do not use the **getservbyport** subroutine in a multithreaded environment. See the multithread alternative in the **getservbyport_r** subroutine article.

Attention: Do not use the **getservbyport** subroutine in a multithreaded environment.

The **getservbyport** subroutine retrieves an entry from the /**etc**/**services** file using a port number as a search key.

An application program can use the **getservbyport** subroutine to access a service, service aliases, the protocol for the service, and a protocol port number for the service.

The **getservbyport** subroutine searches the services file sequentially from the beginning of the file until it finds one of the following:

- Matching protocol and port number
- Matching protocol when the *Port* parameter value equals 0
- End of the file

Upon locating a matching protocol and port number or upon locating a matching protocol only if the *Port* parameter value equals 0, the **getservbyport** subroutine returns a pointer to a **servent** structure, which contains fields for a line of information in the /**etc**/**services** file. The **netdb.h** file defines the **servent** structure and structure fields.

Use the **endservent** subroutine to close the /**etc**/**services** file.

Parameters

Port	Specifies the port where a service resides.
Protocol	Specifies a protocol for use with the service.

Return Values

Upon successful completion, the **getservbyport** subroutine returns a pointer to a **servent** structure.

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

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

Implementation Specifics

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

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

Files

/etc/services

Contains service names.

Related Information

The **endprotoent** subroutine, **endservent** subroutine, **getprotobyname** subroutine, **getprotobynumber** subroutine, **getprotoent** subroutine, **getservbyname** subroutine, **getservent** subroutine, **setprotoent** subroutine, **setservent** subroutine.

Sockets Overview in *AIX Communications Programming Concepts*.

getservent Subroutine

Purpose

Gets services file entry.

Library

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

Syntax

```
#include <netdb.h>

truct servent *getservent ( )
```

Description

Attention: Do not use the **getservent** subroutine in a multithreaded environment. See the multithread alternative in the **getservent_r** subroutine article.

Attention: Do not use the **getservent** subroutine in a multithreaded environment.

The **getservent** subroutine opens and reads the next line of the /**etc**/**services** file.

An application program can use the **getservent** subroutine to retrieve information about network services and the protocol ports they use.

The **getservent** subroutine returns a pointer to a **servent** structure, which contains fields for a line of information from the /**etc**/**services** file. The **servent** structure is defined in the **netdb.h** file.

The /**etc**/**services** file remains open after a call by the **getservent** subroutine. To close the /**etc**/**services** file after each call, use the **setservent** subroutine. Otherwise, use the **endservent** subroutine to close the /**etc**/**services** file.

Return Values

The **getservent** subroutine returns a pointer to a **servent** structure when a successful match occurs.

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

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

Implementation Specifics

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

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

Files

/etc/services

Contains service names.

Related Information

The **endprotoent** subroutine, **endservent** subroutine, **getprotobyname** subroutine, **getprotobynumber** subroutine, **getprotoent** subroutine, **getservbyname** subroutine, **getservbyport** subroutine, **setprotoent** subroutine, **setservent** subroutine.

Sockets Overview, and Understanding Network Address Translation in *AIX Communications Programming Concepts*.

_getshort Subroutine

Purpose

Retrieves short byte quantities.

Library

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

Syntax

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

unsigned short getshort (MessagePtr)
u_char *MessagePtr;
```

Description

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

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

Parameters

MessagePtr Specifies a pointer into the byte stream.

Return Values

The **_getshort** subroutine returns an unsigned short (16–bit) value.

Implementation Specifics

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

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

Files

/**etc**/**resolv.conf**

Defines name server and domain names.

Related Information

The **dn_comp** subroutine, **dn_expand** subroutine, **_getlong** subroutine, **putlong** subroutine, **putshort** subroutine, **res_init** subroutine, **res_mkquery** subroutine, **res_send** subroutine.

Sockets Overview, and Understanding Domain Name Resolution in *AIX Communications Programming Concepts*.

getsockname Subroutine

Purpose

Gets the socket name.

Library

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

Syntax

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

int getsockname (Socket, Name, NameLength)
int Socket;
struct sockaddr *Name;
size_t *NameLength;
```

Description

The **getsockname** subroutine retrieves the locally bound address of the specified socket. The socket address represents a port number in the Internet domain and is stored in the **sockaddr** structure pointed to by the *Name* parameter. The **sys/socket.h** file defines the **sockaddr** data structure.

> **Note:** The **getsockname** subroutine does not perform operations on UNIX domain sockets.

A process created by another process can inherit open sockets. To use the inherited socket, the created process needs to identify their addresses. The **getsockname** subroutine allows a process to retrieve the local address bound to the specified socket.

A process can use the **getpeername** subroutine to determine the address of a destination socket in a socket connection.

Parameters

Socket	Specifies the socket for which the local address is desired.
Name	Points to the structure containing the local address of the specified socket.
NameLength	Specifies the size of the local address in bytes. Initializes the value pointed to by the *NameLength* parameter to indicate the amount of space pointed to by the *Name* parameter.

Return Values

Upon successful completion, a value of 0 is returned, and the *NameLength* parameter points to the size of the socket address.

If the **getsockname** subroutine is unsuccessful, the subroutine handler performs the following functions:

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

Error Codes

The **getsockname** subroutine is unsuccessful if any of the following errors occurs:

EBADF	The *Socket* parameter is not valid.
ENOTSOCK	The *Socket* parameter refers to a file, not a socket.
ENOBUFS	Insufficient resources are available in the system to complete the call.
EFAULT	The *Address* parameter is not in a writable part of the user address space.

Implementation Specifics

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

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

Related Information

The **accept** subroutine, **bind** subroutine, **getpeername** subroutine, **socket** subroutine.

Checking for Pending Connections Example Program, Reading Internet Datagrams Example Program, and Sockets Overview in *AIX Communications Programming Concepts*.

getsockopt Subroutine

Purpose

Gets options on sockets.

Library

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

Syntax

```
#include <sys/types.h>
#include <sys/socket.h>
#include <sys/atmsock.h> /*Needed for SOCK_CONN_DGRAM socket type
only*/

int getsockopt (Socket, Level, OptionName, OptionValue,
OptionLength)
int Socket, Level, OptionName;
void *OptionValue;
size_t *OptionLength;
```

Description

The **getsockopt** subroutine allows an application program to query socket options. The calling program specifies the name of the socket, the name of the option, and a place to store the requested information. The operating system gets the socket option information from its internal data structures and passes the requested information back to the calling program.

Options can exist at multiple protocol levels. They are always present at the uppermost socket level. When retrieving socket options, specify the level where the option resides and the name of the option.

Parameters

Socket Specifies the unique socket name.

Level Specifies the protocol level where the option resides. Options can be retrieved at the following levels:

Socket level Specifies the *Level* parameter as the **SOL_SOCKET** option.

Other levels Supplies the appropriate protocol number for the protocol controlling the option. For example, to indicate that an option will be interpreted by the TCP protocol, set the *Level* parameter to the protocol number of TCP, as defined in the **netinet/in.h** file.

OptionName Specifies a single option. The *OptionName* parameter and any specified options are passed uninterpreted to the appropriate protocol module for interpretation. The **sys/socket.h** file contains definitions for socket level options. The **netinet/tcp.h** file contains definitions for TCP protocol level options. Socket–level options can be enabled or disabled; they operate in a toggle fashion. The **sys/atmsock.h** file contains definitions for ATM protocol level options.

The following list defines socket protocol level options found in the **sys/socket.h** file:

SO_DEBUG Specifies the recording of debugging information. This option enables or disables debugging in the underlying protocol modules.

SO_BROADCAST
Specifies whether transmission of broadcast messages is supported. The option enables or disables broadcast support.

SO_CKSUMREV
Enables performance enhancements in the protocol layers. If the protocol supports this option, enabling causes the protocol to defer checksum verification until the user's data is moved into the user's buffer (on **recv**, **recvfrom**, **read**, or **recvmsg** thread). This can cause applications to be awakened when no data is available, in the case of a checksum error. In this case, EAGAIN is returned. Applications that set this option must handle the EAGAIN error code returned from a receive call.

SO_REUSEADDR
Specifies that the rules used in validating addresses supplied by a **bind** subroutine should allow reuse of a local port. A particular IP address can only be bound once to the same port. This option enables or disables reuse of local ports.

SO_REUSEADDR allows an application to explicitly deny subsequent **bind** subroutine to the port/address of the socket with **SO_REUSEADDR** set. This allows an application to block other applications from binding with the **bind** subroutine.

SO_REUSEPORT
Specifies that the rules used in validating addresses supplied by a **bind** subroutine should allow reuse of a local port/address combination. Each binding of the port/address combination must specify the **SO_REUSEPORT** socket option. This option enables or disables the reuse of local port/address combinations.

SO_KEEPALIVE
Monitors the activity of a connection by enabling or disabling the periodic transmission of ACK messages on a connected socket. The idle interval time can be designated using the TCP/IP **no** command. Broken connections are discussed in "Understanding Socket Types and Protocols" in *AIX Communications Programming Concepts*.

SO_DONTROUTE
Indicates outgoing messages should bypass the standard routing facilities. Does not apply routing on outgoing messages. Directs messages to the appropriate network interface according to the network portion of the destination address. This option enables or disables routing of outgoing messages.

SO_LINGER Lingers on a **close** subroutine if data is present. This option controls the action taken when an unsent messages queue exists for a socket, and a process performs a **close** subroutine on the socket.

If the **SO_LINGER** option is set, the system blocks the process during the **close** subroutine until it can transmit the data or until the time expires. If the **SO_LINGER** option is not specified, and a **close** subroutine is issued, the system handles the call in a way that allows the process to continue as quickly as possible.

The **sys/socket.h** file defines the **linger** structure that contains the **l_linger** value for specifying linger time interval. If linger time is set to anything but 0, the system tries to send any messages queued on the socket. Although a linger interval can be specified in seconds, the system ignores the specified time and makes repeated attempts to send unsent messages.

SO_OOBINLINE
Leaves received out–of–band data (data marked urgent) in line. This option enables or disables the receipt of out–of–band data.

SO_SNDBUF Retrieves buffer size information.

SO_RCVBUF Retrieves buffer size information.

SO_SNDLOWAT
Retrieves send buffer low–water mark information.

SO_RCVLOWAT
Retrieves receive buffer low–water mark information.

SO_SNDTIMEO
Retrieves time–out information. This option is settable, but currently not used.

SO_RCVTIMEO
Retrieves time–out information. This option is settable, but currently not used.

SO_ERROR Retrieves information about error status and clears.

The following list defines TCP protocol level options found in the **netinet/tcp.h** file:

TCP_RFC1323 Indicates whether RFC 1323 is enabled or disabled on the specified socket. A non–zero *OptionValue* returned by the **getsockopt** subroutine indicates the RFC is enabled.

TCP_NODELAY
Specifies whether TCP should follow the Nagle algorithm for deciding when to send data. By default TCP will follow the Nagle algorithm. To disable this behavior, applications can enable **TCP_NODELAY** to force TCP to always send data immediately. A non–zero *OptionValue* returned by the **getsockopt** subroutine indicates **TCP_NODELAY** is enabled. For example, **TCP_NODELAY** should be used when there is an appliciation using TCP for a request/response.

The following list defines ATM protocol level options found in the **sys/atmsock.h** file:

SO_ATM_PARM
Retrieves all ATM parameters. This socket option can be used instead of using individual sockets options described below. It uses the **connect_ie** structure defined in **sys/call_ie.h** file.

SO_ATM_AAL_PARM
Retrieves ATM AAL (Adaptation Layer) parameters. It uses the **aal_parm** structure defined in **sys/call_ie.h** file.

SO_ATM_TRAFFIC_DES
Retrieves ATM Traffic Descriptor values. It uses the **traffic_desc** structure defined in **sys/call_ie.h** file.

SO_ATM_BEARER
Retrieves ATM Bearer capability information. It uses the **bearer** structure defined in **sys/call_ie.h** file.

SO_ATM_BHLI Retrieves ATM Broadband High Layer Information. It uses the **bhli** structure defined in **sys/call_ie.h** file.

SO_ATM_BLLI Retrieves ATM Broadband Low Layer Information. It uses the **blli** structure defined in **sys/call_ie.h** file.

SO_ATM_QoS Retrieves ATM Ouality Of Service values. It uses the **qos_parm** structure defined in **sys/call_ie.h** file.

SO_ATM_TRANSIT_SEL
Retrieves ATM Transit Selector Carrier. It uses the **transit_sel** structure defined in **sys/call_ie.h** file.

SO_ATM_MAX_PEND
Retrieves the number of outstanding transmit buffers that are permitted before an error indication is returned to applications as a result of a transmit operation. This option is only valid for non best effort types of virtual circuits.

SO_ATM_CAUSE
Retrieves cause for the connection failure. It uses the **cause_t** structure defined in the **sys/call_ie.h** file.

OptionValue Specifies a pointer to the address of a buffer. The *OptionValue* parameter takes an integer parameter. The *OptionValue* parameter should be set to a nonzero value to enable a Boolean option or to a value of 0 to disable the option. The following options enable and disable in the same manner:

- **SO_DEBUG**
- **SO_REUSEADDR**
- **SO_KEEPALIVE**
- **SO_DONTROUTE**
- **SO_BROADCAST**
- **SO_OOBINLINE**
- **TCP_RFC1323**

OptionLength Specifies the length of the *OptionValue* parameter. The *OptionLength* parameter initially contains the size of the buffer pointed to by the *OptionValue* parameter. On return, the *OptionLength* parameter is modified to indicate the actual size of the value returned. If no option value is supplied or returned, the *OptionValue* parameter can be 0.

Options at other protocol levels vary in format and name.

IP level (**IPPROTO_IP** level) options are defined as follows:

IP_DONTFRAG	Get current **IP_DONTFRAG** option value.
IP_FINDPMTU	Get current PMTU value.
IP_PMTUAGE	Get current PMTU time out value.

In the case of TCP protocol sockets:

IP_DONTGRAG	Not supported.
IP_FINDPMTU	Get current PMTU value.
IP_PMTUAGE	Not supported.

Return Values

Upon successful completion, the **getsockopt** subroutine returns a value of 0.

If the **getsockopt** subroutine is unsuccessful, the subroutine handler performs the following actions:

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

Upon successful completion of the **IPPROTO_IP** option **IP_PMTUAGE** the returns are:

- OptionValue 0 if PMTU discovery is not enabled.
- OptionValue –1 if PMTU discovery is not complete.
- Positive non–zero OptionValue if PMTU is available.

Upon successful completion of TCP protocol sockets option **IP_FINDPMTU** the returns are:

- OptionValue 0 if PMTU discovery (tcp_pmtu_discover) is not enabled.
- OptionValue –1 if PMTU discovery is not complete/not available.
- Positive non–zero OptionValue if PMTU is available.

Error Codes

The **getsockopt** subroutine is unsuccessful if any of the following errors occurs:

EBADF	The *Socket* parameter is not valid.
ENOTSOCK	The *Socket* parameter refers to a file, not a socket.
ENOPROTOOPT	The option is unknown.
EFAULT	The address pointed to by the *OptionValue* parameter is not in a valid (writable) part of the process space, or the *OptionLength* parameter is not in a valid part of the process address space.

Examples

The following program fragment illustrates the use of the **getsockopt** subroutine to determine an existing socket type:

```
#include <sys/types.h>
#include <sys/socket.h>
int type, size;
size = sizeof(int);
if(getsockopt(s, SOL_SOCKET, SO_TYPE, (char*)&type,&size)<0){
.
.
.
}
```

Implementation Specifics

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

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

Related Information

The **no** command.

The **bind** subroutine, **close** subroutine, **endprotoent** subroutine, **getprotobynumber** subroutine, **getprotoent** subroutine, **setprotoent** subroutine, **setsockopt** subroutine, **socket** subroutine.

Sockets Overview, Understanding Socket Options, and Understanding Socket Types and Protocols in *AIX Communications Programming Concepts*.

htonl Subroutine

Purpose

Converts an unsigned long integer from host byte order to Internet network byte order.

Library

ISODE Library (**libisode.a**)

Syntax

```
#include <sys/types.h>
#include <netinet/in.h>

unsigned long htonl (HostLong)
unsigned long HostLong;
```

Description

The **htonl** subroutine converts an unsigned long (32–bit) integer from host byte order to Internet network byte order.

The Internet network requires addresses and ports in network standard byte order. Use the **htonl** subroutine to convert the host integer representation of addresses and ports to Internet network byte order.

The **htonl** subroutine is defined in the **net/nh.h** file as a macro.

Parameters

HostLong	Specifies a 32–bit integer in host byte order.

Return Values

The **htonl** subroutine returns a 32–bit integer in Internet network byte order (most significant byte first).

Implementation Specifics

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

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

Related Information

The **htons** subroutine, **ntohl** subroutine, **ntohs** subroutine.

Sockets Overview in *AIX Communications Programming Concepts*.

htons Subroutine

Purpose

Converts an unsigned short integer from host byte order to Internet network byte order.

Library

ISODE Library (**libisode.a**)

Syntax

```
#include <sys/types.h>
#include <netinet/in.h>

unsigned short htons (HostShort)
unsigned short HostShort;
```

Description

The **htons** subroutine converts an unsigned short (16–bit) integer from host byte order to Internet network byte order.

The Internet network requires ports and addresses in network standard byte order. Use the **htons** subroutine to convert addresses and ports from their host integer representation to network standard byte order.

The **htons** subroutine is defined in the **net/nh.h** file as a macro.

Parameters

HostShort	Specifies a 16–bit integer in host byte order that is a host address or port.

Return Values

The **htons** subroutine returns a 16–bit integer in Internet network byte order (most significant byte first).

Implementation Specifics

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

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

Related Information

The **htonl** subroutine, **ntohl** subroutine, **ntohs** subroutine.

Sockets Overview in *AIX Communications Programming Concepts*.

inet_addr Subroutine

Purpose

Converts Internet addresses to Internet numbers.

Library

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

Syntax

```
#include <sys/socket.h>
#include <sys/socketvar.h>
#include <netinet/in.h>
#include <arpa/inet.h>

unsigned long inet_addr (CharString)
char *CharString;
```

Description

The **inet_addr** subroutine converts an ASCII string containing a valid Internet address using dot notation into an Internet address number typed as an unsigned long value. An example of dot notation is 120.121.5.123. The **inet_addr** subroutine returns an error value if the Internet address notation in the ASCII string supplied by the application is not valid.

Note: Although they both convert Internet addresses in dot notation to Internet numbers, the **inet_addr** subroutine and **inet_network** process ASCII strings differently. When an application gives the **inet_addr** subroutine a string containing an Internet address value without a delimiter, the subroutine returns the logical product of the value represented by the string and 0xFFFFFFFF. For any other Internet address, if the value of the fields exceeds the previously defined limits, the **inet_addr** subroutine returns an error value of –1.

When an application gives the **inet_network** subroutine a string containing an Internet address value without a delimiter, the **inet_network** subroutine returns the logical product of the value represented by the string and 0xFF. For any other Internet address, the subroutine returns an error value of –1 if the value of the fields exceeds the previously defined limits.

Sample return values for each subroutine are as follows:

Application string	inet_addr returns	inet_netowork returns
0x1234567890abcdef 0x1234567890abcdef. 256.257.258.259	0x090abcdef 0xFFFFFFFF (= –1) 0xFFFFFFFF (= –1)	0x000000ef 0x0000ef00 0x00010203

The ASCII string for the **inet_addr** subroutine must conform to the following format:

```
string::= field | field delimited_field^1-3 | delimited_field^1-3
delimited_field::= delimiter field | delimiter
delimiter::= .
field::= 0X | 0x | 0Xhexadecimal* | 0x hexadecimal* | decimal* |
0 octal
hexadecimal::= decimal |a|b|c|d|e|f|A|B|C|D|E|F
decimal::= octal |8|9
octal::= 0|1|2|3|4|5|6|7
```

Notes:

1. `^n` indicates *n* repetitions of a pattern.

2. ^n-m indicates *n* to *m* repetitions of a pattern.
3. * indicates 0 or more repetitions of a pattern, up to environmental limits.
4. The Backus Naur form (BNF) description states the space character, if one is used. *Text* indicates text, not a BNF symbol.

The **inet_addr** subroutine requires an application to terminate the string with a null terminator (0x00) or a space (0x30). The string is considered invalid if the application does not end it with a null terminator or a space. The subroutine ignores characters trailing a space.

The following describes the restrictions on the field values for the **inet_addr** subroutine:

Format	Field Restrictions (in decimal)
a	*Value_a* < 4,294,967,296
a.b	*Value_a* < 256; *Value_b* < 16,777,216
a.b.c	*Value_a* < 256; *Value_b* < 256; *Value_c < 65536*
a.b.c.d	*Value_a* < 256; *Value_b* < 256; *Value_c < 256; Value_d < 256*

Applications that use the **inet_addr** subroutine can enter field values exceeding these restrictions. The subroutine accepts the least significant bits up to an integer in length, then checks whether the truncated value exceeds the maximum field value. For example, if an application enters a field value of 0x1234567890 and the system uses 16 bits per integer, then the **inet_addr** subroutine uses bits 0 –15. The subroutine returns 0x34567890.

Applications can omit field values between delimiters. The **inet_addr** subroutine interprets empty fields as 0.

Notes:

1. The **inet_addr** subroutine does not check the pointer to the ASCII string. The user must ensure the validity of the address in the ASCII string.
2. The application must verify that the network and host IDs for the Internet address conform to either a Class A, B, or C Internet address. The **inet_attr** subroutine processes any other number as a Class C address.

Parameters

CharString	Represents a string of characters in the Internet address form.

Return Values

For valid input strings, the **inet_addr** subroutine returns an unsigned long value comprised of the bit patterns of the input fields concatenated together. The subroutine places the first pattern in the most significant position and appends any subsequent patterns to the next most significant positions.

The **inet_addr** subroutine returns an error value of –1 for invalid strings.

Note: An Internet address with a dot notation value of 255.255.255.255 or its equivalent in a different base format causes the **inet_addr** subroutine to return an unsigned long value of 4294967295. This value is identical to the unsigned representation of the error value. Otherwise, the **inet_addr** subroutine considers 255.255.255.255 a valid Internet address.

Implementation Specifics

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

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

Files

/etc/hosts

Contains host names.

/etc/networks

Contains network names.

Related Information

The **endhostent** subroutine, **endnetent** subroutine, **gethostbyaddr** subroutine, **gethostbyname** subroutine, **getnetbyaddr** subroutine, **getnetbyname** subroutine, **getnetent** subroutine, **inet_lnaof** subroutine, **inet_makeaddr** subroutine, **inet_netof** subroutine, **inet_network** subroutine, **inet_ntoa** subroutine, **sethostent** subroutine, **setnetent** subroutine.

Sockets Overview and Understanding Network Address Translation in *AIX Communications Programming Concepts*.

inet_lnaof Subroutine

Purpose

Returns the host ID of an Internet address.

Library

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

Syntax

```
#include <sys/socket.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>

int inet_lnaof (InternetAddr)
struct in_addr InternetAddr;
```

Description

The **inet_lnaof** subroutine masks off the host ID of an Internet address based on the Internet address class. The calling application must enter the Internet address as an unsigned long value.

Note: The application must verify that the network and host IDs for the Internet address conform to either a Class A, B, or C Internet address. The **inet_lnaof** subroutine processes any other number as a Class C address.

Parameters

InternetAddr	Specifies the Internet address to separate.

Return Values

The return values of the **inet_lnaof** subroutine depend on the class of Internet address the application provides:

Class A	The logical product of the Internet address and 0x00FFFFFF.
Class B	The logical product of the Internet address and 0x0000FFFF.
Class C	The logical product of the Internet address and 0x000000FF.

Implementation Specifics

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

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

Files

/etc/hosts

Contains host names.

Related Information

The **endhostent** subroutine, **endnetent** subroutine, **gethostbyaddr** subroutine, **gethostbyname** subroutine, **getnetbyaddr** subroutine, **getnetbyname** subroutine, **getnetent** subroutine, **inet_addr** subroutine, **inet_makeaddr** subroutine, **inet_netof**

subroutine, **inet_network** subroutine, **inet_ntoa** subroutine, **sethostent** subroutine. **setnetent** subroutine.

Sockets Overview and Understanding Network Address Translation in *AIX Communications Programming Concepts*.

inet_makeaddr Subroutine

Purpose

Returns a structure containing an Internet address based on a network ID and host ID provided by the application.

Library

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

Syntax

```
#include <sys/socket.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>

struct in_addr inet_makeaddr (Net, LocalNetAddr)
int Net, LocalNetAddr;
```

Description

The **inet_makeaddr** subroutine forms an Internet address from the network ID and Host ID provided by the application (as integer types). If the application provides a Class A network ID, the **inet_makeaddr** subroutine forms the Internet address using the net ID in the highest–order byte and the logical product of the host ID and 0x00FFFFFF in the 3 lowest–order bytes. If the application provides a Class B network ID, the **inet_makeaddr** subroutine forms the Internet address using the net ID in the two highest–order bytes and the logical product of the host ID and 0x0000FFFF in the lowest two ordered bytes. If the application does not provide either a Class A or Class B network ID, the **inet_makeaddr** subroutine forms the Internet address using the network ID in the 3 highest–order bytes and the logical product of the host ID and 0x0000FFFF in the lowest–ordered byte.

The **inet_makeaddr** subroutine ensures that the Internet address format conforms to network order, with the first byte representing the high–order byte. The **inet_makeaddr** subroutine stores the Internet address in the structure as an unsigned long value.

The application must verify that the network ID and host ID for the Internet address conform to class A, B, or C. The **inet_makeaddr** subroutine processes any nonconforming number as a Class C address.

The **inet_makeaddr** subroutine expects the **in_addr** structure to contain only the Internet address field. If the application defines the **in_addr** structure otherwise, then the value returned in **in_addr** by the **inet_makeaddr** subroutine is undefined.

Parameters

Net	Contains an Internet network number.
LocalNetAddr	Contains a local network address.

Return Values

Upon successful completion, the **inet_makeaddr** subroutine returns a structure containing an Internet address.

If the **inet_makeaddr** subroutine is unsuccessful, the subroutine returns a –1.

Implementation Specifics

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

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

Files

/etc/hosts

Contains host names.

Related Information

The **endhostent** subroutine, **endnetent** subroutine, **gethostbyaddr** subroutine, **gethostbyname** subroutine, **getnetbyaddr** subroutine, **getnetbyname** subroutine, **getnetent** subroutine, **inet_addr** subroutine, **inet_lnaof** subroutine, **inet_netof** subroutine, **inet_network** subroutine, **inet_ntoa** subroutine, **sethostent** subroutine, **setnetent** subroutine.

Sockets Overview and Understanding Network Address Translation in *AIX Communications Programming Concepts*.

inet_netof Subroutine

Purpose

Returns the network id of the given Internet address.

Library

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

Syntax

```
#include <sys/socket.h>
#include <sys/socketvar.h>
#include <netinet/in.h>
#include <arpa/inet.h>

int inet_netof (InternetAddr)
struct in_addr InternetAddr;
```

Description

The **inet_netof** subroutine returns the network number from the specified Internet address number typed as unsigned long value. The **inet_netof** subroutine masks off the network number and the host number from the Internet address based on the Internet address class.

Note: The application assumes responsibility for verifying that the network number and the host number for the Internet address conforms to a class A or B or C Internet address. The **inet_netof** subroutine processes any other number as a class C address.

Parameters

InternetAddr

Specifies the Internet address to separate.

Return Values

Upon successful completion, the **inet_netof** subroutine returns a network number from the specified long value representing the Internet address. If the application gives a class A Internet address, the **inet_lnoaf** subroutine returns the logical product of the Internet address and `0xFF000000`. If the application gives a class B Internet address, the **inet_lnoaf** subroutine returns the logical product of the Internet address and `0xFFFF0000`. If the application does not give a class A or B Internet address, the **inet_lnoaf** subroutine returns the logical product of the Internet address and `0xFFFFFF00`.

Implementation Specifics

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

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

Files

/etc/hosts

Contains host names.

/etc/networks

Contains network names.

Related Information

The **endhostent** subroutine, **endnetent** subroutine, **gethostbyaddr** subroutine, **gethostbyname** subroutine, **getnetbyaddr** subroutine, **getnetbyname** subroutine, **getnetent** subroutine, **inet_addr** subroutine, **inet_lnaof** subroutine, **inet_makeaddr** subroutine, **inet_network** subroutine, **inet_ntoa** subroutine, **sethostent** subroutine, **setnetent** subroutine.

Sockets Overview and Understanding Network Address Translation in *AIX Communications Programming Concepts*.

inet_network Subroutine

Purpose

Converts an ASCII string containing an Internet network addressee in . (dot) notation to an Internet address number.

Library

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

Syntax

```
#include <sys/socket.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>

unsigned long inet_network (CharString)
char *CharString;
```

Description

The **inet_network** subroutine converts an ASCII string containing a valid Internet address using . (dot) notation (such as, `120.121.122.123`) to an Internet address number formatted as an unsigned long value. The **inet_network** subroutine returns an error value if the application does not provide an ASCII string containing a valid Internet address using . notation.

The input ASCII string must represent a valid Internet address number, as described in ”TCP/IP Addressing” in *AIX 4.3 System Management Guide: Communications and Networks*. The input string must be terminated with a null terminator (0x00) or a space (0x30). The **inet_network** subroutine ignores characters that follow the terminating character.

The input string can express an Internet address number in decimal, hexadecimal, or octal format. In hexadecimal format, the string must begin with `0x`. The string must begin with `0` to indicate octal format. In decimal format, the string requires no prefix.

Each octet of the input string must be delimited from another by a period. The application can omit values between delimiters. The **inet_network** subroutine interprets missing values as 0.

The following examples show valid strings and their output values in both decimal and hexadecimal notation:

Examples of valid strings		
Input String	**Output Value (in decimal)**	**Output Value (in hex)**
...1	1	0x00000001
.1..	65536	0x00010000
1	256	0x00000100
0xFFFFFFFF	255	0x000000FF
1.	16777216	0x01000000
1.2.3.4	16909060	0x01020304
0x01.0X2.03.004	16909060	0x01020304
1.2. 3.4	16777218	0x01000002
9999.1.1.1	251724033	0x0F010101

The following examples show invalid input strings and the reasons they are not valid:

Examples of invalid strings	
Input String	**Reason**
1.2.3.4.5	Excessive fields.
1.2.3.4.	Excessive delimiters (and therefore fields).
1,2	Bad delimiter.
1p	String not terminated by null terminator nor space.
{empty string}	No field or delimiter present.

Typically, the value of each octet of an Internet address cannot exceed 246. The **inet_network** subroutine can accept larger values, but it uses only the eight least significant bits for each field value. For example, if an application passes `0x1234567890.0xabcdef`, the **inet_network** subroutine returns `37103` (`0x000090EF`).

The application must verify that the network ID and host ID for the Internet address conform to class A, class B, or class C. The **inet_makeaddr** subroutine processes any nonconforming number as a class C address.

The **inet_network** subroutine does not check the pointer to the ASCII input string. The application must verify the validity of the address of the string.

Parameters

CharString Represents a string of characters in the Internet address form.

Return Values

For valid input strings, the **inet_network** subroutine returns an unsigned long value that comprises the bit patterns of the input fields concatenated together. The **inet_network** subroutine places the first pattern in the leftmost (most significant) position and appends subsequent patterns if they exist.

For invalid input strings, the **inet_network** subroutine returns a value of –1.

Implementation Specifics

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

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

Files

/etc/hosts

Contains host names.

/etc/networks

Contains network names.

Related Information

The **endhostent** subroutine, **endnetent** subroutine, **gethostbyaddr** subroutine, **gethostbyname** subroutine, **getnetbyaddr** subroutine, **getnetbyname** subroutine, **getnetent** subroutine, **inet_addr** subroutine, **inet_lnaof** subroutine, **inet_makeaddr** subroutine, **inet_netof** subroutine, **inet_ntoa** subroutine, **sethostent** subroutine, **setnetent** subroutine.

Sockets Overview and Understanding Network Address Translation in *AIX Communications Programming Concepts*.

inet_ntoa Subroutine

Purpose

Converts an Internet address into an ASCII string.

Library

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

Syntax

```
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>

char *inet_ntoa (InternetAddr)
struct in_addr InternetAddr;
```

Description

The **inet_ntoa** subroutine takes an Internet address and returns an ASCII string representing the Internet address in dot notation. All Internet addresses are returned in network order, with the first byte being the high–order byte.

Use C language integers when specifying each part of a dot notation.

Parameters

InternetAddr Contains the Internet address to be converted to ASCII.

Return Values

Upon successful completion, the **inet_ntoa** subroutine returns an Internet address.

If the **inet_ntoa** subroutine is unsuccessful, the subroutine returns a –1.

Implementation Specifics

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

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

Files

/etc/hosts

Contains host names.

/etc/networks

Contains network names.

Related Information

The **endhostent** subroutine, **endnetent** subroutine, **gethostbyaddr** subroutine, **gethostbyname** subroutine, **getnetbyaddr** subroutine, **getnetbyname** subroutine, **getnetent** subroutine, **inet_addr** subroutine, **inet_lnaof** subroutine, **inet_makeaddr** subroutine, **inet_network** subroutine, **sethostent** subroutine, **setnetent** subroutine.

Sockets Overview and Understanding Network Address Translation in *AIX Communications Programming Concepts*.

innetgr, getnetgrent, setnetgrent, or endnetgrent Subroutine

Purpose

Handles the group network entries.

Library

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

Syntax

```
#include <netdb.h>

innetgr (NetGroup, Machine, User, Domain)
char *NetGroup, *Machine, *User, *Domain;

getnetgrent (MachinePointer, UserPointer, DomainPointer)
char **MachinePointer, **UserPointer, **DomainPointer;

void setnetgrent (NetGroup)
char *NetGroup

void endnetgrent ()
```

Description

Attention: Do not use the **innetgr** subroutine in a multithreaded environment. See the multithread alternative in the **innetgr_r** subroutine article.

Attention: Do not use the **innetgr** subroutine in a multithreaded environment.

The **innetgr** subroutine returns *1* or *0*, depending on if **netgroup** contains the *machine*, *user*, *domain* triple as a member. Any of these three strings; *machine*, *user*, or *domain*, can be NULL, in which case it signifies a wild card.

getnetgrent() returns the next member of a network group. After the call, *machinepointer* will contain a pointer to a string containing the name of the machine part of the network group member, and similarly for *userpointer* and *domainpointer*. If any of *machinepointer*, *userpointer*, or *domainpointer* is returned as a NULL pointer, it signifies a wild card. **getnetgrent()** uses malloc to allocate space for the name. This space is released when **endnetgrent()** is called. **getnetgrent** returns *1* if it succeeded in obtaining another member of the network group or 0 when it has reached the end of the group.

setnetgrent() establishes the network group from which **getnetgrent()** will obtain members, and also restarts calls to **getnetgrent()** from the beginnning of the list. If the previous **setnetgrent()** call was to a different netowrk group, an **endnetgrent()** call is implied. **endnetgrent()** frees the space allocated during the **getnetgrent()** calls.

Parameters

Domain	Specifies the domain.
DomainPointer	Points to the string containing *domain* part of the network group.
Machine	Specifies the machine.
MachinePointer	Points to the string containing machine part of the network group.
NetGroup	Points to a network group.
User	Specifies a user.
UserPointer	Points to the string containing user part of the network group.

Return Values

1	Indicates that the subroutine was successful in obtaining a member.
0	Indicates that the subroutine was not successful in obtaining a member.

Implementation Specifics

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

Files

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

Related Information

Sockets Overview in *AIX Communications Programming Concepts*.

isinet_addr Subroutine

Purpose

Determines if the given ASCII string contains an Internet address using dot notation.

Library

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

Syntax

```
#include <sys/types.h>
#include <netinet/in.h>

u_long isinet_addr (name)
char *name;
```

Description

The **isinet_addr** subroutine determines if the given ASCII string contains an Internet address using dot notation (for example, ”120.121.122.123”). The **isaddr_inet** subroutine considers Internet address strings as a valid string, and considers any other string type as an invalid strings.

The **isinet_addr** subrountine expects the ASCII string to conform to the following format:

```
string ::= field | field delimited_field^1-3
delimited_field ::= delimiter field
delimiter ::= .
field ::= 0 X | 0 x | 0 X hexadecimal* | 0 x hexadecimal* |
decimal* | 0 octal*
hexadecimal ::= decimal | a | b | c | d | e | f | A | B | C | D |
E | F
decimal ::= octal | 8 | 9
octal ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7
```

A^*n*	Indicates *n* repetitions of pattern A.
A^*n*–*m*	Indicates *n* to *m* repetitions of pattern A.
A*	Indicates zero or more repetitions of pattern A, up to environmental limits.

The BNF description explicitly states the space character (’ ’), if used.

{*text*}	Indicates *text*, not a BNF symbol.

The **isinet_addr** subrountine allows the application to terminate the string with a null terminator (0x00) or a space (0x30). It ignores characters trailing the space character and considers the string invalid if the application does not terminate the string with a null terminator (0x00) or space (0x30).

The following describes the restrictions on the field values:

Address Format	Field Restrictions (values in decimal base)
a	a < 4294967296.
a.b	a < 256; b < 16777216.
a.b.c	a < 256; b < 256; c < 16777216.
a.b.c.d	a < 256; b < 2^8; c < 256; d < 256.

The **isinet_addr** subrountine applications can enter field values exceeding the field value restrictions specified previously; **isinet_addr** accepts the least significant bits up to an integer in length. The **isinet_addr** subroutine still checks to see if the truncated value exceeds the maximum field value. For example, if an application gives the string `0.0;0;0xFF00000001` then **isinet_addr** interprets the string as `0.0.0.0x00000001` and considers the string as valid.

isinet_addr applications cannot omit field values between delimiters and considers a string with successive periods as invalid.

Examples of valid strings:

Input String	Comment
1	**isinet_addr** uses **a** format.
1.2	**isinet_addr** uses **a.b** format.
1.2.3.4	**isinet_addr** uses **a.b.c.d** format.
0x01.0X2.03.004	**isinet_addr** uses **a.b.c.d** format.
1.2 3.4	**isinet_addr** uses **a.b** format; and ignores ”3.4”.

Examples of invalid strings:

Input String	Reason
...	No explicit field values specified.
1.2.3.4.5	Excessive fields.
1.2.3.4.	Excessive delimiters and fields.
1,2	Bad delimiter.
1p	String not terminated by null terminator nor space.
{empty string}	No field or delimiter present.
9999.1.1.1	Value for field a exceeds limit.

Notes:

1. The **isinet_addr** subroutine does not check the pointer to the ASCII string; the user takes responsibility for ensuring validity of the address of the ASCII string.
2. The application assumes responsibility for verifying that the network number and host number for the Internet address conforms to a class A or B or C Internet address; any other string is processed as a class C address.

Parameters

name	Address of ASCII string buffer.

Return Values

The **isinet_addr** subroutine returns 1 for valid input strings and 0 for invalid input strings. **isinet_addr** returns the value as an unsigned long type.

Implementation Specifics

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

All applications using **isinet_addr** must compile with _BSD defined. Also, all socket applications must include the BSD library **libbsd** when applicable.

Files

#include <ctype.h>

#include <sys/types.h>

Related Information

Internet address conversion subroutines: **inet_addr** subroutine, **inet_lnaof** subroutine, **inet_makeaddr** subroutine, **inet_netof** subroutine, **inet_network** subroutine, **inet_ntoa** subroutine.

Host information retrieval subroutines: **endhostent** subroutine, **gethostbyaddr** subroutine, **gethostbyname** subroutine, **sethostent** subroutine.

Network information retrieval subroutines: **getnetbyaddr** subroutine, **getnetbyname** subroutine, **getnetent** subroutine, **setnetent** subroutine.

listen Subroutine

Purpose

Listens for socket connections and limits the backlog of incoming connections.

Library

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

Syntax

```
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>

int listen (Socket, Backlog)
int Socket, Backlog;
```

Description

The **listen** subroutine performs the following activities:

1. Identifies the socket that receives the connections.
2. Marks the socket as accepting connections.
3. Limits the number of outstanding connection requests in the system queue.

The maximum queue length that the **listen** subroutine can specify is 10. The maximum queue length is indicated by the **SOMAXCONN** value in the /**usr**/**include**/**sys**/**socket.h** file.

Parameters

Socket	Specifies the unique name for the socket.
Backlog	Specifies the maximum number of outstanding connection requests.

Return Values

Upon successful completion, the **listen** subroutine returns a value 0.

If the **listen** subroutine is unsuccessful, the subroutine handler performs the following functions:

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

Error Codes

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

EBADF	The *Socket* parameter is not valid.
ECONNREFUSED	The host refused service, usually due to a server process missing at the requested name or the request exceeding the backlog amount.
ENOTSOCK	The *Socket* parameter refers to a file, not a socket.
EOPNOTSUPP	The referenced socket is not a type that supports the **listen** subroutine.

Examples

The following program fragment illustrates the use of the **listen** subroutine with 5 as the maximum number of outstanding connections which may be queued awaiting acceptance by the server process.

```
listen(s,5)
```

Implementation Specifics

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

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

Related Information

The **accept** subroutine, **connect** subroutine, **socket** subroutine.

Accepting Internet Stream Connections Example Program, Sockets Overview, Understanding Socket Connections in *AIX Communications Programming Concepts*.

ntohl Subroutine

Purpose

Converts an unsigned long integer from Internet network standard byte order to host byte order.

Library

ISODE Library (**libisode.a**)

Syntax

```
#include <sys/types.h>
#include <netinet/in.h>

unsigned long ntohl (NetLong)
unsigned long NetLong;
```

Description

The **ntohl** subroutine converts an unsigned long (32–bit) integer from Internet network standard byte order to host byte order.

Receiving hosts require addresses and ports in host byte order. Use the **ntohl** subroutine to convert Internet addresses and ports to the host integer representation.

The **ntohl** subroutine is defined in the **net/nh.h** file as a macro.

Parameters

NetLong	Requires a 32–bit integer in network byte order.

Return Values

The **ntohl** subroutine returns a 32–bit integer in host byte order.

Implementation Specifics

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

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

Related Information

The **endhostent** subroutine, **endservent** subroutine, **gethostbyaddr** subroutine, **gethostbyname** subroutine, **getservbyname** subroutine, **getservbyport** subroutine, **getservent** subroutine, **htonl** subroutine, **htons** subroutine, **ntohs** subroutine, **sethostent** subroutine, **setservent** subroutine.

Sockets Overview in *AIX Communications Programming Concepts*.

ntohs Subroutine

Purpose

Converts an unsigned short integer from Internet network byte order to host byte order.

Library

ISODE Library (**libisode.a**)

Syntax

```
#include <sys/types.h>
#include <netinet/in.h>

unsigned short ntohs (NetShort)
unsigned short NetShort;
```

Description

The **ntohs** subroutine converts an unsigned short (16–bit) integer from Internet network byte order to the host byte order.

Receiving hosts require Internet addresses and ports in host byte order. Use the **ntohs** subroutine to convert Internet addresses and ports to the host integer representation.

The **ntohs** subroutine is defined in the **net/nh.h** file as a macro.

Parameters

NetShort	Requires a 16–bit integer in network standard byte order.

Return Values

The **ntohs** subroutine returns the supplied integer in host byte order.

Implementation Specifics

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

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

Related Information

The **endhostent** subroutine, **endservent** subroutine, **gethostbyaddr** subroutine, **gethostbyname** subroutine, **getservbyname** subroutine, **getservbyport** subroutine, **getservent** subroutine, **htonl** subroutine, **htons** subroutine, **ntohl** subroutine, **sethostent** subroutine, **setservent** subroutine.

Sockets Overview in *AIX Communications Programming Concepts*.

_putlong Subroutine

Purpose

Places long byte quantities into the byte stream.

Library

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

Syntax

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

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

Description

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

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

Parameters

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

Implementation Specifics

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

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

Files

/etc/resolv.conf

Lists the name server and domain name.

Related Information

The **dn_comp** subroutine, **dn_expand** subroutine, **_getlong** subroutine, **_getshort** subroutine, **putshort** subroutine, **res_init** subroutine, **res_mkquery** subroutine, **res_query** subroutine, **res_search** subroutine, **res_send** subroutine.

Sockets Overview and Understanding Domain Name Resolution in *AIX Communications Programming Concepts*.

_putshort Subroutine

Purpose

Places short byte quantities into the byte stream.

Library

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

Syntax

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

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

Description

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

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

Parameters

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

Implementation Specifics

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

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

Files

/etc/resolv.conf

Lists the name server and domain name.

Related Information

The **dn_comp** subroutine, **dn_expand** subroutine, **_getlong** subroutine, **_getshort** subroutine, **putlong** subroutine, **res_init** subroutine, **res_mkquery** subroutine, **res_send** subroutine.

Sockets Overview and Understanding Domain Name Resolution in *AIX Communications Programming Concepts*.

rcmd Subroutine

Purpose

Allows execution of commands on a remote host.

Library

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

Syntax

```
int rcmd (Host,
Port, LocalUser, RemoteUser,
Command, ErrFileDesc)
char **Host;
u_short Port;
char *LocalUser;
char *RemoteUser;
char *Command;
int *ErrFileDesc;
```

Description

The **rcmd** subroutine allows execution of certain commands on a remote host that supports **rshd**, **rlogin**, and **rpc** among others.

Only processes with an effective user ID of root user can use the **rcmd** subroutine. An authentication scheme based on remote port numbers is used to verify permissions. Ports in the range between 0 and 1023 can only be used by a root user.

The **rcmd** subroutine looks up a host by way of the name server or if the local name server isn't running, in the /**etc**/**hosts** file.

If the connection succeeds, a socket in the Internet domain of type **SOCK_STREAM** is returned to the calling process and given to the remote command as standard input (**stdin**) and standard output (**stdout**).

Always specify the *Host* parameter. If the local domain and remote domain are the same, specifying the domain parts is optional.

Parameters

Host	Specifies the name of a remote host that is listed in the /**etc**/**hosts** file. If the specified name of the host is not found in this file, the **rcmd** subroutine is unsuccessful.
Port	Specifies the well–known port to use for the connection. The /**etc**/**services** file contains the DARPA Internet services, their ports, and socket types.
LocalUser and *RemoteUser*	Points to user names that are valid at the local and remote host, respectively. Any valid user name can be given.

Command Specifies the name of the command to be started at the remote host.

ErrFileDesc

Specifies an integer controlling the set up of communication channels. Integer options are as follows:

Non–zero Indicates an auxiliary channel to a control process is set up, and the *ErrFileDesc* parameter points to the file descriptor for the channel. The control process provides diagnostic output from the remote command on this channel and also accepts bytes as signal numbers to be forwarded to the process group of the command.

0 Indicates the standard error (**stderr**) of the remote command is the same as standard output (**stdout**). No provision is made for sending arbitrary signals to the remote process. However, it is possible to send out–of–band data to the remote command.

Return Values

Upon successful completion, the **rcmd** subroutine returns a valid socket descriptor.

Upon unsuccessful completion, the **rcmd** subroutine returns a value of –1. The subroutine returns a –1, if the effective user ID of the calling process is not root user or if the subroutine is unsuccessful to resolve the host.

Implementation Specifics

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

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

Files

/etc/services	Contains the service names, ports, and socket type.
/etc/hosts	Contains host names and their addresses for hosts in a network.
/etc/resolv.conf	Contains the name server and domain name.

Related Information

The **rlogind** command, **rshd** command.

The **named** daemon.

The **gethostname** subroutine, **rresvport** subroutine, **ruserok** subroutine, **sethostname** subroutine.

Sockets Overview in *AIX Communications Programming Concepts*.

recv Subroutine

Purpose

Receives messages from connected sockets.

Library

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

Syntax

```
#include <sys/types.h>
#include <sys/socket.h>
#include <sys/socketvar.h>

int recv (Socket,
Buffer, Length, Flags)
int Socket;
void *Buffer;
size_t Length;
int Flags;
```

Description

The **recv** subroutine receives messages from a connected socket. The **recvfrom** and **recvmsg** subroutines receive messages from both connected and unconnected sockets. However, they are usually used for unconnected sockets only.

The **recv** subroutine returns the length of the message. If a message is too long to fit in the supplied buffer, excess bytes may be truncated depending on the type of socket that issued the message.

If no messages are available at the socket, the **recv** subroutine waits for a message to arrive, unless the socket is nonblocking. If a socket is nonblocking, the system returns an error.

Use the **select** subroutine to determine when more data arrives.

Parameters

Socket Specifies the socket descriptor.

Buffer Specifies an address where the message should be placed.

Length Specifies the size of the *Buffer* parameter.

Flags

Points to a value controlling the message reception. The /**usr**/**include**/**sys**/**socket.h** file defines the *Flags* parameter. The argument to receive a call is formed by logically ORing one or more of the following values:

MSG_PEEK Peeks at incoming message. The data is treated as unread, and the next **recv** subroutine still returns this data.

MSG_OOB Processes out–of–band data.

Return Values

Upon successful completion, the **recv** subroutine returns the length of the message in bytes.

If the **recv** subroutine is unsuccessful, the subroutine handler performs the following functions:

- Returns a value of –1 to the calling program.
- Returns a 0 if the connection disconnects.
- Moves an error code, indicating the specific error, into the **errno** global variable.

Error Codes

The **recv** subroutine is unsuccessful if any of the following errors occurs:

EBADF	The *Socket* parameter is not valid.
ENOTSOCK	The *Socket* parameter refers to a file, not a socket.
EWOULDBLOCK	The socket is marked nonblocking, and no connections are present to be accepted.
EINTR	A signal interrupted the **recv** subroutine before any data was available.
EFAULT	The data was directed to be received into a nonexistent or protected part of the process address space. The *Buffer* parameter is not valid.

Implementation Specifics

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

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

Related Information

The **fgets** subroutine, **fputs** subroutine, **read** subroutine, **recvfrom** subroutine, **recvmsg** subroutine, **select** subroutine, **send** subroutine, **sendmsg** subroutine, **sendto** subroutine, **shutdown** subroutine, **socket** subroutine, **write** subroutine.

Sockets Overview and Understanding Socket Data Transfer in *AIX Communications Programming Concepts*.

recvfrom Subroutine

Purpose

Receives messages from sockets.

Library

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

Syntax

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

int recvfrom
(Socket, Buffer, Length,
Flags, From, FromLength)
int Socket;
char *Buffer;
int Length, Flags;
struct sockaddr *From;
int *FromLength;
```

Description

The **recvfrom** subroutine allows an application program to receive messages from unconnected sockets. The **recvfrom** subroutine is normally applied to unconnected sockets as it includes parameters that allow the calling program to specify the source point of the data to be received.

To return the source address of the message, specify a nonnull value for the *From* parameter. The *FromLength* parameter is a value–result parameter, initialized to the size of the buffer associated with the *From* parameter. On return, the **recvfrom** subroutine modifies the *FromLength* parameter to indicate the actual size of the stored address. The **recvfrom** subroutine returns the length of the message. If a message is too long to fit in the supplied buffer, excess bytes may be truncated depending on the type of socket that issued the message.

If no messages are available at the socket, the **recvfrom** subroutine waits for a message to arrive, unless the socket is nonblocking. If the socket is nonblocking, the system returns an error.

Parameters

Socket	Specifies the socket descriptor.
Buffer	Specifies an address where the message should be placed.
Length	Specifies the size of the *Buffer* parameter.
Flags	Points to a value controlling the message reception. The argument to receive a call is formed by logically ORing one or more of the values shown in the following list: **MSG_PEEK** Peeks at incoming message. **MSG_OOB** Processes out–of–band data.
From	Points to a socket structure, filled in with the source's address.
FromLength	Specifies the length of the sender's or source's address.

Return Values

If the **recvfrom** subroutine is successful, the subroutine returns the length of the message in bytes.

If the call is unsuccessful, the subroutine handler performs the following functions:

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

Error Codes

The **recvfrom** subroutine is unsuccessful if any of the following errors occurs:

EBADF	The *Socket* parameter is not valid.
ENOTSOCK	The *Socket* parameter refers to a file, not a socket.
EWOULDBLOCK	The socket is marked nonblocking, and no connections are present to be accepted.
EFAULT	The data was directed to be received into a nonexistent or protected part of the process address space. The buffer is not valid.

Implementation Specifics

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

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

Related Information

The **fgets** subroutine, **fputs** subroutine, **read** subroutine, **recv** subroutine, **recvmsg** subroutine, **select** subroutine, **send** subroutine, **sendmsg** subroutine, **sendto** subroutine, **shutdown** subroutine, **socket** subroutine, **write** subroutine.

Sockets Overview and Understanding Socket Data Transfer in *AIX Communications Programming Concepts*.

recvmsg Subroutine

Purpose

Receives a message from any socket.

Library

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

Syntax

```
#include <sys/types.h>
#include <sys/socket.h>
#include <sys/socketvar.h>

int recvmsg (Socket, Message, Flags)
int Socket;
struct msghdr Message [ ];
int Flags;
```

Description

The **recvmsg** subroutine receives messages from unconnected or connected sockets. The **recvmsg** subroutine returns the length of the message. If a message is too long to fit in the supplied buffer, excess bytes may be truncated depending on the type of socket that issued the message.

If no messages are available at the socket, the **recvmsg** subroutine waits for a message to arrive. If the socket is nonblocking and no messages are available, the **recvmsg** subroutine is unsuccessful.

Use the **select** subroutine to determine when more data arrives.

The **recvmsg** subroutine uses a **msghdr** structure to decrease the number of directly supplied parameters. The **msghdr** structure is defined in the **sys/socket.h** file. In BSD 4.3 Reno, the size and members of the **msghdr** structure have been modified. Applications wanting to start the old structure need to compile with **COMPAT_43** defined. The default behaviour is that of BSD 4.4.

Parameters

Socket Specifies the unique name of the socket.

Message Points to the address of the **msghdr** structure, which contains both the address for the incoming message and the space for the sender address.

Flags

Permits the subroutine to exercise control over the reception of messages. The *Flags* parameter used to receive a call is formed by logically ORing one or more of the values shown in the following list:

MSG_PEEK Peeks at incoming message.

MSG_OOB Processes out–of–band data.

The /**sys**/**socket.h** file contains the possible values for the *Flags* parameter.

Return Values

Upon successful completion, the length of the message in bytes is returned.

If the **recvmsg** subroutine is unsuccessful, the subroutine handler performs the following functions:

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

Error Codes

The **recvmsg** subroutine is unsuccessful if any of the following error codes occurs:

EBADF	The *Socket* parameter is not valid.
ENOTSOCK	The *Socket* parameter refers to a file, not a socket.
EWOULDBLOCK	The socket is marked nonblocking, and no connections are present to be accepted.
EINTR	The **recvmsg** subroutine was interrupted by delivery of a signal before any data was available for the receive.
EFAULT	The *Address* parameter is not in a writable part of the user address space.

Implementation Specifics

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

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

Related Information

The **no** command.

The **recv** subroutine, **recvfrom** subroutine, **select** subroutine, **send** subroutine, **sendmsg** subroutine, **sendto** subroutine, **shutdown** subroutine, **socket** subroutine.

Sockets Overview and Understanding Socket Data Transfer in *AIX Communications Programming Concepts*.

res_init Subroutine

Purpose

Searches for a default domain name and Internet address.

Library

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

Syntax

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

void res_init ( )
```

Description

The **res_init** subroutine reads the /**etc**/**resolv.conf** file for the default domain name and the Internet address of the initial hosts running the name server.

Note: If the /**etc**/**resolv.conf** file does not exist, the **res_init** subroutine attempts name resolution using the local /**etc**/**hosts** file. If the system is not using a domain name server, the /**etc**/**resolv.conf** file should not exist. The /**etc**/**hosts** file should be present on the system even if the system is using a name server. In this instance, the file should contain the host IDs that the system requires to function even if the name server is not functioning.

The **res_init** subroutine is one of a set of subroutines that form the resolver, a set of functions that translate domain names to Internet addresses. All resolver subroutines use the /**usr**/**include**/**resolv.h** file, which defines the **_res** structure. The **res_init** subroutine stores domain name information in the **_res** structure. Three environment variables, **LOCALDOMAIN**, **RES_TIMEOUT**, and **RES_RETRY**, affect default values related to the **_res** structure.

For more information on the **_res** structure, see ”Understanding Domain Name Resolution” in *AIX Communications Programming Concepts*.

Implementation Specifics

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

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

Files

/etc/resolv.conf

Contains the name server and domain name.

/etc/hosts

Contains host names and their addresses for hosts in a network. This file is used to resolve a host name into an Internet address.

Related Information

The **dn_comp** subroutine, **dn_expand** subroutine, **_getlong** subroutine, **_getshort** subroutine, **putlong** subroutine, **putshort** subroutine, **res_mkquery** subroutine, **res_query** subroutine, **res_search** subroutine, **res_send** subroutine.

Sockets Overview and Understanding Domain Name Resolution in *AIX Communications Programming Concepts*.

res_mkquery Subroutine

Purpose

Makes query messages for name servers.

Library

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

Syntax

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

int res_mkquery (Operation, DomName, Class, Type, Data,
DataLength)
int res_mkquery (Reserved, Buffer, BufferLength)
int Operation;
char *DomName;
int Class, Type;
char *Data;
int DataLength;
struct rrec *Reserved;
char *Buffer;
int BufferLength;
```

Description

The **res_mkquery** subroutine creates packets for name servers in the Internet domain. The subroutine also creates a standard query message. The *Buffer* parameter determines the location of this message.

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

Parameters

Operation Specifies a query type. The usual type is **QUERY**, but the parameter can be set to any of the query types defined in the **arpa/nameser.h** file.

DomName Points to the name of the domain. If the *DomName* parameter points to a single label and the **RES_DEFNAMES** structure is set, as it is by default, the subroutine appends the *DomName* parameter to the current domain name. The current domain name is defined by the name server in use or in the **/etc/resolv.conf** file.

Class

Specifies one of the following parameters:

C_IN Specifies the ARPA Internet.

C_CHAOS Specifies the Chaos network at MIT.

Type

Requires one of the following values:

T_A	Host address
T_NS	Authoritative server
T_MD	Mail destination
T_MF	Mail forwarder
T_CNAME	Canonical name
T_SOA	Start–of–authority zone
T_MB	Mailbox–domain name
T_MG	Mail–group member
T_MR	Mail–rename name
T_NULL	Null resource record
T_WKS	Well–known service
T_PTR	Domain name pointer
T_HINFO	Host information
T_MINFO	Mailbox information
T_MX	Mail–routing information
T_UINFO	User (**finger** command) information
T_UID	User ID
T_GID	Group ID

Data Points to the data that is sent to the name server as a search key. The data is stored as a character array.

DataLength Defines the size of the array pointed to by the *Data* parameter.

Reserved Specifies a reserved and currently unused parameter.

Buffer Points to a location containing the query message.

BufferLength Specifies the length of the message pointed to by the *Buffer* parameter.

Return Values

Upon successful completion, the **res_mkquery** subroutine returns the size of the query. If the query is larger than the value of the *BufferLength* parameter, the subroutine is unsuccessful and returns a value of –1.

Implementation Specifics

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

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

Files

/etc/resolv.conf

Contains the name server and domain name.

Related Information

The **finger** command.

The **dn_comp** subroutine, **dn_expand** subroutine, **_getlong** subroutine, **_getshort** subroutine, **putlong** subroutine, **putshort** subroutine, **res_init** subroutine, **res_query** subroutine, **res_search** subroutine, **res_send** subroutine.

Sockets Overview and Understanding Domain Name Resolution in *AIX Communications Programming Concepts*.

res_query Subroutine

Purpose

Provides an interface to the server query mechanism.

Library

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

Syntax

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

int res_query (DomName, Class, Type, Answer, AnswerLength)
char *DomName;
int Class;
int Type;
u_char *Answer;
int AnswerLength;
```

Description

The **res_query** subroutine provides an interface to the server query mechanism. It constructs a query, sends it to the local server, awaits a response, and makes preliminary checks on the reply. The query requests information of the specified type and class for the fully–qualified domain name specified in the *DomName* parameter. The reply message is left in the answer buffer whose size is specified by the *AnswerLength* parameter, which is supplied by the caller.

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

Parameters

DomName Points to the name of the domain. If the *DomName* parameter points to a single–component name and the **RES_DEFNAMES** structure is set, as it is by default, the subroutine appends the default domain name to the single–component name. The current domain name is defined by the name server in use or is specified in the **/etc/resolv.conf** file.

Class Specifies one of the following values:

C_IN	Specifies the ARPA Internet.
C_CHAOS	Specifies the Chaos network at MIT.

Type

Requires one of the following values:

T_A	Host address
T_NS	Authoritative server
T_MD	Mail destination
T_MF	Mail forwarder
T_CNAME	Canonical name
T_SOA	Start–of–authority zone
T_MB	Mailbox–domain name
T_MG	Mail–group member
T_MR	Mail–rename name
T_NULL	Null resource record
T_WKS	Well–known service
T_PTR	Domain name pointer
T_HINFO	Host information
T_MINFO	Mailbox information
T_MX	Mail–routing information
T_UINFO	User (**finger** command) information
T_UID	User ID
T_GID	Group ID

Answer Points to an address where the response is stored.

AnswerLength Specifies the size of the answer buffer.

Return Values

Upon successful completion, the **res_query** subroutine returns the size of the response. Upon unsuccessful completion, the **res_query** subroutine returns a value of –1 and sets the **h_errno** value to the appropriate error.

Implementation Specifics

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

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

Files

/etc/resolv.conf

Contains the name server and domain name.

Related Information

The **finger** command.

The **dn_comp** subroutine, **dn_expand** subroutine, **_getlong** subroutine, **_getshort** subroutine, **putlong** subroutine, **putshort** subroutine, **res_init** subroutine, **res_mkquery** subroutine, **res_search** subroutine, **res_send** subroutine.

Sockets Overview and Understanding Domain Name Resolution in *AIX Communications Programming Concepts*.

res_search Subroutine

Purpose

Makes a query and awaits a response.

Library

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

Syntax

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

int res_search (DomName, Class, Type, Answer, AnswerLength)
char *DomName;
int Class;
int Type;
u_char *Answer;
int AnswerLength;
```

Description

The **res_search** subroutine makes a query and awaits a response like the **res_query** subroutine. However, it also implements the default and search rules controlled by the **RES_DEFNAMES** and **RES_DNSRCH** options.

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

Parameters

DomName Points to the name of the domain. If the *DomName* parameter points to a single–component name and the **RES_DEFNAMES** structure is set, as it is by default, the subroutine appends the default domain name to the single–component name. The current domain name is defined by the name server in use or is specified in the /**etc**/**resolv.conf** file.

If the **RES_DNSRCH** bit is set, as it is by default, the **res_search** subroutine searches for host names in both the current domain and in parent domains.

Class

Specifies one of the following values:

C_IN Specifies the ARPA Internet.

C_CHAOS Specifies the Chaos network at MIT.

Type

Requires one of the following values:

T_A	Host address
T_NS	Authoritative server
T_MD	Mail destination
T_MF	Mail forwarder
T_CNAME	Canonical name
T_SOA	Start–of–authority zone
T_MB	Mailbox–domain name
T_MG	Mail–group member
T_MR	Mail–rename name
T_NULL	Null resource record
T_WKS	Well–known service
T_PTR	Domain name pointer
T_HINFO	Host information
T_MINFO	Mailbox information
T_MX	Mail–routing information
T_UINFO	User (**finger** command) information
T_UID	User ID
T_GID	Group ID

Answer Points to an address where the response is stored.

AnswerLength Specifies the size of the answer buffer.

Return Values

Upon successful completion, the **res_search** subroutine returns the size of the response. Upon unsuccessful completion, the **res_search** subroutine returns a value of –1 and sets the **h_errno** value to the appropriate error.

Implementation Specifics

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

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

Files

/etc/resolv.conf

Contains the name server and domain name.

Related Information

The **finger** command.

The **dn_comp** subroutine, **dn_expand** subroutine, **_getlong** subroutine, **_getshort** subroutine, **putlong** subroutine, **putshort** subroutine, **res_init** subroutine, **res_mkquery** subroutine, **res_query** subroutine, **res_send** subroutine.

Sockets Overview and Understanding Domain Name Resolution in *AIX Communications Programming Concepts*.

res_send Subroutine

Purpose

Sends a query to a name server and retrieves a response.

Library

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

Syntax

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

int res_send (MessagePtr, MessageLength, Answer, AnswerLength)
char *MsgPtr;
int MsgLength;
char *Answer;
int AnswerLength;
```

Description

The **res_send** subroutine sends a query to name servers and calls the **res_init** subroutine if the **RES_INIT** option of the **_res** structure is not set. This subroutine sends the query to the local name server and handles time outs and retries.

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

Parameters

MessagePtr	Points to the beginning of a message.
MessageLength	Specifies the length of the message.
Answer	Points to an address where the response is stored.
AnswerLength	Specifies the size of the answer area.

Return Values

Upon successful completion, the **res_send** subroutine returns the length of the message.

If the **res_send** subroutine is unsuccessful, the subroutine returns a –1.

Implementation Specifics

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

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

Files

/etc/resolv.conf

Contains general name server and domain name information.

Related Information

The **dn_comp** subroutine, **dn_expand** subroutine, **_getlong** subroutine, **_getshort** subroutine, **putlong** subroutine, **putshort** subroutine, **res_init** subroutine, **res_mkquery** subroutine, **res_query** subroutine, **res_search** subroutine.

Sockets Overview and Understanding Domain Name Resolution in *AIX Communications Programming Concepts*.

rexec Subroutine

Purpose

Allows command execution on a remote host.

Library

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

Syntax

```
int rexec (Host, Port, User, Passwd, Command, ErrFileDescParam)
char **Host;
int Port;
char *User, *Passwd,
*Command;
int *ErrFileDescParam;
```

Description

The **rexec** subroutine allows the calling process to start commands on a remote host.

If the **rexec** connection succeeds, a socket in the Internet domain of type **SOCK_STREAM** is returned to the calling process and is given to the remote command as standard input and standard output.

Parameters

Host Contains the name of a remote host that is listed in the /**etc**/**hosts** file or /**etc**/**resolv.config** file. If the name of the host is not found in either file, the **rexec** subroutine is unsuccessful.

Port Specifies the well–known DARPA Internet port to use for the connection. A pointer to the structure that contains the necessary port can be obtained by issuing the following library call:

```
getservbyname("exec","tcp")
```

User and *Passwd* Points to a user ID and password valid at the host. If these parameters are not supplied, the **rexec** subroutine takes the following actions until finding a user ID and password to send to the remote host:

1. Searches the current environment for the user ID and password on the remote host.
2. Searches the user's home directory for a file called **$HOME/.netrc** that contains a user ID and password.
3. Prompts the user for a user ID and password.

Command Points to the name of the command to be executed at the remote host.

ErrFileDescParam

Specifies one of the following values:

Non–zero Indicates an auxiliary channel to a control process is set up, and a descriptor for it is placed in the *ErrFileDescParam* parameter. The control process provides diagnostic output from the remote command on this channel and also accepts bytes as signal numbers to be forwarded to the process group of the command. This diagnostic information does not include remote authorization failure, since this connection is set up after authorization has been verified.

0 Indicates the standard error of the remote command is the same as standard output, and no provision is made for sending arbitrary signals to the remote process. In this case, however, it may be possible to send out–of–band data to the remote command.

Return Values

Upon successful completion, the system returns a socket to the remote command.

If the **rexec** subroutine is unsuccessful, the system returns a –1 indicating that the specified host name does not exist.

Implementation Specifics

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

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

Files

/etc/hosts	Contains host names and their addresses for hosts in a network. This file is used to resolve a host name into an Internet address.
/etc/resolv.conf	Contains the name server and domain name.
$HOME/.netrc	Contains automatic login information.

Related Information

The **getservbyname** subroutine, **rcmd** subroutine, **rresvport** subroutine, **ruserok** subroutine.

The **rexecd** daemon.

The TCP/IP Overview for System Management in *AIX 4.3 System Management Guide: Communications and Networks*.

Sockets Overview in *AIX Communications Programming Concepts*.

rresvport Subroutine

Purpose

Retrieves a socket with a privileged address.

Library

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

Syntax

```
int rresvport (Port)
int *Port;
```

Description

The **rresvport** subroutine obtains a socket with a privileged address bound to the socket. A privileged Internet port is one that falls in a range between 0 and 1023.

Only processes with an effective user ID of root user can use the **rresvport** subroutine. An authentication scheme based on remote port numbers is used to verify permissions.

If the connection succeeds, a socket in the Internet domain of type **SOCK_STREAM** is returned to the calling process.

Parameters

Port	Specifies the port to use for the connection.

Return Values

Upon successful completion, the **rresvport** subroutine returns a valid, bound socket descriptor.

If the **rresvport** subroutine is unsuccessful, the subroutine handler performs the following functions:

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

Error Codes

The **rresvport** subroutine is unsuccessful if any of the following errors occurs:

EAGAIN	All network ports are in use.
EAFNOSUPPORT	The addresses in the specified address family cannot be used with this socket.
EMFILE	Two hundred file descriptors are currently open.
ENFILE	The system file table is full.
ENOBUFS	Insufficient buffers are available in the system to complete the subroutine.

Implementation Specifics

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

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

Files

/etc/services	Contains the service names.

Related Information

The **rcmd** subroutine, **ruserok** subroutine.

Sockets Overview in *AIX Communications Programming Concepts*.

ruserok Subroutine

Purpose

Allows servers to authenticate clients.

Library

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

Syntax

```
int ruserok (Host,
RootUser, RemoteUser, LocalUser)
char *Host;
int RootUser;
char *RemoteUser,
*LocalUser;
```

Description

The **ruserok** subroutine allows servers to authenticate clients requesting services.

Always specify the host name. If the local domain and remote domain are the same, specifying the domain parts is optional. To determine the domain of the host, use the **gethostname** subroutine.

Parameters

Host	Specifies the name of a remote host. The **ruserok** subroutine checks for this host in the /**etc**/**host.equiv** file. Then, if necessary, the subroutine checks a file in the user's home directory at the server called /**$HOME**/.**rhosts** for a host and remote user ID.
RootUser	Specifies a value to indicate whether the effective user ID of the calling process is a root user. A value of 0 indicates the process does not have a root user ID. A value of 1 indicates that the process has local root user privileges, and the /**etc**/**hosts.equiv** file is not checked.
RemoteUser	Points to a user name that is valid at the remote host. Any valid user name can be specified.
LocalUser	Points to a user name that is valid at the local host. Any valid user name can be specified.

Return Values

The **ruserok** subroutine returns a 0, if the subroutine successfully locates the name specified by the *Host* parameter in the /**etc**/**hosts.equiv** file or the IDs specified by the *Host* and *RemoteUser* parameters are found in the /**$HOME**/**.rhosts** file.

If the name specified by the *Host* parameter was not found, the **ruserok** subroutine returns a –1.

Implementation Specifics

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

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

Files

/etc/services	Contains service names.
/etc/host.equiv	Specifies foreign host names.
/$HOME/.rhosts	Specifies the remote users of a local user account.

Related Information

The **rlogind** command, **rshd** command.

The **gethostname** subroutine, **rcmd** subroutine, **rresvport** subroutine, **sethostname** subroutine.

Sockets Overview in *AIX Communications Programming Concepts*.

send Subroutine

Purpose

Sends messages from a connected socket.

Library

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

Syntax

```
#include <sys/types.h>
#include <sys/socketvar.h>
#include <sys/socket.h>

int send (Socket,
Message, Length, Flags)
int Socket;
const void *Message;
size_t Length;
int Flags;
```

Description

The **send** subroutine sends a message only when the socket is connected. The **sendto** and **sendmsg** subroutines can be used with unconnected or connected sockets.

To broadcast on a socket, first issue a **setsockopt** subroutine using the **SO_BROADCAST** option to gain broadcast permissions.

Specify the length of the message with the *Length* parameter. If the message is too long to pass through the underlying protocol, the system returns an error and does not transmit the message.

No indication of failure to deliver is implied in a **send** subroutine. A return value of –1 indicates some locally detected errors.

If no space for messages is available at the sending socket to hold the message to be transmitted, the **send** subroutine blocks unless the socket is in a nonblocking I/O mode. Use the **select** subroutine to determine when it is possible to send more data.

Parameters

Socket Specifies the unique name for the socket.

Message Points to the address of the message to send.

Length Specifies the length of the message in bytes.

Flags

Allows the sender to control the transmission of the message. The *Flags* parameter used to send a call is formed by logically ORing one or both of the values shown in the following list:

MSG_OOB Processes out–of–band data on sockets that support **SOCK_STREAM** communication.

MSG_DONTROUTE
Sends without using routing tables.

MSG_MPEG2 Indicates that this block is a MPEG2 block. This flag is valid **SOCK_CONN_DGRAM** types of sockets only.

Return Values

Upon successful completion, the **send** subroutine returns the number of characters sent.

If the **send** subroutine is unsuccessful, the subroutine handler performs the following functions:

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

Error Codes

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

EBADF	The *Socket* parameter is not valid.
ENOTSOCK	The *Socket* parameter refers to a file, not a socket.
EFAULT	The *Address* parameter is not in a writable part of the user address space.
EWOULDBLOCK	The socket is marked nonblocking, and no connections are present to be accepted.

Implementation Specifics

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

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

Related Information

The **connect** subroutine, **getsockopt** subroutine, **recv** subroutine, **recvfrom** subroutine, **recvmsg** subroutine, **select** subroutine, **sendmsg** subroutine, **sendto** subroutine, **setsockopt** subroutine. **shutdown** subroutine, **socket** subroutine.

Sockets Overview and Understanding Socket Data Transfer in *AIX Communications Programming Concepts*.

sendmsg Subroutine

Purpose

Sends a message from a socket using a message structure.

Library

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

Syntax

```
#include <sys/types.h>
#include <sys/socketvar.h>
#include <sys/socket.h>

int sendmsg (Socket, Message, Flags)
int Socket;
const struct msghdr Message [ ];
int Flags;
```

Description

The **sendmsg** subroutine sends messages through connected or unconnected sockets using the **msghdr** message structure. The /**usr**/**include**/**sys**/**socket.h** file contains the **msghdr** structure and defines the structure members. In BSD 4.4, the size and members of the **msghdr** message structure have been modified. Applications wanting to start the old structure need to compile with **COMPAT_43** defined. The default behaviour is that of BSD 4.4.

To broadcast on a socket, the application program must first issue a **setsockopt** subroutine using the **SO_BROADCAST** option to gain broadcast permissions.

The **sendmsg** subroutine supports only 15 message elements.

Parameters

Socket Specifies the socket descriptor.

Message Points to the **msghdr** message structure containing the message to be sent.

Flags

Allows the sender to control the message transmission. The **sys**/**socket.h** file contains the *Flags* parameter. The *Flags* parameter used to send a call is formed by logically ORing one or both of the following values:

MSG_OOB Processes out–of–band data on sockets that support **SOCK_STREAM**.

Note: The following value is not for general use. It is an administrative tool used for debugging or for routing programs.

MSG_DONTROUTE
Sends without using routing tables.

MSG_MPEG2 Indicates that this block is a MPEG2 block. It only applies to **SOCK_CONN_DGRAM** types of sockets only.

Return Values

Upon successful completion, the **sendmsg** subroutine returns the number of characters sent.

If the **sendmsg** subroutine is unsuccessful, the system handler performs the following functions:

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

Error Codes

The **sendmsg** subroutine is unsuccessful if any of the following errors occurs:

EBADF	The *Socket* parameter is not valid.
ENOTSOCK	The *Socket* parameter refers to a file, not a socket.
EWOULDBLOCK	The socket is marked nonblocking, and no connections are present to be accepted.

Implementation Specifics

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

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

Related Information

The **on** command.

The **connect** subroutine, **getsockopt** subroutine, **recv** subroutine, **recvfrom** subroutine, **recvmsg** subroutine, **select** subroutine, **send** subroutine, **sendto** subroutine, **setsockopt** subroutine. **shutdown** subroutine, **socket** subroutine.

Sockets Overview and Understanding Socket Data Transfer in *AIX Communications Programming Concepts*.

sendto Subroutine

Purpose

Sends messages through a socket.

Library

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

Syntax

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

int sendto
(Socket, Message, Length,
Flags, To, ToLength)
int Socket;
const void *Message;
size_t Length;
int Flags;
const struct sockaddr *To;
size_t ToLength;
```

Description

The **sendto** subroutine allows an application program to send messages through an unconnected socket by specifying a destination address.

To broadcast on a socket, first issue a **setsockopt** subroutine using the **SO_BROADCAST** option to gain broadcast permissions.

Provide the address of the target using the *To* parameter. Specify the length of the message with the *Length* parameter. If the message is too long to pass through the underlying protocol, the error **EMSGSIZE** is returned and the message is not transmitted.

If the **sending** socket has no space to hold the message to be transmitted, the **sendto** subroutine blocks the message unless the socket is in a nonblocking I/O mode.

Use the **select** subroutine to determine when it is possible to send more data.

Parameters

Socket Specifies the unique name for the socket.

Message Specifies the address containing the message to be sent.

Length Specifies the size of the message in bytes.

Flags

Allows the sender to control the message transmission. The *Flags* parameter used to send a call is formed by logically ORing one or both of the following values:

MSG_OOB Processes out–of–band data on sockets that support **SOCK_STREAM**.

Note: The following value is not for general use.

MSG_DONTROUTE
Sends without using routing tables.

The /**usr**/**include**/**sys**/**socket.h** file defines the *Flags* parameter.

To	Specifies the destination address for the message. The destination address is a **sockaddr** structure defined in the **/usr/include/sys/socket.h** file.
ToLength	Specifies the size of the destination address.

Return Values

Upon successful completion, the **sendto** subroutine returns the number of characters sent.

If the **sendto** subroutine is unsuccessful, the system returns a value of –1, and the **errno** global variable is set to indicate the error.

Error Codes

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

EBADF	The *Socket* parameter is not valid.
ENOTSOCK	The *Socket* parameter refers to a file, not a socket.
EFAULT	The *Address* parameter is not in a writable part of the user address space.
EMSGSIZE	The message is too large to be sent all at once as the socket requires.
EWOULDBLOCK	The socket is marked nonblocking, and no connections are present to be accepted.

Implementation Specifics

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

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

Related Information

The **getsockopt** subroutine, **recv** subroutine, **recvfrom** subroutine, **recvmsg** subroutine, **select** subroutine, **send** subroutine, **sendmsg** subroutine, **setsockopt** subroutine. **shutdown** subroutine, **socket** subroutine.

Sending UNIX Datagrams Example Program, Sending Internet Datagrams Example Program, Sockets Overview, Understanding Socket Data Transfer in *AIX Communications Programming Concepts*.

send_file Subroutine

Purpose

Sends the contents of a file through a socket.

Library

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

Syntax

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

ssize_t send_file(Socket_p, sf_iobuf, flags)

int *Socket_p;
struct sf_parms *sf_iobuf;
uint_t flags;
```

Description

The **send_file** subroutine sends data from the opened file specified in the *sf_iobuf* parameter, over the connected socket pointed to by the *Socket_p* parameter.

Note: Currently, the **send_file** only supports the TCP/IP protocol (SOCK_STREAM socket in AF_INET). An error will be returned when this function is used on any other types of sockets.

Parameters

Socket_p Points to the socket descriptor of the socket which the file will be sent to.

Note: This is different from most of the socket functions.

sf_iobuf

Points to a *sf_parms* structure defined as follows:

```
/*
 * Structure for the send_file system call
 */
#ifdef __64BIT__
#define SF_INT64(x)     int64_t x;
#define SF_UINT64(x)    uint64_t x;
#else
#ifdef _LONG_LONG
#define SF_INT64(x)     int64_t x;
#define SF_UINT64(x)    uint64_t x;
#else
#define SF_INT64(x)     int filler_##x; int x;
#define SF_UINT64(x)    int filler_##x; uint_t x;
#endif
#endif

struct sf_parms {
    /* --------- header parms ---------- */
    void      *header_data;         /* Input/Output.
Points to header buf */
    uint_t      header_length;      /* Input/Output.
Length of the header */
    /* --------- file parms ------------ */
    int       file_descriptor;      /* Input. File
descriptor of the file */
    SF_UINT64(file_size)            /* Output. Size of
the file */
    SF_UINT64(file_offset)          /* Input/Output.
Starting offset */
    SF_INT64(file_bytes)            /* Input/Output.
number of bytes to send */
    /* --------- trailer parms --------- */
   void      *trailer_data;         /* Input/Output.
Points to trailer buf */
   uint_t    trailer_length;        /* Input/Output.
Length of the trailer */
    /* --------- return info ----------- */
    SF_UINT64(bytes_sent)           /* Output. number
of bytes sent */
};
```

header_data Points to a buffer that contains header data which is to be sent before the file data. May be a NULL pointer if *header_length* is 0. This field will be updated by **send_file** when header is transmitted – that is, *header_data* + number of bytes of the header sent.

header_length Specifies the number of bytes in the *header_data*. This field must be set to 0 to indicate that header data is not to be sent. This field will be updated by **send_file** when header is transmitted – that is, *header_length* – number of bytes of the header sent.

file_descriptor Specifies the file descriptor for a file that has been opened and is readable. This is the descriptor for the file that contains the data to be transmitted. The *file_descriptor* is ignored when *file_bytes* = 0. This field is not updated by **send_file**.

file_size Contains the byte size of the file specified by *file_descriptor*. This field is filled in by the kernel.

file_offset Specifies the byte offset into the file from which to start sending data. This field is updated by the **send_file** when file data is transmitted – that is, `file_offset` + number of bytes of the file data sent.

file_bytes Specifies the number of bytes from the file to be transmitted. Setting *file_bytes* to –1 transmits the entire file from the *file_offset*. When this field is not set to –1, it is updated by **send_file** when file data is transmitted – that is, `file_bytes` – number of bytes of the file data sent.

trailer_data Points to a buffer that contains trailer data which is to be sent after the file data. May be a NULL pointer if *trailer_length* is 0. This field will be updated by **send_file** when trailer is transmitted – that is, `trailer_data` + number of bytes of the trailer sent.

trailer_length Specifies the number of bytes in the *trailer_data*. This field must be set to 0 to indicate that trailer data is not to be sent. This field will be updated by **send_file** when trailer is transmitted – that is, `trailer_length` – number of bytes of the trailer sent.

bytes_sent Contains number of bytes that were actually sent in this call to **send_file**. This field is filled in by the kernel.

All fields marked with `Input` in the *sf_parms* structure requires setup by an application prior to the **send_file** calls. All fields marked with `Output` in the *sf_parms* structure adjusts by **send_file** when it successfully transmitted data, that is, either the specified data transmission is partially or completely done.

The **send_file** subroutine attempts to write *header_length* bytes from the buffer pointed to by *header_data*, followed by *file_bytes* from the file associated with *file_descriptor*, followed by *trailer_length* bytes from the buffer pointed to by *trailer_data*, over the connection associated with the socket pointed to by *Socket_p*.

As the data is sent, the kernel updates the parameters pointed by *sf_iobuf* so that if the **send_file** has to be called multiple times (either due to interruptions by signals, or due to non–blocking I/O mode) in order to complete a file data transmission, the application can reissue the **send_file** command without setting or re–adjusting the parameters over and over again.

If the application sets *file_offset* greater than the actual file size, or *file_bytes* greater than (the actual file size – *file_offset*), the return value will be –1 with errno EINVAL.

flags

Specifies the following attributes:

SF_CLOSE Closes the socket pointed to by *Socket_p* after the data has been successfully sent or queued for transmission.

SF_REUSE Prepares the socket for reuse after the data has been successfully sent or queued for transmission and the existing connection closed.

Note: This option is currently not supported on AIX.

SF_DONT_CACHE
Does not put the specified file in the Network Buffer Cache.

SF_SYNC_CACHE
Verifies/Updates the Network Buffer Cache for the specified file before transmission.

When the *SF_CLOSE* flag is set, the connected socket specified by *Socket_p* will be disconnected and closed by **send_file** after the requested transmission has been successfully done. The socket descriptor pointed to by *Socket_p* will be set to –1. This flag won't take effect if **send_file** returns non–0.

The flag *SF_REUSE* currently is not supported by AIX. When this flag is specified, the socket pointed by *Socket_p* will be closed and returned as –1. A new socket needs to be created for the next connection.

send_file will take advantage of a Network Buffer Cache in kernel memory to dynamically cache the output file data. This will help to improve the **send_file** performance for files which are:

1. accessed repetitively through network and
2. not changed frequently.

Applications can exclude the specified file from being cached by using the *SF_DONT_CACHE* flag. **send_file** will update the cache every so often to make sure that the file data in cache is valid for a certain time period. The network option parameter "send_file_duration" controlled by the **no** command can be modified to configure the interval of the **send_file** cache validation, the default is 300 (in seconds). Applications can use the *SF_SYNC_CACHE* flag to ensure that a cache validation of the specified file will occur before the file is sent by **send_file**, regardless the value of the "send_file_duration". Other Network Buffer Cache related parameters are "nbc_limit", nbc_max_cache", and nbc_min_cache". For additional infromation, see the **no** command.

Return Value

There are three possible return values from **send_file**:

–1	an error has occurred, errno contains the error code.
0	the command has completed successfully.
1	the command was completed partially, some data has been transmitted but the command has to return for some reason, for example, the command was interrupted by signals.

The fields marked with `Output` in the *sf_parms* structure (pointed to by *sf_iobuf*) is updated by **send_file** when the return value is either 0 or 1. The *bytes_sent* field contains the total number of bytes that were sent in this call. It is always true that *bytes_sent* (`Output`) <= *header_length*(`Input`) + *file_bytes*(`Input`) + *trailer_length* (`Input`).

The **send_file** supports the blocking I/O mode and the non–blocking I/O mode. In the blocking I/O mode, **send_file** blocks until all file data (plus the header and the trailer) is sent. It adjusts the *sf_iobuf* to reflect the transmission results, and return 0. It is possible that **send_file** can be interrupted before the request is fully done, in that case, it adjusts the *sf_iobuf* to reflect the transmission progress, and return 1.

In the non–blocking I/O mode, the **send_file** transmits as much as the socket space allows, adjusts the *sf_iobuf* to reflect the transmission progress, and returns either 0 or 1. When there is no socket space in the system to buffer any of the data, the **send_file** returns –1 and sets errno to EWOULDBLOCK. **select** or **poll** can be used to determine when it is possible to send more data.

Possible errno returned:	
EBADF	Either the socket or the file descriptor parameter is not valid.
ENOTSOCK	The socket parameter refers to a file, not a socket.
EPROTONOSUPPORT	Protocol not supported.
EFAULT	The addresses specified in the HeaderTailer parameter is not in a writable part of the user–address space.
EINTR	The operation was interrupted by a signal before any data was sent. (If some data was sent, **send_file** returns the number of bytes sent before the signal, and EINTR is not set).
EINVAL	The offset, length of the HeaderTrailer, or flags parameter is invalid.
ENOTCONN	A **send_file** on a socket that is not connected, a **send_file** on a socket that has not completed the connect sequence with its peer, or is no longer connected to its peer.
EWOULDBLOCK	The socket is marked non–blocking and the requested operation would block.
ENOMEM	No memory is available in the system to perform the operation.

PerformanceNote

By taking advantage of the Network Buffer Cache, **send_file** provides better performance and network throughput for file transmission. It is recommanded for files bigger than 4K bytes.

Related Information

The **connect** subroutine, **getsockopt** subroutine, **recv** subroutine, **recvfrom** subroutine, **recvmsg** subroutine, **select** subroutine, **sendmsg** subroutine, **sendto** subroutine, **setsockopt** subroutine, **shutdown** subroutine, **socket** subroutine.

Sockets Overview and Understanding Socket Data Transfer in *AIX Communications Programming Concepts*.

setdomainname Subroutine

Purpose

Sets the name of the current domain.

Library

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

Syntax

```
int setdomainname (Name, Namelen)
char *Name;
int Namelen;
```

Description

The **setdomainname** subroutine sets the name of the domain for the host machine. It is normally used when the system is bootstrapped. You must have root user authority to run this subroutine.

The purpose of domains is to enable two distinct networks that may have host names in common to merge. Each network would be distinguished by having a different domain name. At the current time, only Network Information Service (NIS) makes use of domains set by this subroutine.

Note: Domain names are restricted to 64 characters.

Parameters

Name	Specifies the domain name to be set.
Namelen	Specifies the size of the array pointed to by the *Name* parameter.

Return Values

If the call succeeds, a value of 0 is returned. If the call is unsuccessful, a value of –1 is returned and an error code is placed in the **errno** global variable.

Error Codes

The following error may be returned by this subroutine:

EFAULT	The *Name* parameter gave an invalid address.
EPERM	The caller was not the root user.

Implementation Specifics

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

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

Related Information

The **getdomainname** subroutine, **gethostname** subroutine, **sethostname** subroutine.

Sockets Overview in *AIX Communications Programming Concepts*.

sethostent Subroutine

Purpose

Opens network host file.

Library

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

Syntax

```
#include <netdb.h>

sethostent (StayOpen)
int StayOpen;
```

Description

When using the **sethostent** subroutine in DNS/BIND name service resolution, **sethostent** allows a request for the use of a connected socket using TCP for queries. If the *StayOpen* parameter is non–zero, this sets the option to send all queries to the name server using TCP and to retain the connection after each call to **gethostbyname** or **gethostbyaddr**.

When using the **sethostent subroutine** to search the /**etc**/**hosts** file, **sethostent** opens and rewinds the /**etc**/**hosts** file. If the *StayOpen* parameter is non–zero, the hosts database is not closed after each call to **gethostbyname** or **gethostbyaddr**.

Parameters

StayOpen	When used in NIS name resolution and to search the local /**etc**/**hosts** file, it contains a value used to indicate whether to close the host file after each call to **gethostbyname** and **gethostbyaddr**. A non–zero value indicates not to close the host file after each call and a zero value allows the file to be closed. When used in DNS/BIND name resolution, a non–zero value retains the TCP connection after each call to **gethostbyname** and **gethostbyaddr** . A value of zero allows the connection to be closed.

Implementation Specifics

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

Files

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

Related Information

The **endhostent** subroutine, **gethostbyaddr** subroutine, **gethostbyname** subroutine, **gethostent** subroutine.

Sockets Overview and Network Address Translation in *AIX Communications Programming Concepts*.

sethostid Subroutine

Purpose

Sets the unique identifier of the current host.

Library

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

Syntax

```
int sethostid (HostID)
int HostID;
```

Description

The **sethostid** subroutine allows a calling process with a root user ID to set a new 32–bit identifier for the current host. The **sethostid** subroutine enables an application program to reset the host ID.

Parameters

HostID	Specifies the unique 32–bit identifier for the current host.

Return Values

Upon successful completion, the **sethostid** subroutine returns a value of 0.

If the **sethostid** subroutine is unsuccessful, the subroutine handler performs the following functions:

- Returns a value of –1 to the calling program.
- Moves an error code, indicating the specific error, into the **errno** global variable. For further explanation of the **errno** variable see Error Notification Object Class in *AIX General Programming Concepts : Writing and Debugging Programs*.

Error Codes

The **sethostid** subroutine is unsuccessful if the following is true:

EPERM	The calling process did not have an effective user ID of root user.

Implementation Specifics

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

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

Related Information

The **getsockname** subroutine, **gethostid** subroutine, **gethostname** subroutine.

Sockets Overview in *AIX Communications Programming Concepts*.

sethostname Subroutine

Purpose

Sets the name of the current host.

Library

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

Syntax

```
int sethostname (Name, NameLength)
char *Name;
int NameLength;
```

Description

The **sethostname** subroutine sets the name of a host machine. Only programs with a root user ID can use this subroutine.

The **sethostname** subroutine allows a calling process with root user authority to set the internal host name of a machine on a network.

Parameters

Name	Specifies the name of the host machine.
NameLength	Specifies the length of the *Name* array.

Return Values

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

If the **sethostname** subroutine is unsuccessful, the subroutine handler performs the following functions:

- Returns a value of –1 to the calling program.
- Moves an error code, indicating the specific error, into the **errno** global variable. For further explanation of the **errno** variable see Error Notification Object Class in *AIX General Programming Concepts : Writing and Debugging Programs*.

Error Codes

The **sethostname** subroutine is unsuccessful if any of the following errors occurs:

EFAULT	The *Name* parameter or *NameLength* parameter gives an address that is not valid.
EPERM	The calling process did not have an effective root user ID.

Implementation Specifics

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

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

Related Information

The **gethostid** subroutine, **gethostname** subroutine, **sethostid** subroutine.

Sockets Overview and Understanding Network Address Translation in *AIX Communications Programming Concepts*.

setnetent Subroutine

Purpose

Opens the /**etc**/**networks** file and sets the file marker.

Library

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

Syntax

```
#include <netdb.h>
void setnetent (StayOpen)
int StayOpen;
```

Description

Attention: Do not use the **setnetent** subroutine in a multithreaded environment. See the multithread alternative in the **setnetent_r** subroutine article.

Attention: Do not use the **setnetent** subroutine in a multithreaded environment.

The **setnetent** subroutine opens the /**etc**/**networks** file and sets the file marker at the beginning of the file.

Parameters

StayOpen	Contains a value used to indicate when to close the /**etc**/**networks** file. Specifying a value of 0 closes the /**etc**/**networks** file after each call to the **getnetent** subroutine. Specifying a nonzero value leaves the /**etc**/**networks** file open after each call.

Return Values

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

Implementation Specifics

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

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

Files

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

Related Information

The **endnetent** subroutine, **getnetbyaddr** subroutine, **getnetbyname** subroutine, **getnetent** subroutine.

Sockets Overview and Understanding Network Address Translation in *AIX Communications Programming Concepts*.

setprotoent Subroutine

Purpose

Opens the **/etc/protocols** file and sets the file marker.

Library

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

Syntax

```
#include <netdb.h>

void setprotoent (StayOpen)
int StayOpen;
```

Description

Attention: Do not use the **setprotoent** subroutine in a multithreaded environment. See the multithread alternative in the **setprotoent_r** subroutine article.

Attention: Do not use the **setprotoent** subroutine in a multithreaded environment.

The **setprotoent** subroutine opens the **/etc/protocols** file and sets the file marker to the beginning of the file.

Parameters

StayOpen	Indicates when to close the **/etc/protocols** file. Specifying a value of 0 closes the file after each call to **getprotoent**. Specifying a nonzero value allows the **/etc/protocols** file to remain open after each subroutine.

Return Values

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

Implementation Specifics

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

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

Files

/etc/protocols	Contains the protocol names.

Related Information

The **endprotoent** subroutine, **getprotobyname** subroutine, **getprotobynumber** subroutine, **getprotoent** subroutine.

Sockets Overview and Understanding Network Address Translation in *AIX Communications Programming Concepts*.

setservent Subroutine

Purpose

Opens /**etc**/**services** file and sets the file marker.

Library

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

Syntax

```
#include <netdb.h>

void setservent (StayOpen)
int StayOpen;
```

Description

Attention: Do not use the **setservent** subroutine in a multithreaded environment. See the multithread alternative in the **setservent_r** subroutine article.

Attention: Do not use the **setservent** subroutine in a multithreaded environment.

The **setservent** subroutine opens the /**etc**/**services** file and sets the file marker at the beginning of the file.

Parameters

StayOpen Indicates when to close the /**etc**/**services** file.

Specifying a value of 0 closes the file after each call to the **getservent** subroutine.

Specifying a nonzero value allows the file to remain open after each call.

Return Values

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

Implementation Specifics

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

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

Files

/etc/services Contains service names.

Related Information

The **endprotoent** subroutine, **endservent** subroutine, **getprotobyname** subroutine, **getprotobynumber** subroutine, **getprotoent** subroutine, **getservbyname** subroutine, **getservbyport** subroutine, **getservent** subroutine, **setprotoent** subroutine.

Sockets Overview and Understanding Network Address Translation in *AIX Communications Programming Concepts*.

setsockopt Subroutine

Purpose

Sets socket options.

Library

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

Syntax

```
#include <sys/types.h>
#include <sys/socket.h>
#include <sys/socketvar.h>
#include <sys/atmsock.h> /*Needed for SOCK_CONN_DGRAM socket type

only*/

int setsockopt
(Socket, Level, OptionName,
OptionValue, OptionLength)
int Socket, Level, OptionName;
const void *OptionValue;
size_t OptionLength;
```

Description

The **setsockopt** subroutine sets options associated with a socket. Options can exist at multiple protocol levels. The options are always present at the uppermost socket level.

The **setsockopt** subroutine provides an application program with the means to control a socket communication. An application program can use the **setsockopt** subroutine to enable debugging at the protocol level, allocate buffer space, control time outs, or permit socket data broadcasts. The **/usr/include/sys/socket.h** file defines all the options available to the **setsockopt** subroutine.

When setting socket options, specify the protocol level at which the option resides and the name of the option.

Use the parameters *OptionValue* and *OptionLength* to access option values for the **setsockopt** subroutine. These parameters identify a buffer in which the value for the requested option or options is returned.

Parameters

Socket Specifies the unique socket name.

Level Specifies the protocol level at which the option resides. To set options at:

Socket level Specifies the *Level* parameter as **SOL_SOCKET**.

Other levels Supplies the appropriate protocol number for the protocol controlling the option. For example, to indicate that an option will be interpreted by the TCP protocol, set the *Level* parameter to the protocol number of TCP, as defined in the **netinet/in.h** file. Similarly, to indicate that an option will be interpreted by ATM protocol, set the *Level* parameter to NDDPROTO_ATM, as defined in **sys/atmsock.h**.

OptionName Specifies the option to set. The *OptionName* parameter and any specified options are passed uninterpreted to the appropriate protocol module for interpretation. The **sys/socket.h** file defines the socket protocol level options. The **netinet/tcp.h** file defines the TCP protocol level options. The socket level options can be enabled or disabled; they operate in a toggle fashion.

The following list defines socket protocol level options found in the **sys/socket.h** file:

SO_DEBUG Turns on recording of debugging information. This option enables or disables debugging in the underlying protocol modules.

SO_REUSEADDR
Specifies that the rules used in validating addresses supplied by a **bind** subroutine should allow reuse of a local port.

SO_REUSEADDR allows an application to explicitly deny subsequent **bind** subroutine to the port/address of the socket with **SO_REUSEADDR** set. This allows an application to block other applications from binding with the **bind** subroutine.

SO_REUSEPORT
Specifies that the rules used in validating addresses supplied by a **bind** subroutine should allow reuse of a local port/address combination. Each binding of the port/address combination must specify the **SO_REUSEPORT** socket option

SO_CKSUMREV
Enables performance enhancements in the protocol layers. If the protocol supports this option, enabling causes the protocol to defer checksum verification until the user's data is moved into the user's buffer (on **recv**, **recvfrom**, **read**, or **recvmsg** thread). This can cause applications to be awakened when no data is available, in the case of a checksum error. In this case, EAGAIN is returned. Applications that set this option must handle the EAGAIN error code returned from a receive call.

SO_KEEPALIVE
Monitors the activity of a connection by enabling or disabling the periodic transmission of ACK messages on a connected socket. The idle interval time can be designated using the TCP/IP **no** command. Broken connections are discussed in "Understanding Socket Types and Protocols" in *AIX Communications Programming Concepts*.

SO_DONTROUTE
Does not apply routing on outgoing messages. Indicates that outgoing messages should bypass the standard routing facilities. Instead, they are directed to the appropriate network interface according to the network portion of the destination address.

SO_BROADCAST
Permits sending of broadcast messages.

SO_LINGER Lingers on a **close** subroutine if data is present. This option controls the action taken when an unsent messages queue exists for a socket, and a process performs a **close** subroutine on the socket.

If **SO_LINGER** is set, the system blocks the process during the **close** subroutine until it can transmit the data or until the time expires. If **SO_LINGER** is not specified and a **close** subroutine is issued, the system handles the call in a way that allows the process to continue as quickly as possible.

The **sys/socket.h** file defines the **linger** structure that contains the **l_linger** value for specifying linger time interval. If linger time is set to anything but 0, the system tries to send any messages queued on the socket. Although a linger interval can be specified in seconds, the system ignores the specified time and makes repeated attempts to send unsent messages.

SO_OOBINLINE
Leaves received out–of–band data (data marked urgent) in line.

SO_SNDBUF Sets send buffer size.

SO_RCVBUF Sets receive buffer size.

SO_SNDLOWAT
Sets send low–water mark.

SO_RCVLOWAT
Sets receive low–water mark.

SO_SNDTIMEO
Sets send time out. This option is setable, but currently not used.

SO_RCVTIMEO
Sets receive time out. This option is setable, but currently not used.

SO_ERROR Sets the retrieval of error status and clear.

SO_TYPE Sets the retrieval of a socket type.

The following list defines TCP protocol level options found in the **netinet/tcp.h** file:

TCP_RFC1323 Enables or disables RFC 1323 enhancements on the specified TCP socket. An application might contain the following lines to enable RFC 1323:

```
int on=1;
setsockopt(s,IPPROTO_TCP,TCP_RFC1323,&on,sizeof(on));
```

TCP_NODELAY
Specifies whether TCP should follow the Nagle algorithm for deciding when to send data. By default, TCP will follow the Nagle algorithm. To disable this behavior, applications can enable **TCP_NODELAY** to force TCP to always send data immediately. For example, **TCP_NODELAY** should be used when there is an application using TCP for a request/response.

The following list defines ATM protocol level options found in the **sys/atmsock.h** file:

SO_ATM_PARAM
Sets all ATM parameters. This socket option can be used instead of using individual sockets options described below. It uses the **connect_ie** structure defined in **sys/call_ie.h** file.

SO_ATM_AAL_PARM
Sets ATM AAL(Adaptation Layer) parameters. It uses the **aal_parm** structure defined in **sys/call_ie.h** file.

SO_ATM_TRAFFIC_DES
Sets ATM Traffic Descriptor values. It uses the **traffic** structure defined in **sys/call_ie.h** file.

SO_ATM_BEARER
Sets ATM Bearer capability. It uses the **bearer** structure defined in **sys/call_ie.h** file.

SO_ATM_BHLI Sets ATM Broadband High Layer Information. It uses the **bhli** structure defined in **sys/call_ie.h** file.

SO_ATM_BLLI Sets ATM Broadband Low Layer Information. It uses the **blli** structure defined in **sys/call_ie.h** file.

SO_ATM_QOS Sets ATM Quality Of Service values. It uses the **qos_parm** structure defined in **sys/call_ie.h** file.

SO_ATM_TRANSIT_SEL
Sets ATM Transit Selector Carrier. It uses the **transit_sel** structure defined in **sys/call_ie.h** file.

SO_ATM_ACCEPT
Indicates acceptance of an incoming ATM call, which was indicated to the application via *ACCEPT* system call. This must be issues for the incoming

connection to be fully established. This allows negotiation of ATM parameters.

SO_ATM_MAX_PEND
Sets the number of outstanding transmit buffers that are permitted before an error indication is returned to applications as a result of a transmit operation. This option is only valid for non best effort types of virtual circuits. OptionValue/OptionLength point to a byte which contains the value that this parameter will be set to.

OptionValue The *OptionValue* parameter takes an *Int* parameter. To enable a Boolean option, set the *OptionValue* parameter to a nonzero value. To disable an option, set the *OptionValue* parameter to 0.

The following options enable and disable in the same manner:

- **SO_DEBUG**
- **SO_REUSEADDR**
- **SO_KEEPALIVE**
- **SO_DONTROUTE**
- **SO_BROADCAST**
- **SO_OOBINLINE**
- **SO_LINGER**
- **TCP_RFC1323**

OptionLength The OptionLength parameter contains the size of the buffer pointed to by the *OptionValue* parameter.

Options at other protocol levels vary in format and name.

IP level (**IPPROTO_IP** level) options are defined as follows:

IP_DONTFRAG	Sets DF bit from now on for every packet in the IP header.
IP_FINDPMTU	Sets enable/disable PMTU discovery for this path. Protocol level path MTU discovery should be enabled for the discovery to happen.
IP_PMTUAGE	Sets the age of PMTU. Specifies the frequency of PMT reductions discovery for the session. Setting it to 0 (zero) implies infinite age and PMTU reduction discovery will not be attempted. This will replace the previously set PMTU age. The new PMTU age will become effective after the currently set timer expires.

Return Values

Upon successful completion, a value of 0 is returned.

If the **setsockopt** subroutine is unsuccessful, the subroutine handler performs the following functions:

- Returns a value of –1 to the calling program.
- Moves an error code, indicating the specific error, into the **errno** global variable. For further explanation of the **errno** variable see Error Notification Object Class in *AIX General Programming Concepts : Writing and Debugging Programs*.

Error Codes

The **setsockopt** subroutine is unsuccessful if any of the following errors occurs:

EBADF	The *Socket* parameter is not valid.
ENOTSOCK	The *Socket* parameter refers to a file, not a socket.
ENOPROTOOPT	The option is unknown.
EFAULT	The *Address* parameter is not in a writable part of the user address space.

Examples

To mark a socket for broadcasting:

```
int on=1;
setsockopt(s, SOL_SOCKET, SO_BROADCAST, &on, sizeof(on));
```

Implementation Specifics

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

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

Related Information

The **no** command.

The **bind** subroutine, **endprotoent** subroutine, **getprotobynumber** subroutine, **getprotoent** subroutine, **getsockopt** subroutine, **setprotoent** subroutine, **socket** subroutine.

Sockets Overview, Understanding Socket Options, Understanding Socket Types and Protocols in *AIX Communications Programming Concepts*.

shutdown Subroutine

Purpose

Shuts down all socket send and receive operations.

Library

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

Syntax

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

int shutdown (Socket, How)
int Socket, How;
```

Description

The **shutdown** subroutine disables all receive and send operations on the specified socket.

Parameters

Socket Specifies the unique name of the socket.

How

Specifies the type of subroutine shutdown. Use the following values:

0	Disables further receive operations.
1	Disables further send operations.
2	Disables further send operations and receive operations.

Return Values

Upon successful completion, a value of 0 is returned.

If the **shutdown** subroutine is unsuccessful, the subroutine handler performs the following functions:

- Returns a value of –1 to the calling program.
- Moves an error code, indicating the specific error, into the **errno** global variable. For further explanation of the **errno** variable see Error Notification Object Class in *AIX General Programming Concepts : Writing and Debugging Programs*.

Error Codes

The **shutdown** subroutine is unsuccessful if any of the following errors occurs:

EBADF	The *Socket* parameter is not valid.
ENOTSOCK	The *Socket* parameter refers to a file, not a socket.
ENOTCONN	The socket is not connected.

Implementation Specifics

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

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

Files

/usr/include/sys/socket.h	Contains socket definitions.
/usr/include/sys/types.h	Contains definitions of unsigned data types.

Related Information

The **getsockopt** subroutine, **recv** subroutine, **recvfrom** subroutine, **recvmsg** subroutine, **read** subroutine, **select** subroutine, **send** subroutine, **sendto** subroutine, **setsockopt** subroutine, **socket** subroutine, **write** subroutine.

Sockets Overview in *AIX Communications Programming Concepts*.

socket Subroutine

Purpose

Creates an end point for communication and returns a descriptor.

Library

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

Syntax

```
#include <sys/types.h>
#include <sys/socket.h>
#include <sys/socketvar.h>

int socket (AddressFamily, Type, Protocol)
int AddressFamily, Type, Protocol;
```

Description

The **socket** subroutine creates a socket in the specified *AddressFamily* and of the specified type. A protocol can be specified or assigned by the system. If the protocol is left unspecified (a value of 0), the system selects an appropriate protocol from those protocols in the address family that can be used to support the requested socket type.

The **socket** subroutine returns a descriptor (an integer) that can be used in later subroutines that operate on sockets.

Socket level options control socket operations. The **getsockopt** and **setsockopt** subroutines are used to get and set these options, which are defined in the **/usr/include/sys/socket.h** file.

Parameters

AddressFamily

Specifies an address family with which addresses specified in later socket operations should be interpreted. The **/usr/include/sys/socket.h** file contains the definitions of the address families. Commonly used families are:

AF_UNIX Denotes the operating system path names.

AF_INET Denotes the ARPA Internet addresses.

AF_NS Denotes the XEROX Network Systems protocol.

Type

Specifies the semantics of communication. The **/usr/include/sys/socket.h** file defines the socket types. The operating system supports the following types:

SOCK_STREAM
Provides sequenced, two–way byte streams with a transmission mechanism for out–of–band data.

SOCK_DGRAM
Provides datagrams, which are connectionless messages of a fixed maximum length (usually short).

SOCK_RAW Provides access to internal network protocols and interfaces. This type of socket is available only to the root user.

Protocol Specifies a particular protocol to be used with the socket. Specifying the *Protocol* parameter of 0 causes the **socket** subroutine to default to the typical protocol for the requested type of returned socket.

Return Values

Upon successful completion, the **socket** subroutine returns an integer (the socket descriptor).

If the **socket** subroutine is unsuccessful, the subroutine handler performs the following functions:

- Returns a value of –1 to the calling program.
- Moves an error code, indicating the specific error, into the **errno** global variable. For further explanation of the **errno** variable see Error Notification Object Class in *AIX General Programming Concepts : Writing and Debugging Programs*.

Error Codes

The **socket** subroutine is unsuccessful if any of the following errors occurs:

EAFNOSUPPORT	The addresses in the specified address family cannot be used with this socket.
ESOCKTNOSUPPORT	The socket in the specified address family is not supported.
EMFILE	The per–process descriptor table is full.
ENOBUFS	Insufficient resources were available in the system to complete the call.

Examples

The following program fragment illustrates the use of the **socket** subroutine to create a datagram socket for on–machine use:

```
s = socket(AF_UNIX, SOCK_DGRAM,0);
```

Implementation Specifics

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

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

Related Information

The **accept** subroutine, **bind** subroutine, **connect** subroutine, **getsockname** subroutine, **getsockopt** subroutine, **ioctl** subroutine, **listen** subroutine, **recv** subroutine, **recvfrom** subroutine, **recvmsg** subroutine, **select** subroutine, **send** subroutine, **sendmsg** subroutine, **sendto** subroutine, **setsockopt** subroutine, **shutdown** subroutine, **socketpair** subroutine.

Initiating Internet Stream Connections Example Program, Sockets Overview, Understanding Socket Creation in *AIX Communications Programming Concepts*.

socketpair Subroutine

Purpose

Creates a pair of connected sockets.

Library

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

Syntax

```
#include <sys/types.h>
#include <sys/socket.h>
#include <sys/socketvar.h>

int socketpair (Domain, Type, Protocol, SocketVector[0])
int Domain, Type, Protocol;
int SocketVector[2];
```

Description

The **socketpair** subroutine creates an unnamed pair of connected sockets in a specified domain, of a specified type, and using the optionally specified protocol. The two sockets are identical.

Note: Create sockets with this subroutine only in the **AF_UNIX** protocol family.

The descriptors used in referencing the new sockets are returned in the *SocketVector*[**0**] and *SocketVector*[**1**] parameters.

The /**usr**/**include**/**sys**/**socket.h** file contains the definitions for socket domains, types, and protocols.

Parameters

Domain	Specifies the communications domain within which the sockets are created. This subroutine does not create sockets in the Internet domain.
Type	Specifies the communications method, whether **SOCK_DGRAM** or **SOCK_STREAM**, that the socket uses.
Protocol	Points to an optional identifier used to specify which standard set of rules (such as UDP/IP and TCP/IP) governs the transfer of data.
SocketVector	Points to a two–element vector that contains the integer descriptors of a pair of created sockets.

Return Values

Upon successful completion, the **socketpair** subroutine returns a value of 0.

If the **socketpair** subroutine is unsuccessful, the subroutine handler performs the following functions:

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

Error Codes

If the **socketpair** subroutine is unsuccessful, it returns one of the following errors codes:

EMFILE	This process has too many descriptors in use.
EAFNOSUPPORT	The addresses in the specified address family cannot be used with this socket.
EPROTONOSUPPORT	The specified protocol cannot be used on this system.
EOPNOTSUPP	The specified protocol does not allow the creation of socket pairs.
EFAULT	The *SocketVector* parameter is not in a writable part of the user address space.

Implementation Specifics

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

All applications containing the **socketpair** subroutine must be compiled with **_BSD** set to a value of 43 or 44. Socket applications must include the BSD **libbsd.a** library.

Related Information

The **socket** subroutine.

Socketpair Communication Example Program, Sockets Overview, and Understanding Socket Creation in *AIX Communications Programming Concepts*.

Chapter 11. Streams

adjmsg Utility

Purpose

Trims bytes in a message.

Syntax

```
int adjmsg (mp, len)
mblk_t *mp;
register int len;
```

Description

The **adjmsg** utility trims bytes from either the head or tail of the message specified by the *mp* parameter. It only trims bytes across message blocks of the same type. The **adjmsg** utility is unsuccessful if the *mp* parameter points to a message containing fewer than *len* bytes of similar type at the message position indicated.

Parameters

mp	Specifies the message to be trimmed.
len	Specifies the number of bytes to remove from the message.

If the value of the *len* parameter is greater than 0, the **adjmsg** utility removes the number of bytes specified by the *len* parameter from the beginning of the *mp* message. If the value of the *len* parameter is less than 0, it removes *len* bytes from the end of the *mp* message. If the value of the *len* parameter is 0, the **adjmsg** utility does nothing.

Return Values

On successful completion, the **adjmsg** utility returns a value of 1. Otherwise, it returns a value of 0.

Implementation Specifics

This utility is part of STREAMS Kernel Extensions.

Related Information

The **msgdsize** utility.

List of Streams Programming References and Understanding STREAMS Messages in *AIX Communications Programming Concepts*.

allocb Utility

Purpose

Allocates message and data blocks.

Syntax

```
struct msgb *
allocb(size, pri)
register int size;
uint pri;
```

Description

The **allocb** utility allocates blocks for a message. When a message is allocated in this manner, the `b_band` field of the **mblk_t** structure is initially set to a value of 0. Modules and drivers can set this field.

Parameters

size Specifies the minimum number of bytes needed in the data buffer.

pri Specifies the relative importance of the allocated blocks to the module. The possible values are:

- **BPRI_LO**
- **BPRI_MED**
- **BPRI_HI**

Return Values

The **allocb** utility returns a pointer to a message block of type **M_DATA** in which the data buffer contains at least the number of bytes specified by the *size* parameter. If a block cannot be allocated as requested, the **allocb** utility returns a null pointer.

Implementation Specifics

This utility is part of STREAMS Kernel Extensions.

Related Information

The **esballoc** utility.

List of Streams Programming References and Understanding STREAMS Messages in *AIX Communications Programming Concepts*.

backq Utility

Purpose

Returns a pointer to the queue behind a given queue.

Syntax

```
queue_t *
backq(q)
register queue_t *q;
```

Description

The **backq** utility returns a pointer to the queue preceding a given queue. If no such queue exists (as when the *q* parameter points to a stream end), the **backq** utility returns a null pointer.

Parameters

q	Specifies the queue from which to begin.

Return Values

The **backq** utility returns a pointer to the queue behind a given queue. If no such queue exists, the **backq** utility returns a null pointer.

Implementation Specifics

This utility is part of STREAMS Kernel Extensions.

Related Information

The **RD** utility, **WR** utility.

List of Streams Programming References and Understanding STREAMS Messages in *AIX Communications Programming Concepts*.

bcanput Utility

Purpose

Tests for flow control in the given priority band.

Syntax

```
int
bcanput(q, pri)
register queue_t *q;
unsigned char pri;
```

Description

The **bcanput** utility provides modules and drivers with a way to test flow control in the given priority band.

The **bcanput (q, 0)** call is equivalent to the **canput (q)** call.

Parameters

q	Specifies the queue from which to begin to test.
pri	Specifies the priority band to test.

Return Values

The **bcanput** utility returns a value of 1 if a message of the specified priority can be placed on the queue. It returns a value of 0 if the priority band is flow–controlled and sets the **QWANTW** flag to 0 band (the **QB_WANTW** flag is set to nonzero band). If the band does not yet exist on the queue in question, it returns a value of 1.

Implementation Specifics

This utility is part of STREAMS Kernel Extensions.

Related Information

List of Streams Programming References, Understanding STREAMS Flow Control in *AIX Communications Programming Concepts*.

bufcall Utility

Purpose

Recovers from a failure of the **allocb** utility.

Syntax

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

int
bufcall(size, pri, func, arg)
uint size;
int pri;
void (*func)();
long arg;
```

Description

The **bufcall** utility assists in the event of a block–allocation failure. If the **allocb** utility returns a null, indicating a message block is not currently available, the **bufcall** utility may be invoked.

The **bufcall** utility arranges for (**func*)(*arg*) call to be made when a buffer of the number of bytes specified by the *size* parameter is available. The *pri* parameter is as described in the **allocb** utility. When the function specified by the *func* parameter is called, it has no user context. It cannot reference the **u_area** and must return without sleeping. The **bufcall** utility does not guarantee that the desired buffer will be available when the function specified by the *func* parameter is called since interrupt processing may acquire it.

On an unsuccessful return, the function specified by the *func* parameter will never be called. A failure indicates a temporary inability to allocate required internal data structures.

On multiprocessor systems, the function specified by the *func* parameter should be interrupt–safe. Otherwise, the **STR_QSAFETY** flag must be set when installing the module or driver with the **str_install** utility.

Note: The **stream.h** header file must be the last included header file of each source file using the stream library.

Parameters

size Specifies the number of bytes needed.

pri Specifies the relative importance of the allocated blocks to the module. The possible values are:

- **BPRI_LO**
- **BPRI_MED**
- **BPRI_HI**

func Specifies the function to be called.

arg Specifies an argument passed to the function.

Return Values

The **bufcall** utility returns a value of 1 when the request is successfully recorded. Otherwise, it returns a value of 0.

Implementation Specifics

This utility is part of STREAMS Kernel Extensions.

Related Information

The **allocb** utility, **unbufcall** utility, **mi_bufcall** utility.

List of Streams Programming References and Understanding STREAMS Messages in *AIX Communications Programming Concepts*.

Understanding STREAMS Synchronization in *AIX Communications Programming Concepts*.

canput Utility

Purpose

Tests for available room in a queue.

Syntax

```
int
canput(q)
register queue_t *q;
```

Description

The **canput** utility determines if there is room left in a message queue. If the queue does not have a service procedure, the **canput** utility searches farther in the same direction in the stream until it finds a queue containing a service procedure. This is the first queue on which the passed message can actually be queued. If such a queue cannot be found, the search terminates on the queue at the end of the stream.

The **canput** utility only takes into account normal data flow control.

Parameters

q	Specifies the queue at which to begin the search.

Return Values

The **canput** utility tests the queue found by the search. If the message queue in this queue is not full, the **canput** utility returns a value of 1. This return indicates that a message can be put to the queue. If the message queue is full, the **canput** utility returns a value of 0. In this case, the caller is generally referred to as ”blocked”.

Implementation Specifics

This utility is part of STREAMS Kernel Extensions.

Related Information

List of Streams Programming References, Understanding STREAMS Messages in *AIX Communications Programming Concepts*.

clone Device Driver

Purpose

Opens an unused minor device on another STREAMS driver.

Description

The **clone** device driver is a STREAMS software driver that finds and opens an unused minor device on another STREAMS driver. The minor device passed to the **clone** device driver during the open routine is interpreted as the major device number of another STREAMS driver for which an unused minor device is to be obtained. Each such open operation results in a separate stream to a previously unused minor device.

The **clone** device driver consists solely of an **open** subroutine. This open function performs all of the necessary work so that subsequent subroutine calls (including the **close** subroutine) require no further involvement of the **clone** device driver.

The **clone** device driver generates an **ENXIO** error, without opening the device, if the minor device number provided does not correspond to a valid major device, or if the driver indicated is not a STREAMS driver.

> **Note:** Multiple opens of the same minor device cannot be done through the **clone** interface. Executing the **stat** subroutine on the file system node for a cloned device yields a different result from executing the **fstat** subroutine using a file descriptor obtained from opening the node.

Related Information

The **close** subroutine, **fstat** subroutine, **open** subroutine, **stat** subroutine.

Understanding STREAMS Drivers and Modules and Understanding the log Device Driver in *AIX Communications Programming Concepts*.

copyb Utility

Purpose

Copies a message block.

Syntax

```
mblk_t *
copyb(bp)
register mblk_t *bp;
```

Description

The **copyb** utility copies the contents of the message block pointed to by the *bp* parameter into a newly allocated message block of at least the same size. The **copyb** utility allocates a new block by calling the **allocb** utility. All data between the `b_rptr` and `b_wptr` pointers of a message block are copied to the new block, and these pointers in the new block are given the same offset values they had in the original message block.

Parameters

bp	Contains a pointer to the message block to be copied.

Return Values

On successful completion, the **copyb** utility returns a pointer to the new message block containing the copied data. Otherwise, it returns a null value. The copy is rounded to a fullword boundary.

Implementation Specifics

This utility is part of STREAMS Kernel Extensions.

Related Information

The **allocb** utility, **copymsg** utility.

List of Streams Programming References and Understanding STREAMS Messages in *AIX Communications Programming Concepts*.

copymsg Utility

Purpose

Copies a message.

Syntax

```
mblk_t *
copymsg(bp)
register mblk_t *bp;
```

Description

The **copymsg** utility uses the **copyb** utility to copy the message blocks contained in the message pointed to by the *bp* parameter to newly allocated message blocks. It then links the new message blocks to form the new message.

Parameters

bp	Contains a pointer to the message to be copied.

Return Values

On successful compilation, the **copymsg** utility returns a pointer to the new message. Otherwise, it returns a null value.

Implementation Specifics

This utility is part of STREAMS Kernel Extensions.

Related Information

The **copyb** utility.

List of Streams Programming References and Understanding STREAMS Messages in *AIX Communications Programming Concepts*.

datamsg Utility

Purpose

Tests whether message is a data message.

Syntax

```
#define datamsg(type) ((type) == M_DATA | | (type) == M_PROTO | |
(type) ==
M_PCPROTO | | (type) == M_DELAY)
```

Description

The **datamsg** utility determines if a message is a data–type message. It returns a value of True if `mp->b_datap->db_type` (where `mp` is declared as **mblk_t *mp**) is a data–type message. The possible data types are **M_DATA**, **M_PROTO**, **M_PCPROTO**, and **M_DELAY**.

Parameters

type	Specifies acceptable data types.

Return Values

The **datamsg** utility returns a value of True if the message is a data–type message. Otherwise, it returns a value of False.

Implementation Specifics

This utility is part of STREAMS Kernel Extensions.

Related Information

List of Streams Programming References, Understanding STREAMS Messages in *AIX Communications Programming Concepts*.

dlpi STREAMS Driver

Purpose

Provides an interface to the data link provider.

Description

The **dlpi** driver is a STREAMS–based pseudo–driver that provides a Data Link Provider Interface (DLPI) style 2 interface to the data link providers in the operating system.

The data link provider interface supports both the connectionless and connection–oriented modes of service, using the DL_UNITDATA_REQ and DL_UNITDATA_IND primitives. See Data Link Provider Interface Information in *AIX Communications Programming Concepts*.

Refer to the ”STREAMS Overview” in *AIX Communications Programming Concepts* for related publications about the DLPI.

File System Name

Each provider supported by the **dlpi** driver has a unique name in the file system. The supported interfaces are:

Driver Name	Interface
/dev/dlpi/en	Ethernet
/dev/dlpi/et	802.3
/dev/dlpi/tr	802.5
/dev/dlpi/fi	FDDI

Physical Point of Attachment

The PPA is used to identify one of several of the same type of interface in the system. It should be a nonnegative integer in the range 0 through 99.

The **dlpi** drivers use the network interface drivers to access the communication adapter drivers. For example, the **/dev/dlpi/tr** file uses the network interface driver **if_tr** (interface **tr0**, **tr1**, **tr2**, . . .) to access the token–ring adapter driver. The PPA value used attaches the device open instance with the corresponding network interface. For example, opening to the **/dev/dlpi/en** device and then performing an attach with PPA value of 1 attaches this open instance to the network interface **en1**. Therefore, choosing a PPA value selects a network interface. The specific network interface should be active before a certain PPA value is used.

Examples of client and server **dlpi** programs are located in the **/usr/samples/dlpi** directory.

Files

/dev/dlpi/*	Contains names of supported protocols.
/usr/samples/dlpi	Contains client and server **dlpi** sample programs.

Implementation Specifics

This driver is part of STREAMS Kernel Extensions.

Related Information

The **ifconfig** command, **strload** command.

Understanding STREAMS Drivers and Modules, Obtaining Copies of the DLPI and TPI Specification, Data Link Provider Interface Information, in *AIX Communications Programming Concepts*.

dupb Utility

Purpose

Duplicates a message–block descriptor.

Syntax

```
mblk_t *
dupb(bp)
register mblk_t *bp;
```

Description

The **dupb** utility duplicates the message block descriptor (**mblk_t**) pointed to by the *bp* parameter by copying the descriptor into a newly allocated message–block descriptor. A message block is formed with the new message–block descriptor pointing to the same data block as the original descriptor. The reference count in the data–block descriptor (**dblk_t**) is then incremented. The **dupb** utility does not copy the data buffer, only the message–block descriptor.

Message blocks that exist on different queues can reference the same data block. In general, if the contents of a message block with a reference count greater than 1 are to be modified, the **copymsg** utility should be used to create a new message block. Only the new message block should be modified to ensure that other references to the original message block are not invalidated by unwanted changes.

Parameters

bp	Contains a pointer to the message–block descriptor to be copied.

Return Values

On successful compilation, the **dupb** utility returns a pointer to the new message block. If the **dupb** utility cannot allocate a new message–block descriptor, it returns a null pointer.

Implementation Specifics

This utility is part of STREAMS Kernel Extensions.

Related Information

The **copymsg** utility, **dupmsg** utility.

List of Streams Programming References and Understanding STREAMS Messages in *AIX Communications Programming Concepts*.

dupmsg Utility

Purpose

Duplicates a message.

Syntax

```
mblk_t *
dupmsg(bp)
register mblk_t *bp;
```

Description

The **dupmsg** utility calls the **dupb** utility to duplicate the message pointed to by the *bp* parameter by copying all individual message block descriptors and then linking the new message blocks to form the new message. The **dupmsg** utility does not copy data buffers, only message–block descriptors.

Parameters

bp	Specifies the message to be copied.

Return Values

On successful completion, the **dupmsg** utility returns a pointer to the new message. Otherwise, it returns a null pointer.

Implementation Specifics

This utility is part of STREAMS Kernel Extensions.

Related Information

The **dupb** utility.

List of Streams Programming References and Understanding STREAMS Messages in *AIX Communications Programming Concepts*.

enableok Utility

Purpose

Enables a queue to be scheduled for service.

Syntax

```
void
enableok(q)
queue_t *q;
```

Description

The **enableok** utility cancels the effect of an earlier **noenable** utility on the same queue. It allows a queue to be scheduled for service that had previously been excluded from queue service by a call to the **noenable** utility.

Parameters

q	Specifies the queue to be enabled.

Implementation Specifics

This utility is part of STREAMS Kernel Extensions.

Related Information

The **noenable** utility.

List of Streams Programming References and Understanding STREAMS Messages in *AIX Communications Programming Concepts*.

esballoc Utility

Purpose

Allocates message and data blocks.

Syntax

```
mblk_t *
esballoc(base, size, pri, free_rtn)
unsigned char *base;
int size, pri;
frn_t *free_rtn;
```

Description

The **esballoc** utility allocates message and data blocks that point directly to a client–supplied buffer. The **esballoc** utility sets the `db_base`, `b_rptr`, and `b_wptr` fields to the value specified in the *base* parameter (data buffer size) and the `db_lim` field to the *base* value plus the *size* value. The pointer to the **free_rtn** structure is placed in the `db_freep` field of the data block.

The success of the **esballoc** utility depends on the success of the **allocb** utility and also that the *base, size*, and *free_rtn* parameters are not null. If successful, the **esballoc** utility returns a pointer to a message block. If an error occurs, the **esballoc** utility returns a null pointer.

Parameters

base	Specifies the data buffer size.
size	Specifies the number of bytes.
pri	Specifies the relative importance of this block to the module. The possible values are: • **BPRI_LO** • **BPRI_MED** • **BPRI_HI** The *pri* parameter is currently unused and is maintained only for compatibility with applications developed prior to UNIX System V Release 4.0.
free_rtn	Specifies the function and argument to be called when the message is freed.

Return Values

On successful completion, the **esballoc** utility returns a pointer to a message block. Otherwise, it returns a null pointer.

Implementation Specifics

This utility is part of STREAMS Kernel Extensions.

Related Information

The **allocb** utility.

List of Streams Programming References and Understanding STREAMS Messages in *AIX Communications Programming Concepts*.

flushband Utility

Purpose

Flushes the messages in a given priority band.

Syntax

```
void flushband(q, pri, flag)
register queue_t *q;
unsigned char pri;
int flag;
```

Description

The **flushband** utility provides modules and drivers with the capability to flush the messages associated in a given priority band. The *flag* parameter is defined the same as in the **flushq** utility. Otherwise, messages are flushed from the band specified by the *pri* parameter according to the value of the *flag* parameter.

Parameters

q Specifies the queue to flush.

pri Specifies the priority band to flush. If the value of the *pri* parameter is 0, only ordinary messages are flushed.

flag

Specifies which messages to flush from the queue. Possible values are:

FLUSHDATA Discards all **M_DATA**, **M_PROTO**, **M_PCPROTO**, and **M_DELAY** messages, but leaves all other messages on the queue.

FLUSHALL Discards all messages from the queue.

Implementation Specifics

This utility is part of STREAMS Kernel Extensions.

Related Information

The **flushq** utility.

List of Streams Programming References and Understanding STREAMS Messages in *AIX Communications Programming Concepts*.

flushq Utility

Purpose

Flushes a queue.

Syntax

```
void flushq(q, flag)
register queue_t *q;
int flag;
```

Description

The **flushq** utility removes messages from the message queue specified by the *q* parameter and then frees them using the **freemsg** utility.

If a queue behind the *q* parameter is blocked, the **flushq** utility may enable the blocked queue, as described in the **putq** utility.

Parameters

q Specifies the queue to flush.

flag

Specifies the types of messages to flush. Possible values are:

FLUSHDATA Discards all **M_DATA**, **M_PROTO**, **M_PCPROTO**, and **M_DELAY** messages, but leaves all other messages on the queue.

FLUSHALL Discards all messages from the queue.

Implementation Specifics

This utility is part of STREAMS Kernel Extensions.

Related Information

The **freemsg** utility, **putq** utility.

List of Streams Programming References and Understanding STREAMS Messages in *AIX Communications Programming Concepts*.

freeb Utility

Purpose

Frees a single message block.

Syntax

```
void freeb(bp)
register struct msgb *bp;
```

Description

The **freeb** utility frees (deallocate) the message–block descriptor pointed to by the *bp* parameter. It also frees the corresponding data block if the reference count (see the **dupb** utility) in the data–block descriptor (**datab** structure) is equal to 1. If the reference count is greater than 1, the **freeb** utility does not free the data block, but decrements the reference count instead.

If the reference count is 1 and if the message was allocated by the **esballoc** utility, the function specified by the `db_frtnp->free_func` pointer is called with the parameter specified by the `db_frtnp->free_arg` pointer.

The **freeb** utility cannot be used to free a multiple–block message (see the **freemsg** utility). Results are unpredictable if the **freeb** utility is called with a null argument. Always ensure that the pointer is nonnull before using the **freeb** utility.

Parameters

bp	Contains a pointer to the message–block descriptor that is to be freed.

Implementation Specifics

This utility is part of STREAMS Kernel Extensions.

Related Information

The **dupb** utility, **esballoc** utility, **freemsg** utility.

List of Streams Programming References and Understanding STREAMS Messages in *AIX Communications Programming Concepts*.

freemsg Utility

Purpose

Frees all message blocks in a message.

Syntax

```
void freemsg(bp)
register mblk_t *bp;
```

Description

The **freemsg** utility uses the **freeb** utility to free all message blocks and their corresponding data blocks for the message pointed to by the *bp* parameter.

Parameters

bp	Contains a pointer to the message that is to be freed.

Implementation Specifics

This utility is part of STREAMS Kernel Extensions.

Related Information

The **freeb** utility.

List of Streams Programming References and Understanding STREAMS Messages in *AIX Communications Programming Concepts*.

getadmin Utility

Purpose

Returns a pointer to a module.

Syntax

```
int
(*getadmin(mid))()
ushort mid;
```

Description

The **getadmin** utility returns a pointer to the module identified by the *mid* parameter.

Parameters

mid	Identifies the module to locate.

Return Values

On successful completion, the **getadmin** utility returns a pointer to the specified module. Otherwise, it returns a null pointer.

Implementation Specifics

This utility is part of STREAMS Kernel Extensions.

Related Information

List of Streams Programming References, Understanding STREAMS Drivers and Modules in *AIX Communications Programming Concepts*.

getmid Utility

Purpose

Returns a module ID.

Syntax

```
ushort
getmid(name)
char name;
```

Description

The **getmid** utility returns the module ID for the module identified by the *name* parameter.

Parameters

name	Specifies the module to be identified.

Return Values

On successful completion, the **getmid** utility returns the module ID. Otherwise, it returns a value of 0.

Implementation Specifics

This utility is part of STREAMS Kernel Extensions.

Related Information

List of Streams Programming References, Understanding STREAMS Drivers and Modules in *AIX Communications Programming Concepts*.

getmsg System Call

Purpose

Gets the next message off a stream.

Syntax

```
#include <stropts.h>

int getmsg (fd, ctlptr, dataptr, flags)
int fd;
struct strbuf *ctlptr;
struct strbuf *dataptr;
int *flags;
```

Description

The **getmsg** system call retrieves from a STREAMS file the contents of a message located at the stream–head read queue, and places the contents into user–specified buffers. The message must contain either a data part, a control part, or both. The data and control parts of the message are placed into separate buffers, as described in the ”Parameters” section. The semantics of each part are defined by the STREAMS module that generated the message.

Note: To use the **getmsg** system call you must import the /**lib**/**pse.exp** file during the compile. For example, compile the system call by entering the following command:

```
cc -bI:/lib/pse.exp
```

Failure to import this file means that the **getmsg** system call will be unresolved during the link edit phase.

Parameters

fd	Specifies a file descriptor referencing an open stream.
ctlptr	Holds the control part of the message.
dataptr	Holds the data part of the message.
flags	Indicates the type of message to be retrieved. Acceptable values are:

0	Process the next message of any type.
RS_HIPRI	Process the next message only if it is a priority message.

The *ctlptr* and *dataptr* parameters each point to a **strbuf** structure that contains the following members:

```
int maxlen;  /* maximum buffer length */
int len;     /* length of data  */
char *buf;   /* ptr to buffer  */
```

In the **strbuf** structure, the `maxlen` field indicates the maximum number of bytes this buffer can hold, the `len` field contains the number of bytes of data or control information received, and the `buf` field points to a buffer in which the data or control information is to be placed.

If the *ctlptr* (or *dataptr*) parameter is null or the `maxlen` field is –1, the following events occur:

- The control part of the message is not processed. Thus, it is left on the stream–head read queue.

- The `len` field is set to –1.

If the `maxlen` field is set to 0 and there is a zero–length control (or data) part, the following events occur:

- The zero–length part is removed from the read queue.
- The `len` field is set to 0.

If the `maxlen` field is set to 0 and there are more than 0 bytes of control (or data) information, the following events occur:

- The information is left on the read queue.
- The `len` field is set to 0.

If the `maxlen` field in the *ctlptr* or *dataptr* parameter is less than, respectively, the control or data part of the message, the following events occur:

- The `maxlen` bytes are retrieved.
- The remainder of the message is left on the stream–head read queue.
- A nonzero return value is provided.

By default, the **getmsg** system call processes the first priority or nonpriority message available on the stream–head read queue. However, a user may choose to retrieve only priority messages by setting the *flags* parameter to **RS_HIPRI**. In this case, the **getmsg** system call processes the next message only if it is a priority message.

If the **O_NDELAY** flag has not been set, the **getmsg** system call blocks until a message of the types specified by the *flags* parameter (priority only or either type) is available on the stream–head read queue. If the **O_DELAY** flag has been set and a message of the specified types is not present on the read queue, the **getmsg** system call fails and sets the **errno** global variable to **EAGAIN**.

If a hangup occurs on the stream from which messages are to be retrieved, the **getmsg** system call continues to operate until the stream–head read queue is empty. Thereafter, it returns 0 in the `len` fields of both the *ctlptr* and *dataptr* parameters.

Return Values

Upon successful completion, the **getmsg** system call returns a nonnegative value. The possible values are:

0	Indicates that a full message was read successfully.
MORECTL	Indicates that more control information is waiting for retrieval.
MOREDATA	Indicates that more data is waiting for retrieval.
MORECTL|.MOREDATA	Indicates that both types of information remain. Subsequent **getmsg** calls retrieve the remainder of the message.

On return, the `len` field contains one of the following:

- The number of bytes of control information or data actually received
- 0 if there is a zero–length control or data part
- –1 if no control information or data is present in the message.

If information is retrieved from a priority message, the *flags* parameter is set to **RS_HIPRI** on return.

Error Codes

The **getmsg** system call fails if one or more of the following is true:

EAGAIN	The **O_NDELAY** flag is set, and no messages are available.
EBADF	The *fd* parameter is not a valid file descriptor open for reading.
EBADMSG	Queued message to be read is not valid for the **getmsg** system call.
EFAULT	The *ctlptr, dataptr,* or *flags* parameter points to a location outside the allocated address space.
EINTR	A signal was caught during the **getmsg** system call.
EINVAL	An illegal value was specified in the *flags* parameter or else the stream referenced by the *fd* parameter is linked under a multiplexer.
ENOSTR	A stream is not associated with the *fd* parameter.

The **getmsg** system call can also fail if a STREAMS error message had been received at the stream head before the call to the **getmsg** system call. The error returned is the value contained in the STREAMS error message.

Implementation Specifics

This system call is part of the STREAMS Kernel Extensions.

Files

/lib/pse.exp	Contains the STREAMS export symbols.

Related Information

The **poll** subroutine, **read** subroutine, **write** subroutine.

The **getpmsg** system call, **putmsg** system call, **putpmsg** system call.

List of Streams Programming References and STREAMS Overview in *AIX Communications Programming Concepts*.

getpmsg System Call

Purpose

Gets the next priority message off a stream.

Syntax

```
#include <stropts.h>

int getpmsg (fd, ctlptr, dataptr, bandp, flags)
int fd;
struct strbuf *ctlptr;
struct strbuf *dataptr;
int *bandp;
int *flags;
```

Description

The **getpmsg** system call is identical to the **getmsg** system call, except that the message priority can be specified.

Parameters

fd Specifies a file descriptor referencing an open stream.

ctlptr Holds the control part of the message.

dataptr Holds the data part of the message.

bandp Specifies the priority band of the message. If the value of the *bandp* parameter is set to 0, then the priority band is not limited.

flags

Indicates the type of message priority to be retrieved. Acceptable values are:

MSG_ANY Process the next message of any type.

MSG_BAND Process the next message only if it is of the specified priority band.

MSG_HIPRI Process the next message only if it is a priority message.

If the value of the *flags* parameter is **MSG_ANY** or **MSG_HIPRI**, then the *bandp* parameter must be set to 0.

Implementation Specifics

This system call is part of the STREAMS Kernel Extensions.

Related Information

The **poll** subroutine, **read** subroutine, **write** subroutine.

The **getmsg** system call, **putmsg** system call, **putpmsg** system call.

List of Streams Programming References and STREAMS Overview in *AIX Communications Programming Concepts*.

getq Utility

Purpose

Gets a message from a queue.

Syntax

```
mblk_t *
getq(q)
register queue_t *q;
```

Description

The **getq** utility gets the next available message from the queue pointed to by the *q* parameter. The **getq** utility returns a pointer to the message and removes that message from the queue. If no message is queued, the **getq** utility returns null.

The **getq** utility, and certain other utility routines, affect flow control in the Stream as follows: If the **getq** utility returns null, the queue is marked with the **QWANTR** flag so that the next time a message is placed on it, it will be scheduled for service (that is, enabled – see the **qenable** utility). If the data in the enqueued messages in the queue drops below the low–water mark, as specified by the `q_lowat` field, and if a queue behind the current queue has previously attempted to place a message in the queue and failed, (that is, was blocked – see the **canput** utility), then the queue behind the current queue is scheduled for service.

The queue count is maintained on a per–band basis. Priority band 0 (normal messages) uses the `q_count` and `q_lowat` fields. Nonzero priority bands use the fields in their respective **qband** structures (the `qb_count` and `qb_lowat` fields). All messages appear on the same list, linked according to their `b_next` pointers.

The `q_count` field does not reflect the size of all messages on the queue; it only reflects those messages in the normal band of flow.

Parameters

q	Specifies the queue from which to get the message.

Return Values

On successful completion, the **getq** utility returns a pointer to the message. Otherwise, it returns a null value.

Implementation Specifics

This utility is part of STREAMS Kernel Extensions.

Related Information

The **canput** utility, **qenable** utility, **rmvq** utility.

List of Streams Programming References and Understanding STREAMS Messages in *AIX Communications Programming Concepts*.

insq Utility

Purpose

Puts a message at a specific place in a queue.

Syntax

```
int
insq(q, emp, mp)
register queue_t *q;
register mblk_t *emp;
register mblk_t *mp;
```

Description

The **insq** utility places the message pointed to by the *mp* parameter in the message queue pointed to by the *q* parameter, immediately before the already–queued message pointed to by the *emp* parameter.

If an attempt is made to insert a message out of order in a queue by using the **insq** utility, the message will not be inserted and the routine is not successful.

The queue class of the new message is ignored. However, the priority band of the new message must adhere to the following format:

```
emp->b_prev->b_band >= mp->b_band >= emp->b_band.
```

Parameters

q	Specifies the queue on which to place the message.
emp	Specifies the existing message before which the new message is to be placed. If the *emp* parameter has a value of null, the message is placed at the end of the queue. If the *emp* parameter is nonnull, it must point to a message that exists on the queue specified by the *q* parameter, or undesirable results could occur.
mp	Specifies the message that is to be inserted on the queue.

Return Values

On successful completion, the **insq** utility returns a value of 1. Otherwise, it returns a value of 0.

Implementation Specifics

This utility is part of STREAMS Kernel Extensions.

Related Information

The **getq** utility.

List of Streams Programming References and Understanding STREAMS Messages in *AIX Communications Programming Concepts*.

isastream Function

Purpose

Tests a file descriptor.

Library

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

Syntax

```
int isastream(int fildes);
```

Description

The **isastream** subroutine determines if a file descriptor represents a STREAMS file.

Parameters

fildes	Specifies which open file to check.

Return Values

On successful completion, the **isastream** subroutine returns a value of 1 if the *fildes* parameter represents a STREAMS file, or a value of 0 if not. Otherwise, it returns a value of –1 and sets the **errno** global variable to indicate the error.

Error Codes

If unsuccessful, the **errno** global variable is set to the following value:

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

Implementation Specifics

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

Related Information

streamio operations.

List of Streams Programming References in *AIX Communications Programming Concepts*.

linkb Utility

Purpose

Concatenates two messages into one.

Syntax

```
void link(mp, bp)
register mblk_t *mp;
register mblk_t *bp;
```

Description

The **linkb** utility puts the message pointed to by the *bp* parameter at the tail of the message pointed to by the *mp* parameter. This results in a single message.

Parameters

mp	Specifies the message to which the second message is to be linked.
bp	Specifies the message that is to be linked to the end of first message.

Implementation Specifics

This utility is part of STREAMS Kernel Extensions.

Related Information

The **unlinkb** utility.

List of Streams Programming References and Understanding STREAMS Messages in *AIX Communications Programming Concepts*.

mi_bufcall Utility

Purpose

Provides a reliable alternative to the **bufcall** utility.

Syntax

```
#include <pse/mi.h>
#include <sys/stream.h>

void mi_bufcall (Queue, Size, Priority)
queue_t *Queue;
int Size;
int Priortity;
```

Description

The **mi_bufcall** utility provides a reliable alternative to the **bufcall** utility. The standard STREAMS **bufcall** utility is intended to be called when the **allocb** utility is unable to allocate a block for a message, and invokes a specified callback function (typically the **qenable** utility) with a given queue when a large enough block becomes available. This can cause system problems if the stream closes so that the queue becomes invalid before the callback function is invoked.

The **mi_bufcall** utility is a reliable alternative, as the queue is not deallocated until the call is complete. This utility uses the standard **bufcall** mechanism with its own internal callback routine. The callback routine either invokes the **qenable** utility with the specified *Queue* parameter, or simply deallocates the instance data associated with the stream if the queue has already been closed.

Note: The **stream.h** header file must be the last included header file of each source file using the stream library.

Parameters

Queue	Specifies the queue which is to be passed to the **qenable** utility.
Size	Specifies the required buffer size.
Priority	Specifies the priority as used by the standard STREAMS **bufcall** mechanism.

Implementation Specifics

The **mi_bufcall** utility is part of STREAMS kernel extensions.

Related Information

List of Streams Programming References in *AIX Communications Programming Concepts* .

STREAMS Overview in *AIX Communications Programming Concepts* .

The **bufcall** utility, **mi_close_comm** utility, **mi_next_ptr** utility, **mi_open_comm** utility.

mi_close_comm Utility

Purpose

Performs housekeeping during STREAMS driver or module close operations.

Syntax

```
#include <pse/mi.h>
#include <sys/stream.h>

int mi_close_comm (StaticPointer, Queue)
caddr_t *StaticPointer;
queue_t *Queue;
```

Description

The **mi_close_comm** utility performs housekeeping during STREAMS driver or module close operations. It is intended to be called by the driver or module **close** routine. It releases the memory allocated by the corresponding call to the **mi_open_comm** utility, and frees the minor number for reuse.

If an **mi_bufcall** operation is outstanding, module resources are not freed until the **mi_buffcall** operation is complete.

Notes:

1. Each call to the **mi_close_comm** utility must have a corresponding call to the **mi_open_comm** utility. Executing one of these utilities without making a corresponding call to the other will lead to unpredictable results.
2. The **stream.h** header file must be the last included header file of each source file using the stream library.

Parameters

StaticPointer	Specifies the address of the static pointer which was passed to the corresponding call to the **mi_open_comm** utility to store the address of the module's list of open streams.
Queue	Specifies the *Queue* parameter which was passed to the corresponding call to the **mi_open_comm** utility.

Return Values

The **mi_close_comm** utility always returns a value of zero.

Implementation Specifics

The **mi_close_comm** utility is part of STREAMS kernel extensions.

Related Information

List of Streams Programming References in *AIX Communications Programming Concepts* .

STREAMS Overview in *AIX Communications Programming Concepts* .

The **mi_open_comm** utility, **mi_next_ptr** utility, **mi_bufcall** utility.

mi_next_ptr Utility

Purpose

Traverses a STREAMS module's linked list of open streams.

Syntax

```
#include <pse/mi.h>
#include <sys/stream.h>

caddr_t mi_next_ptr (Origin)
caddr_t Origin;
```

Description

The **mi_next_ptr** utility traverses a module's linked list of open streams. The *Origin* argument specifies the address of a per–instance list item, and the return value indicates the address of the next item. The first time the **mi_next_ptr** utility is called, the *Origin* parameter should be initialized with the value of the static pointer which was passed to the **mi_open_comm** utility. Subsequent calls to the **mi_next_ptr** utility should pass the address which was returned by the previous call, until a **NULL** address is returned, indicating that the end of the queue has been reached.

Note: The **stream.h** header file must be the last included header file of each source file using the stream library.

Parameter

Origin	Specifies the address of the current list item being examined.

Return Values

The **mi_next_ptr** utility returns the address of the next list item, or **NULL** if the end of the list has been reached.

Implementation Specifics

The **mi_next_ptr** utility is part of STREAMS kernel extensions.

Related Information

List of Streams Programming References in *AIX Communications Programming Concepts* .

STREAMS Overview in *AIX Communications Programming Concepts* .

The **mi_close_comm** utility, **mi_open_comm** utility, **mi_bufcall** utility.

mi_open_comm Utility

Purpose

Performs housekeeping during STREAMS driver or module open operations.

Syntax

```
#include <pse/mi.h>
#include <sys/stream.h>

int mi_open_comm (StaticPointer, Size, Queue, Device, Flag,
SFlag, credp)
caddr_t *StaticPointer;
uint Size;
queue_t *Queue;
dev_t *Device;
int Flag;
int SFlag;
cred_t *credp;
```

Description

The **mi_open_comm** subroutine performs housekeeping during STREAMS driver or module open operations. It is intended to be called by the driver or module **open** routine. It assigns a minor device number to the stream (as specified by the *SFlag* parameter), allocates the requested per–stream data, and sets the **q_ptr** fields of the stream being opened.

Notes:

1. Each call to the **mi_open_comm** subroutine must have a corresponding call to the **mi_close_comm** subroutine. Executing one of these utilities without making a corresponding call to the other will lead to unpredictable results.
2. The **stream.h** header file must be the last included header file of each source file using the stream library.

Parameters

StaticPointer	Specifies the address of a static pointer which will be used internally by the **mi_open_comm** and related utilities to store the address of the module's list of open streams. This pointer should be initialized to **NULL**.
Size	Specifies the amount of memory the module needs for its per–stream data. It is usually the size of the local structure which contains the module's instance data.
Queue	Specifies the address of a **queue_t** structure. The **q_ptr** field of the of this structure, and of the corresponding read queue structure (if *Queue* points to a write queue) or write queue structure (if *Queue* points to a read queue), are filled in with the address of the **queue_t** structure being initialized.
Device	Specifies the address of a **dev_t** structure. The use of this parameter depends on the value of the *SFlag* parameter.
Flag	Unused.

SFlag

Specifies how the *Device* parameter is to be used. The *SFlag* parameter may take one of the following values:

DEVOPEN The minor device number specified by the *Device* argument is used.

MODOPEN The *Device* parameter is **NULL**. This value should be used if the **mi_open_com** subroutine is called from the open routine of a STREAMS module rather than a STREAMS driver.

CLONEOPEN A unique minor device number above 5 is assigned (minor numbers 0–5 are reserved as special access codes).

credp Unused

Return Values

On successful completion, the **mi_open_comm** subroutine returns a value of zero, otherwise one of the following codes is returned:

ENXIO Indicates an invalid parameter.

EAGAIN Indicates that an internal structure could not be allocated, and that the call should be retried.

Implementation Specifics

The **mi_open_comm** subroutine is part of STREAMS kernel extensions.

Related Information

List of Streams Programming References in *AIX Communications Programming Concepts* .

STREAMS Overview in *AIX Communications Programming Concepts* .

The **mi_close_comm** subroutine, **mi_next_ptr** subroutine, **mi_bufcall** subroutine.

msgdsize Utility

Purpose

Gets the number of data bytes in a message.

Syntax

```
int
msgdsize(bp)
register mblk_t *bp;
```

Description

The **msgdsize** utility returns the number of bytes of data in the message pointed to by the *bp* parameter. Only bytes included in data blocks of type **M_DATA** are included in the total.

Parameters

bp	Specifies the message from which to get the number of bytes.

Return Values

The **msgdsize** utility returns the number of bytes of data in a message.

Implementation Specifics

This utility is part of STREAMS Kernel Extensions.

Related Information

List of Streams Programming References, Understanding STREAMS Messages in *AIX Communications Programming Concepts*.

noenable Utility

Purpose

Prevents a queue from being scheduled.

Syntax

```
void noenable(q)
queue_t *q;
```

Description

The **noenable** utility prevents the queue specified by the *q* parameter from being scheduled for service either by the **putq** or **putbq** utility, when these routines queue an ordinary priority message, or by the **insq** utility when it queues any message. The **noenable** utility does not prevent the scheduling of queues when a high–priority message is queued, unless the message is queued by the **insq** utility.

Parameters

q	Specifies the queue to disable.

Implementation Specifics

This utility is part of STREAMS Kernel Extensions.

Related Information

The **enableok** utility, **insq** utility, **putbq** utility, **putq** utility.

List of Streams Programming References and Understanding STREAMS Messages in *AIX Communications Programming Concepts*.

OTHERQ Utility

Purpose

Returns the pointer to the mate queue.

Syntax

```
#define OTHERQ(q) ((q)->flag&QREADER? (q)+1: (q)-1)
```

Description

The **OTHERQ** utility returns a pointer to the mate queue of the *q* parameter.

Parameters

q	Specifies that queue whose mate is to be returned.

Return Values

If the *q* parameter specifies the read queue for the module, the **OTHERQ** utility returns a pointer to the module's write queue. If the *q* parameter specifies the write queue for the module, this utility returns a pointer to the read queue.

Implementation Specifics

This utility is part of STREAMS Kernel Extensions.

Related Information

The **RD** utility, **WR** utility.

List of Streams Programming References and Understanding STREAMS Messages in *AIX Communications Programming Concepts*.

pfmod Packet Filter Module

Purpose

Selectively removes upstream data messages on a Stream.

Synopsis

```
#include <stropts.h>
#include <sys/pfmod.h>

ioctl(fd, I_PUSH, ”pfmod”);
```

Description

The pfmod module implements a programmable packet filter facility that may be pushed over any stream. Every data message that pfmod receives on its read side is subjected to a filter program. If the filter program accepts a message, it will be passed along upstream, and will otherwise be freed. If no filter program has been set (as is the case when pfmod is first pushed), all messages are accepted. Non–data messages (for example, M_FLUSH, M_PCPROTO, M_IOCACK) are never examined and always accepted. The write side is not filtered.

Data messages are defined as either M_PROTO or M_DATA. If an M_PROTO message is received, pfmod will skip over all the leading blocks until it finds an M_DATA block. If none is found, the message is accepted. The M_DATA portion of the message is then made contiguous with pullupmsg(), if necessary, to ensure the data area referenced by the filter program can be accessed in a single mblk_t.

IOCTLs

The following ioctls are defined for this module. All other ioctls are passed downstream without examination.

PFIOCSETF

Install a new filter program, replacing any previous program. It uses the following data structure:

```
typedef struct packetfilt {
      uchar   Pf_Priority;
      uchar   Pf_FilterLen;
      ushort  Pf_Filter[MAXFILTERS];
} pfilter_t;
```

`Pf_Priority` is currently ignored, and should be set to zero. `Pf_FilterLen` indicates the number of shortwords in the `Pf_Filter` array. `Pf_Filter` is an array of shortwords that comprise the filter program. See ”Filters” for details on how to write filter programs.

This ioctl may be issued either transparently or as an I_STR. It will return 0 on success, or –1 on failure, and set errno to one of:

ERANGE	The length of the M_IOCTL message data was not exactly size of (pfilter_t). The data structure is not variable length, although the filter program is.
EFAULT	The ioctl argument points out of bounds.

Filters

A filter program consists of a linear array of shortword instructions. These instructions operate upon a stack of shortwords. Flow of control is strictly linear; there are no branches or loops. When the filter program completes, the top of the stack is examined. If it is non–zero, or if the stack is empty, the packet being examined is passed upstream (accepted), otherwise the packet is freed (rejected).

Instructions are composed of three portions: push command PF_CMD(), argument PF_ARG(), and operation PF_OP(). Each instruction optionally pushes a shortword onto the stack, then optionally performs a binary operation on the top two elements on the stack, leaving its result on the stack. If there are not at least two elements on the stack, the operation will immediately fail and the packet will be rejected. The argument portion is used only by certain push commands, as documented below.

The following push commands are defined:

PF_NOPUSH	Nothing is pushed onto the stack.
PF_PUSHZERO	Pushes 0x0000.
PF_PUSHONE	Pushes 0x0001.
PF_PUSHFFFF	Pushes 0xffff.
PF_PUSHFF00	Pushes 0xff00.
PF_PUSH00FF	Pushes 0x00ff.
PF_PUSHLIT	Pushes the next shortword in the filter program as literal data. Execution resumes with the next shortword after the literal data.
PF_PUSHWORD+N	Pushes shortword N of the message onto the data stack. N must be in the range 0–255, as enforced by the macro PF_ARG().

The following operations are defined. Each operation pops the top two elements from the stack, and pushes the result of the operation onto the stack. The operations below are described in terms of v1 and v2. The top of stack is popped into v2, then the new top of stack is popped into v1. The result of v1 op v2 is then pushed onto the stack.

PF_NOP	The stack is unchanged; nothing is popped.
PF_EQ	v1 == v2
PF_NEQ	v1 != v2
PF_LT	v1 < v2
PF_LE	v1 <= v2
PF_GT	v1 > v2
PF_GE	v1 >= v2
PF_AND	v1 & v2; bitwise
PF_OR	v1 | v2; bitwise
PF_XOR	v1 ^ v2; bitwise

The remaining operations are ”short–circuit” operations. If the condition checked for is found, then the filter program terminates immediately, either accepting or rejecting the packet as specified, without examining the top of stack. If the condition is not found, the filter program continues. These operators do not push any result onto the stack.

PF_COR	If v1 == v2, accept.
PF_CNOR	If v1 == v2, reject.

PF_CAND If v1 != v2, reject.

PF_CNAND If v1 != v2, accept.

If an unknown push command or operation is specified, the filter program terminates immediately and the packet is rejected.

Configuration

Before using pfmod, it must be loaded into the kernel. This may be accomplished with the **strload** command, using the following syntax:

```
strload –m pfmod
```

This command will load the pfmod into the kernel and make it available to I_PUSH. Note that attempting to I_PUSH pfmod before loading it will result in an **EINVAL** error code.

Example

The following program fragment will push pfmod on a stream, then program it to only accept messages with an Ethertype of 0x8137. This example assumes the stream is a promiscuous DLPI ethernet stream (see **dlpi** for details).

```
#include <stddef.h>
#include <sys/types.h>
#include <netinet/if_ether.h>

#define scale(x)        ((x)/sizeof(ushort))

setfilter(int fd)
{
   pfilter_t filter;
    ushort *fp, offset;

    if (ioctl(fd, I_PUSH, ”pfmod”))
               return –1;

    offset = scale(offsetof(struct ether_header, ether_type));
    fp = filter.Pf_Filter;

   /* the filter program */
   *fp++ = PF_PUSHLIT;
   *fp++ = 0x8137;
   *fp++ = PF_PUSHWORD + offset;
   *fp++ = PF_EQ;

   filter.Pf_FilterLen = fp – filter.Pf_Filter;

   if (ioctl(fd, PFIOCSETF, &filter))
              return –1;

   return 0;
}
```

This program may be shortened by combining the operation with the push command:

```
*fp++ = PF_PUSHLIT;
*fp++ = 0x8137;
*fp++ = (PF_PUSHWORD + offset) | PF_EQ;
```

The following filter will accept 802.3 frames addressed to either the Netware raw sap 0xff or the 802.2 sap 0xe0:

```
offset = scale(offsetof(struct ie3_hdr, llc));
*fp++ = PF_PUSHWORD + offset;   /* get ssap, dsap */
*fp++ = PF_PUSH00FF | PF_AND;   /* keep only dsap */
*fp++ = PF_PUSH00FF | PF_COR;   /* is dsap == 0xff? */
*fp++ = PF_PUSHWORD + offset;   /* get ssap, dsap again */
*fp++ = PF_PUSH00FF | PF_AND;   /* keep only dsap */
*fp++ = PF_PUSHLIT  | PF_CAND;  /* is dsap == 0xe0? */
*fp++ = 0x00e0;
```

Note the use of PF_COR in this example. If the dsap is 0xff, then the frame is accepted immediately, without continuing the filter program.

pullupmsg Utility

Purpose

Concatenates and aligns bytes in a message.

Syntax

```
int
pullupmsg(mp, len)
register struct msgb *mp;
register int len;
```

Description

The **pullupmsg** utility concatenates and aligns the number of data bytes specified by the *len* parameter of the passed message into a single, contiguous message block. Proper alignment is hardware–dependent. The **pullupmsg** utility only concatenates across message blocks of similar type. It fails if the *mp* parameter points to a message of less than *len* bytes of similar type. If the *len* parameter contains a value of –1, the **pullupmsg** utility concatenates all blocks of the same type at the beginning of the message pointed to by the *mp* parameter.

As a result of the concatenation, the contents of the message pointed to by the *mp* parameter may be altered.

Parameters

mp	Specifies the message that is to be aligned.
len	Specifies the number of bytes to align.

Return Values

On success, the **pullupmsg** utility returns a value of 1. On failure, it returns a value of 0.

Implementation Specifics

This utility is part of STREAMS Kernel Extensions.

Related Information

List of Streams Programming References, Understanding STREAMS Messages in *AIX Communications Programming Concepts*.

putbq Utility

Purpose

Returns a message to the beginning of a queue.

Syntax

```
int
putbq(q, bp)
register queue_t *q;
register mblk_t *bp;
```

Description

The **putbq** utility puts the message pointed to by the *bp* parameter at the beginning of the queue pointed to by the *q* parameter, in a position in accordance with the message type. High–priority messages are placed at the head of the queue, followed by priority–band messages and ordinary messages. Ordinary messages are placed after all high–priority and priority–band messages, but before all other ordinary messages already on the queue. The queue is scheduled in accordance with the rules described in the **putq** utility. This utility is typically used to replace a message on the queue from which it was just removed.

Note: A service procedure must never put a high–priority message back on its own queue, as this would result in an infinite loop.

Parameters

q	Specifies the queue on which to place the message.
bp	Specifies the message to place on the queue.

Return Values

The **putbq** utility returns a value of 1 on success. Otherwise, it returns a value of 0.

Implementation Specifics

This utility is part of STREAMS Kernel Extensions.

Related Information

The **putq** utility.

List of Streams Programming References and Understanding STREAMS Messages in *AIX Communications Programming Concepts*.

putctl1 Utility

Purpose

Passes a control message with a one–byte parameter.

Syntax

```
int
putctl1(q, type, param)
queue_t *q;
```

Description

The **putctl1** utility creates a control message of the type specified by the *type* parameter with a one–byte parameter specified by the *param* parameter, and calls the put procedure of the queue pointed to by the *q* parameter, with a pointer to the created message as an argument.

The **putctl1** utility allocates new blocks by calling the **allocb** utility.

Parameters

q	Specifies the queue.
type	Specifies the type of control message.
param	Specifies the one–byte parameter.

Return Values

On successful completion, the **putctl1** utility returns a value of 1. It returns a value of 0 if it cannot allocate a message block, or if the value of the *type* parameter is **M_DATA**, **M_PROTO**, or **M_PCPROTO**. The **M_DELAY** type is allowed.

Implementation Specifics

This utility is part of STREAMS Kernel Extensions.

Related Information

The **allocb** utility, **putctl** utility.

List of Streams Programming References and Understanding STREAMS Messages in *AIX Communications Programming Concepts*.

putctl Utility

Purpose

Passes a control message.

Syntax

```
int
putctl(q, type)
queue_t *q;
```

Description

The **putctl** utility creates a control message of the type specified by the *type* parameter, and calls the put procedure of the queue pointed to by the *q* parameter. The argument of the put procedure is a pointer to the created message. The **putctl** utility allocates new blocks by calling the **allocb** utility.

Parameters

q	Specifies the queue that contains the desired put procedure.
type	Specifies the type of control message to create.

Return Values

On successful completion, the **putctl** utility returns a value of 1. It returns a value of 0 if it cannot allocate a message block, or if the value of the *type* parameter is **M_DATA**, **M_PROTO**, **M_PCPROTO**, or **M_DELAY**.

Implementation Specifics

This utility is part of STREAMS Kernel Extensions.

Related Information

The **allocb** utility, **putctl1** utility.

List of Streams Programming References and Understanding STREAMS Messages in *AIX Communications Programming Concepts*.

putmsg System Call

Purpose

Sends a message on a stream.

Syntax

```
#include <stropts.h>

int putmsg (fd, ctlptr,
dataptr, flags)
int fd;
struct strbuf *ctlptr;
struct strbuf *dataptr;
int flags;
```

Description

The **putmsg** system call creates a message from user–specified buffers and sends the message to a STREAMS file. The message may contain either a data part, a control part or both. The data and control parts to be sent are distinguished by placement in separate buffers. The semantics of each part is defined by the STREAMS module that receives the message.

Note: To use the **putmsg** system call you must import the /**lib**/**pse.exp** file during the compile. For example, compile the system call using the following command:

```
cc –bI:/lib/pse.exp
```

statement. Failure to import this file means that the **putmsg** system call will be unresolved during the link edit phase.

Parameters

fd	Specifies a file descriptor referencing an open stream.
ctlptr	Holds the control part of the message.
dataptr	Holds the data part of the message.
flags	Indicates the type of message to be sent. Acceptable values are:

0	Sends a nonpriority message.
RS_HIPRI	Sends a priority message.

The *ctlptr* and *dataptr* parameters each point to a **strbuf** structure that contains the following members:

```
int maxlen;     /* not used */
int len;        /* length of data */
char *buf;      /* ptr to buffer */
```

The `len` field in the **strbuf** structure indicates the number of bytes to be sent, and the `buf` field points to the buffer where the control information or data resides. The `maxlen` field is not used in the **putmsg** system call.

To send the data part of a message, the *dataptr* parameter must be nonnull and the `len` field of the *dataptr* parameter must have a value of 0 or greater. To send the control part of a message, the corresponding values must be set for the *ctlptr* parameter. No data (control) part will be sent if either the *dataptr* (*ctlptr*) parameter is null or the `len` field of the *dataptr* (*ctlptr*) parameter is set to –1.

If a control part is specified, and the *flags* parameter is set to **RS_HIPRI**, a priority message is sent. If the *flags* parameter is set to 0, a nonpriority message is sent. If no control part is specified and the *flags* parameter is set to **RS_HIPRI**, the **putmsg** system call fails and sets the **errno** global variable to **EINVAL**. If neither a control part nor a data part is specified and the *flags* parameter is set to 0, no message is sent and 0 is returned.

For nonpriority messages, the **putmsg** system call blocks if the stream write queue is full due to internal flow–control conditions. For priority messages, the **putmsg** system call does not block on this condition. For nonpriority messages, the **putmsg** system call does not block when the write queue is full and the **O_NDELAY** flag is set. Instead, the system call fails and sets the **errno** global variable to **EAGAIN**.

The **putmsg** system call also blocks, unless prevented by lack of internal resources, while waiting for the availability of message blocks in the stream, regardless of priority or whether the **O_NDELAY** flag has been specified. No partial message is sent.

Return Values

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

Error Codes

The **putmsg** system call fails if one of the following is true:

EAGAIN	A nonpriority message was specified, the **O_NDELAY** flag is set, and the stream write queue is full due to internal flow–control conditions.
EAGAIN	Buffers could not be allocated for the message that was to be created.
EBADF	The value of the *fd* parameter is not a valid file descriptor open for writing.
EFAULT	The *ctlptr* or *dataptr* parameter points outside the allocated address space.
EINTR	A signal was caught during the **putmsg** system call.
EINVAL	An undefined value was specified in the *flags* parameter, or the *flags* parameter is set to **RS_HIPRI** and no control part was supplied.
EINVAL	The stream referenced by the *fd* parameter is linked below a multiplexer.
ENOSTR	A stream is not associated with the *fd* parameter.
ENXIO	A hangup condition was generated downstream for the specified stream.
ERANGE	The size of the data part of the message does not fall within the range specified by the maximum and minimum packet sizes of the topmost STREAMS module. OR The control part of the message is larger than the maximum configured size of the control part of a message. OR The data part of a message is larger than the maximum configured size of the data part of a message.

The **putmsg** system call also fails if a STREAMS error message was processed by the stream head before the call. The error returned is the value contained in the STREAMS error message.

Implementation Specifics

This system call is part of STREAMS Kernel Extensions.

Files

/lib/pse.exp	Contains the STREAMS export symbols.

Related Information

The **getmsg** system call, **getpmsg** system call, **putpmsg** system call.

The **read** subroutine, **poll** subroutine, **write** subroutine.

List of Streams Programming References and STREAMS Overview in *AIX Communications Programming Concepts*.

putnext Utility

Purpose

Passes a message to the next queue.

Syntax

```
#define putnext(q, mp)
((*(q)->q_next->q_qinfo->qi_putp)((q)->q_next, (mp)))
```

Description

The **putnext** utility calls the put procedure of the next queue in a stream and passes to the procedure a message pointer as an argument. The **putnext** utility is the typical means of passing messages to the next queue in a stream.

Parameters

q	Specifies the calling queue.
mp	Specifies the message that is to be passed.

Implementation Specifics

This utility is part of STREAMS Kernel Extensions.

Related Information

List of Streams Programming References, Understanding STREAMS Messages in *AIX Communications Programming Concepts*.

putpmsg System Call

Purpose

Sends a priority message on a stream.

Syntax

```
#include <stropts.h>

int putpmsg (fd, ctlptr,
dataptr, band, flags)
int fd;
struct strbuf *ctlptr;
struct strbuf *dataptr;
int band;
int flags;
```

Description

The **putpmsg** system call is identical to the **putmsg** system call except that it sends a priority message. All information except for flag settings are found in the description for the **putmsg** system call. The differences in the flag settings are noted in the error codes section.

Parameters

fd	Specifies a file descriptor referencing an open stream.
ctlptr	Holds the control part of the message.
dataptr	Holds the data part of the message.
band	Indicates the priority band.
flags	Indicates the priority type of message to be sent. Acceptable values are:

MSG_BAND	Sends a non–priority message.
MSG_HIPRI	Sends a priority message.

Error Codes

The **putpmsg** system call is unsuccessful under the following conditions:

- The *flags* parameter is set to a value of 0.
- The *flags* parameter is set to **MSG_HIPRI** and the *band* parameter is set to a nonzero value.
- The *flags* parameter is set to **MSG_HIPRI** and no control part is specified.

Implementation Specifics

This system call is part of STREAMS Kernel Extensions.

Related Information

The **poll** subroutine, **read** subroutine, **write** subroutine.

The **getmsg** system call, **getpmsg** system call, **putmsg** system call.

List of Streams Programming References and STREAMS Overview in *AIX Communications Programming Concepts*.

putq Utility

Purpose

Puts a message on a queue.

Syntax

```
int
putq(q, bp)
register queue_t *q;
register mblk_t *bp;
```

Description

The **putq** utility puts the message pointed to by the *bp* parameter on the message queue pointed to by the *q* parameter, and then enables that queue. The **putq** utility queues messages based on message–queuing priority.

The priority classes are:

`type >= QPCTL`	High–priority
`type < QPCTL && band > 0`	Priority band
`type < QPCTL && band == 0`	Normal

When a high–priority message is queued, the **putq** utility always enables the queue. For a priority–band message, the **putq** utility is allowed to enable the queue (the **QNOENAB** flag is not set). Otherwise, the **QWANTR** flag is set, indicating that the service procedure is ready to read the queue. When an ordinary message is queued, the **putq** utility enables the queue if the following condition is set and if enabling is not inhibited by the **noenable** utility: the module has just been pushed, or else no message was queued on the last **getq** call and no message has been queued since.

The **putq** utility looks only at the priority band in the first message block of a message. If a high–priority message is passed to the **putq** utility with a nonzero `b_band` field value, the `b_band` field is reset to 0 before the message is placed on the queue. If the message passed to the **putq** utility has a `b_band` field value greater than the number of **qband** structures associated with the queue, the **putq** utility tries to allocate a new **qband** structure for each band up to and including the band of the message.

The **putq** utility should be used in the put procedure for the same queue in which the message is queued. A module should not call the **putq** utility directly in order to pass messages to a neighboring module. Instead, the **putq** utility itself can be used as the value of the `qi_putp` field in the put procedure for either or both of the module **qinit** structures. Doing so effectively bypasses any put–procedure processing and uses only the module service procedures.

> **Note:** The service procedure must never put a priority message back on its own queue, as this would result in an infinite loop.

Parameters

q	Specifies the queue on which to place the message.
bp	Specifies the message to put on the queue.

Return Values

On successful completion, the **putq** utility returns a value of 1. Otherwise, it returns a value of 0.

Implementation Specifics

This utility is part of STREAMS Kernel Extensions.

Related Information

The **getq** utility.

List of Streams Programming References and Understanding STREAMS Messages in *AIX Communications Programming Concepts*.

qenable Utility

Purpose

Enables a queue.

Syntax

```
void qenable (q)
register queue_t *q;
```

Description

The **qenable** utility places the queue pointed to by the *q* parameter on the linked list of queues ready to be called by the STREAMS scheduler.

Parameters

q	Specifies the queue to be enabled.

Implementation Specifics

This utility is part of STREAMS Kernel Extensions.

Related Information

List of Streams Programming References, Understanding STREAMS Messages in *AIX Communications Programming Concepts*.

qreply Utility

Purpose

Sends a message on a stream in the reverse direction.

Syntax

```
void qreply (q,
bp)
register queue_t *q;
register mblk_t *bp;
```

Description

The **qreply** utility sends the message pointed to by the *bp* parameter either up or down the stream–in the reverse direction from the queue pointed to by the *q* parameter. The utility does this by locating the partner of the queue specified by the *q* parameter (see the **OTHERQ** utility), and then calling the put procedure of that queue's neighbor (as in the **putnext** utility). The **qreply** utility is typically used to send back a response (**M_IOCACK** or **M_IOCNAK** message) to an **M_IOCTL** message.

Parameters

q	Specifies which queue to send the message up or down.
bp	Specifies the message to send.

Implementation Specifics

This utility is part of STREAMS Kernel Extensions.

Related Information

The **OTHERQ** utility, **putnext** utility.

List of Streams Programming References and Understanding STREAMS Messages in *AIX Communications Programming Concepts*.

qsize Utility

Purpose

Finds the number of messages on a queue.

Syntax

```
int
qsize(qp)
register queue_t *qp;
```

Description

The **qsize** utility returns the number of messages present in the queue specified by the *qp* parameter. If there are no messages on the queue, the **qsize** parameter returns a value of 0.

Parameters

qp	Specifies the queue on which to count the messages.

Implementation Specifics

This utility is part of STREAMS Kernel Extensions.

Related Information

List of Streams Programming References, Understanding STREAMS Messages in *AIX Communications Programming Concepts*.

RD Utility

Purpose

Gets the pointer to the read queue.

Syntax

```
#define RD(q) ((q)->q_next)
```

Description

The **RD** utility accepts a write–queue pointer, specified by the *q* parameter, as an argument and returns a pointer to the read queue for the same module.

Parameters

q	Specifies the write queue.

Implementation Specifics

This utility is part of STREAMS Kernel Extensions.

Related Information

The **OTHERQ** utility, **WR** utility.

List of Streams Programming References and Understanding STREAMS Messages in *AIX Communications Programming Concepts*.

rmvb Utility

Purpose

Removes a message block from a message.

Syntax

```
mblk_t *
rmvb(mp, bp)
register mblk_t *mp;
register mblk_t *bp;
```

Description

The **rmvb** utility removes the message block pointed to by the *bp* parameter from the message pointed to by the *mp* parameter, and then restores the linkage of the message blocks remaining in the message. The **rmvb** utility does not free the removed message block, but returns a pointer to the head of the resulting message. If the message block specified by the *bp* parameter is not contained in the message specified by the *mp* parameter, the **rmvb** utility returns a –1. If there are no message blocks in the resulting message, the **rmvb** utility returns a null pointer.

Parameters

bp	Specifies the message block to be removed.
mp	Specifies the message from which to remove the message block.

Implementation Specifics

This utility is part of STREAMS Kernel Extensions.

Related Information

List of Streams Programming References, Understanding STREAMS Messages in *AIX Communications Programming Concepts*.

rmvq Utility

Purpose

Removes a message from a queue.

Syntax

```
void rmvq (q, mp)
register queue_t *q;
register mblk_t *mp;
```

Description

Attention: If the *mp* parameter does not point to a message that is present on the specified queue, a system panic could result.

The **rmvq** utility removes the message pointed to by the *mp* parameter from the message queue pointed to by the *q* parameter, and then restores the linkage of the messages remaining on the queue.

Parameters

q	Specifies the queue from which to remove the message.
mp	Specifies the message to be removed.

Implementation Specifics

This utility is part of STREAMS Kernel Extensions.

Related Information

List of Streams Programming References, Understanding STREAMS Messages in *AIX Communications Programming Concepts*.

sad Device Driver

Purpose

Provides an interface for administrative operations.

Syntax

```
#include <sys/types.h>
#include <sys/conf.h>
#include <sys/sad.h>
#include <sys/stropts.h>

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

Description

The STREAMS Administrative Driver (**sad**) provides an interface for applications to perform administrative operations on STREAMS modules and drivers. The interface is provided through **ioctl** operations. Privileged operation can access the **sad** device driver in the /**dev**/**sad**/**user** directory.

Parameters

fildes	Specifies an open file descriptor that refers to the **sad** device driver.
command	Determines the control function to be performed.
arg	Supplies additional information for the given control function.

Values for the command Parameter

The **autopush** command allows a user to configure a list of modules to be automatically pushed on a stream when a driver is first opened. The **autopush** command is controlled by the following commands.

SAD_SAP

Allows the person performing administrative duties to configure the information for the given device, which is used by the **autopush** command. The *arg* parameter points to a **strapush** structure containing the following elements:

```
uint sap_cmd;
long sap_major;
long sap_minor;
long sap_lastminor;
long sap_npush;
uint sap_list[MAXAPUSH] [FMNAMESZ + 1];
```

The elements are described as follows:

`sap_cmd` Indicates the type of configuration being done. Acceptable values are:

- **SAP_ONE** Configures one minor device of a driver.
- **SAP_RANGE** Configures a range of minor devices of a driver.
- **SAP_ALL** Configures all minor devices of a driver.
- **SAP_CLEAR** Undoes configuration information for a driver.

`sap_major` Specifies the major device number of the device to be configured.

`sap_minor` Specifies the minor device number of the device to be configured.

`sap_lastminor` Specifies the last minor device number in a range of devices to be configured. This field is used only with the **SAP_RANGE** value in the `sap_cmd` field.

`sap_npush` Indicates the number of modules to be automatically pushed when the device is opened. The value of this field must be less than or equal to **MAXAPUSH**, which is defined in the **sad.h** file. It must also be less than or equal to **NSTRPUSH**, which is defined in the kernel master file.

`sap_list` Specifies an array of module names to be pushed in the order in which they appear in the list.

When using the **SAP_CLEAR** value, the user sets only the `sap_major` and `sap_minor` fields. This undoes the configuration information for any of the other values. If a previous entry was configured with the **SAP_ALL** value, the `sap_minor` field is set to 0. If a previous entry was configured with the **SAP_RANGE** value, the `sap_minor` field is set to the lowest minor device number in the range configured.

On successful completion, the return value from the **ioctl** operation is 0. Otherwise, the return value is –1.

SAD_GAP Allows any user to query the **sad** device driver to get the **autopush** configuration information for a given device. The *arg* parameter points to a **strapush** structure as described under the **SAD_SAP** value.

The user sets the `sap_major` and `sap_minor` fields to the major and minor device numbers, respectively, of the device in question. On return, the **strapush** structure is filled with the entire information used to configure the device. Unused entries are filled with zeros.

On successful completion, the return value from the **ioctl** operation is 0. Otherwise, the return value is –1.

SAD_VML

Allows any user to validate a list of modules; that is, to see if they are installed on the system. The *arg* parameter is a pointer to a **str_list** structure containing the following elements:

```
int sl_nmods;
struct str_mlist *sl_modlist;
```

The **str_mlist** structure contains the following element:

```
char l_name[FMNAMESZ+1];
```

The fields are defined as follows:

`sl_nmods` Indicates the number of entries the user has allocated in the array.

`sl_modlist` Points to the array of module names.

Return Values

On successful completion, the return value from the **ioctl** operation is 0 if the list is valid or 1 if the list contains an invalid module name. Otherwise the return value is –1.

Error Codes

On failure, the **errno** global variable is set to one of the following values:

EFAULT The *arg* parameter points outside the allocated address space.

EINVAL The major device number is not valid, the number of modules is not valid.

OR

The list of module names is not valid.

ENOSTR The major device number does not represent a STREAMS driver.

EEXIST The major–minor device pair is already configured.

ERANGE The value of the *command* parameter is **SAP_RANGE** and the value in the `sap_lastminor` field is not greater than the value in the `sap_minor` field.

OR

The value of the *command* parameter is **SAP_CLEAR** and the value in the `sap_minor` field is not equal to the first minor in the range.

ENODEV The value in the *command* parameter is **SAP_CLEAR** and the device is not configured for the **autopush** command.

ENOSR An internal **autopush** data structure cannot be allocated.

Related Information

The **autopush** command.

The **close** subroutine, **fstat** subroutine, **open** subroutine, **stat** subroutine.

Understanding streamio (STREAMS ioctl) Operations, Understanding STREAMS Drivers and Modules, Understanding the log Device Driver in *AIX Communications Programming Concepts*.

splstr Utility

Purpose

Sets the processor level.

Syntax

```
int splstr()
```

Description

The **splstr** utility increases the system processor level in order to block interrupts at a level appropriate for STREAMS modules and drivers when they are executing critical portions of their code. The **splstr** utility returns the processor level at the time of its invocation. Module developers are expected to use the standard **splx(s)** utility, where **s** is the integer value returned by the **splstr** operation, to restore the processor level to its previous value after the critical portions of code are passed.

Implementation Specifics

This utility is part of STREAMS Kernel Extensions.

Related Information

The **splx** utility.

List of Streams Programming References and STREAMS Overview in *AIX Communications Programming Concepts*.

splx Utility

Purpose

Terminates a section of code.

Syntax

```
int splx(x)
int x;
```

Description

The **splx** utility terminates a section of protected critical code. This utility restores the interrupt level to the previous level specified by the *x* parameter.

Implementation Specifics

This utility is part of STREAMS Kernel Extensions.

Related Information

The **splstr** utility.

List of Streams Programming References and Understanding STREAMS Drivers and Modules in *AIX Communications Programming Concepts*.

srv Utility

Purpose

Services queued messages for STREAMS modules or drivers.

Syntax

```
#include <sys/types.h>
#include <sys/stream.h>
#include <sys/stropts.h>
```

int <prefix>rsrv(queue_t *q); /* read side */

int <prefix>wsrv(queue_t *q); /* write side */

Parameters

q	Pointer to the queue structure.

Description

The optional service (<prefix>**srv**) routine can be included in a STREAMS module or driver for one or more of the following reasons:

- To provide greater control over the flow of messages in a stream
- To make it possible to defer the processing of some messages to avoid depleting system resources
- To combine small messages into larger ones, or break large messages into smaller ones
- To recover from resource allocation failure. A module's or driver's **put** routine can test for the availability of a resource, and if it is not available, enqueue the message for later processing by the **srv** routine.

A message is first passed to a module's or driver's **put** routine, which may or may not do some processing. It must then either:

- Pass the message to the next stream component with **putnext**
- If a **srv** routine has been included, it may call **putq** to place the message on the queue

Once a message has been enqueued, the STREAMS scheduler controls the invocation of the service routine. Service routines are called in FIFO order by the scheduler. No guarantees can be made about how long it will take for a **srv** routine to be called except that it will happen before any user level process is run.

Every stream component (stream head, module or driver) has limit values it uses to implement flow control. Tunable high and low water marks should be checked to stop and restart the flow of message processing. Flow control limits apply only between two adjacent components with **srv** routines.

STREAMS messages can be defined to have up to 256 different priorities to support requirements for multiple bands of data flow. At a minimum, a stream must distinguish between normal (priority zero) messages and high priority messages (such as M_IOCACK). High priority messages are always placed at the head of the **srv** routine's queue, after any other enqueued high priority messages. Next are messages from all included priority bands, which are enqueued in decreasing order of priority. Each priority band has its own flow control limits. If a flow controlled band is stopped, all lower priority bands are also stopped.

Once the STREAMS scheduler calls a **srv** routine, it must process all messages on its queue. The following steps are general guidelines for processing messages. Keep in mind that many of the details of how a **srv** routine should be written depend on the implementation, the direction of flow (upstream or downstream), and whether it is for a module or a driver.

1. Use **getq** to get the next enqueued message.
2. If the message is high priority, process it (if appropriate) and pass it to the next stream component with **putnext**.
3. If it is not a high priority message (and therefore subject to flow control), attempt to send it to the next stream component with a **srv** routine. Use **canput** or **bcanput** to determine if this can be done.
4. If the message cannot be passed, put it back on the queue with **putbq**. If it can be passed, process it (if appropriate) and pass it with **putnext**.

Rules for service routines:

1. Service routines must not call any kernel services that sleep or are not interrupt safe.
2. Service routines are called by the STREAMS scheduler with most interrupts enabled.

   Note: Each stream module must specify a read and a write service (**srv**) routine. If a service routine is not needed (because the **put** routine processes all messages), a NULL pointer should be placed in module's qinit structure. Do not use **nulldev** instead of the NULL pointer. Use of **nulldev** for a **srv** routine may result in flow control errors.

   Prior to AIX 4.1, STREAMS service routines were permitted which were not coded to specification (that is, the service routine called sleep or called kernel services that slept, other possibilities). In AIX 4.1, this behavior will cause a system failure because the STREAMS scheduler is executed with some interrupts disabled. Modules or drivers can force the old style scheduling by setting the sc_flags field of the kstrconf_t structure to STR_Q_NOTTOSPEC. This structure is passed to the system when the module or driver calls the str_install STREAMS service. This flag will cause STREAMS to schedule the module's or driver's service routines with all interrupts enabled. There is a severe performance penalty for this type of STREAMS scheduling and future releases of AIX may not support STR_Q_NOTTOSPEC.

Return Values

Ignored.

Related Information

put, **bcanput**, **canput**, **getq**, **putbq**, **putnext**, **putq** utilities.

The **queue** structure in /**usr/include/sys/stream.h**.

The STREAMS Entry Points article in InfoExplorer.

str_install Utility

Purpose

Installs streams modules and drivers.

Syntax

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


int
str_install(cmd, conf)
int cmd;
strconf_t *conf;
```

Description

The **str_install** utility adds or removes Portable Streams Environment (PSE) drivers and modules from the internal tables of PSE. The extension is pinned when added and unpinned when removed (see the **pincode** kernel service). It uses a configuration structure to provide sufficient information to perform the specified command.

The configuration structure, **strconf_t**, is defined as follows:

```
typedef struct {
        char *sc_name;
        struct streamtab *sc_str;
        int sc_open_stylesc_flags;
        int sc_major;
        int sc_sqlevel;
        caddr_t sc_sqinfo;
} strconf_t;
```

The elements of the **strconf_t** structure are defined as follows:

`sc_name`	Specifies the name of the extension in the internal tables of PSE. For modules, this name is installed in the **fmodsw** table and is used for **I_PUSH** operations. For drivers, this name is used only for reporting with the **scls** and **strinfo** commands.
`sc_str`	Points to a **streamtab** structure.

`sc_open_stylesc_flags`

Specifies the style of the driver or module open routine. The acceptable values are:

STR_NEW_OPEN
Specifies the open syntax and semantics used in System V Release 4.

STR_OLD_OPEN
Specifies the open syntax and semantics used in System V Release 3.

If the module is multiprocessor–safe, the following flag should be added by using the bitwise OR operator:

STR_MPSAFE Specifies that the extension was designed to run on a multiprocessor system.

If the module uses callback functions that need to be protected against interrupts (non–interrupt–safe callback functions) for the **timeout** or **bufcall** utilities, the following flag should be added by using the bitwise OR operator:

STR_QSAFETY
Specifies that the extension uses non–interrupt–safe callback functions for the **timeout** or **bufcall** utilities.

This flag is automatically set by STREAMS if the module is not multiprocessor–safe.

STR_PERSTREAM
Specifies that the module accepts to run at perstream synchronization level.

STR_Q_NOTTOSPEC
Specifies that the extension is designed to run it's service routine under process context.

By default STREAMS service routine runs under interrupt context (INTOFFL3). If Streams drivers or modules want to execute their service routine under process context (INTBASE), they need to set this flag.

STR_64BIT Specifies that the extension is capable to support 64–bit data types.

STR_NEWCLONING
Specifies the driver open uses new–style cloning. Under this style, the driver open() is not checking for CLONEOPEN flag and returns new device number.

`sc_major` Specifies the major number of the device.

`sc_sqlevel`

Reserved for future use. Specifies the synchronization level to be used by PSE. There are seven levels of synchronization:

SQLVL_NOP No synchronization
: Specifies that each queue can be accessed by more than one thread at the same time. The protection of internal data and of **put** and **service** routines against the **timeout** or **bufcall** utilities is done by the module or driver itself. This synchronization level should be used essentially for multiprocessor–efficient modules.

SQLVL_QUEUE Queue Level
: Specifies that each queue can be accessed by only one thread at the same time. This is the finest synchronization level, and should only be used when the two sides of a queue pair do not share common data.

SQLVL_QUEUEPAIR Queue Pair Level
: Specifies that each queue pair can be accessed by only one thread at the same time.

SQLVL_MODULE Module Level
: Specifies that all instances of a module can be accessed by only one thread at the same time. This is the default value.

SQLVL_ELSEWHERE Arbitrary Level
: Specifies that a group of modules can be accessed by only one thread at the same time. Usually, the group of modules is a set of cooperating modules, such as a protocol family. The group is defined by using the same name in the `sc_sqinfo` field for each module in the group.

SQLVL_GLOBAL Global Level
: Specifies that all of PSE can be accessed by only one thread at the same time. This option should normally be used only for debugging.

SQLVL_DEFAULT Default Level
: Specifies the default level, set to **SQLVL_MODULE**.

`sc_sqinfo`

Specifies an optional group name. This field is only used when the **SQLVL_ELSEWHERE** arbitrary synchronization level is set; all modules having the same name belong to one group. The name size is limited to eight characters.

Parameters

cmd

Specifies which operation to perform. Acceptable values are:

STR_LOAD_DEV
Adds a device into PSE internal tables.

STR_UNLOAD_DEV
Removes a device from PSE internal tables.

STR_LOAD_MOD
Adds a module into PSE internal tables.

STR_UNLOAD_MOD
Removes a module from PSE internal tables.

conf

Points to a **strconf_t** structure, which contains all the necessary information to successfully load and unload a PSE kernel extension.

Return Values

On successful completion, the **str_install** utility returns a value of 0. Otherwise, it returns an error code.

Error Codes

On failure, the **str_install** utility returns one of the following error codes:

EBUSY	The PSE kernel extension is already in use and cannot be unloaded.
EEXIST	The PSE kernel extension already exists in the system.
EINVAL	A parameter contains an unacceptable value.
ENODEV	The PSE kernel extension could not be loaded.
ENOENT	The PSE kernel is not present and could not be unloaded.
ENOMEM	Not enough memory for the extension could be allocated and pinned.
ENXIO	PSE is currently locked for use.

Implementation Specifics

This utility is part of STREAMS Kernel Extensions.

Related Information

The **pincode** kernel service, **unpincode** kernel service.

The **streamio** operations.

Configuring Drivers and Modules in the Portable Streams Environment (PSE) and List of Streams Programming References in *AIX Communications Programming Concepts*.

streamio Operations

Purpose

Perform a variety of control functions on streams.

Syntax

```
#include <stropts.h>

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

Description

See individual **streamio** operations for a description of each one.

Parameters

fildes	Specifies an open file descriptor that refers to a stream.
command	Determines the control function to be performed.
arg	Represents additional information that is needed by this operation.

The type of the *arg* parameter depends upon the operation, but it is generally an integer or a pointer to a *command*–specific data structure.

The *command* and *arg* parameters are passed to the file designated by the *fildes* parameter and are interpreted by the stream head. Certain combinations of these arguments can be passed to a module or driver in the stream.

Values of the command Parameter

The following ioctl operations are applicable to all STREAMS files:

I_ATMARK
Checks if the current message on the stream–head read queue is marked.

I_CANPUT Checks if a given band is writable.

I_CKBAND Checks if a message of a particular band is on the stream–head queue.

I_FDINSERT
Creates a message from user specified buffers, adds information about another stream and sends the message downstream.

I_FIND Compares the names of all modules currently present in the stream to a specified name.

I_FLUSH
Flushes all input or output queues.

I_FLUSHBAND
Flushes all message of a particular band.

I_GETBAND
Gets the band of the first message on the stream–head read queue.

I_GETCLTIME
Returns the delay time.

I_GETSIG

Returns the events for which the calling process is currently registered to be sent a **SIGPOLL** signal.

I_GRDOPT

Returns the current read mode setting.

I_LINK Connects two specified streams.

I_LIST Lists all the module names on the stream.

I_LOOK

Retrieves the name of the module just below the stream head.

I_NREAD

Counts the number of data bytes in data blocks in the first message on the stream–head read queue, and places this value in a specified location.

I_PEEK Allows a user to retrieve the information in the first message on the stream–head read queue without taking the message off the queue.

I_PLINK Connects two specified streams.

I_POP

Removes the module just below the stream head.

I_PUNLINK

Disconnects the two specified streams.

I_PUSH

Pushes a module onto the top of the current stream.

I_RECVFD Retrieves the file descriptor associated with the message sent by an **I_SENDFD** operation over a stream pipe.

I_SENDFD

Requests a stream to send a message to the stream head at the other end of a stream pipe.

I_SETCLTIME

Sets the time that the stream head delays when a stream is closing.

I_SETSIG

Informs the stream head that the user wishes the kernel to issue the **SIGPOLL** signal when a particular event occurs on the stream.

I_SRDOPT Sets the read mode.

I_STR Constructs an internal STREAMS ioctl message.

I_UNLINK

Disconnects the two specified streams.

Return Values

Unless specified otherwise, the return value from the **ioctl** subroutine is 0 upon success and –1 if unsuccessful with the **errno** global variable set as indicated.

Implementation Specifics

This operation is part of STREAMS Kernel Extensions.

Related Information

List of Streams Programming References, Understanding streamio (STREAMS ioctl) Operations in *AIX Communications Programming Concepts*.

I_ATMARK streamio Operation

Purpose

Checks to see if a message is marked.

Description

The **I_ATMARK** operation shows the user if the current message on the stream–head read queue is marked by a downstream module. The *arg* parameter determines how the checking is done when there are multiple marked messages on the stream–head read queue. The possible values for the *arg* parameter are:

ANYMARK	Read to determine if the message is marked by a downstream module.
LASTMARK	Read to determine if the message is the last one marked on the queue by a downstream module.

The **I_ATMARK** operation returns a value of 1 if the mark condition is satisfied. Otherwise, it returns a value of 0.

Error Codes

If unsuccessful, the **errno** global variable is set to the following value:

EINVAL	The value of the *arg* parameter could not be used.

Implementation Specifics

This operation is part of STREAMS Kernel Extensions.

Related Information

The **streamio** operations.

Understanding STREAMS Messages in *AIX Communications Programming Concepts*.

I_CANPUT streamio Operation

Purpose

Checks if a given band is writable.

Description

The **I_CANPUT** operation checks a given priority band to see if it can be written on. The *arg* parameter contains the priority band to be checked.

Return Values

The return value is set to one of the following:

0	The band is flow controlled.
1	The band is writable.
–1	An error occurred.

Error Codes

If unsuccessful, the **errno** global variable is set to the following value:

EINVAL	The value in the *arg* parameter is invalid.

Implementation Specifics

This operation is part of STREAMS Kernel Extensions.

Related Information

The **streamio** operations.

Understanding STREAMS Messages in *AIX Communications Programming Concepts*.

I_CKBAND streamio Operation

Purpose

Checks if a message of a particular band is on the stream–head read queue.

Description

The **I_CKBAND** operation checks to see if a message of a given priority band exists on the stream–head read queue. The *arg* parameter is an integer containing the value of the priority band being searched for.

The **I_CKBAND** operation returns a value of 1 if a message of the given band exists. Otherwise, it returns a value of –1.

Error Codes

If unsuccessful, the **errno** global variable is set to the following value:

EINVAL	The value in the *arg* parameter is not valid.

Implementation Specifics

This operation is part of STREAMS Kernel Extensions.

Related Information

The **streamio** operations.

Understanding STREAMS Messages in *AIX Communications Programming Concepts*.

I_FDINSERT streamio Operation

Purpose

Creates a message from user–specified buffers, adds information about another stream and sends the message downstream.

Description

The **I_FDINSERT** operation creates a message from user–specified buffers, adds information about another stream, and sends the message downstream. The message contains a control part and an optional data part. The data and control parts transmitted are identified by their placement in separate buffers. The *arg* parameter points to a **strfdinsert** structure that contains the following elements:

```
struct strbuf  ctlbuf;
struct strbuf  databuf;
long     flags;
int      fildes;
int      offset;
```

The `len` field in the **strbuf** structure must be set to the size of a pointer plus the number of bytes of control information sent with the message. The `fildes` field in the **strfdinsert** structure specifies the file descriptor of the other stream. The `offset` field, which must be word–aligned, specifies the number of bytes beyond the beginning of the control buffer to store a pointer. This pointer will be the address of the read queue structure of the driver for the stream corresponding to the `fildes` field in the **strfdinsert** structure. The `len` field in the **strbuf** structure of the `databuf` field must be set to the number of bytes of data information sent with the message or to 0 if no data part is sent.

The `flags` field specifies the type of message created. There are two valid values for the `flags` field:

0	Creates a nonpriority message.
RS_HIPRI	Creates a priority message.

For nonpriority messages, the **I_FDINSERT** operation blocks if the stream write queue is full due to internal flow–control conditions. For priority messages, the **I_FDINSERT** operation does not block on this condition. For nonpriority messages, the **I_FDINSERT** operation does not block when the write queue is full and the **O_NDELAY** flag is set. Instead, the operation fails and sets the **errno** global variable to **EAGAIN**.

The **I_FDINSERT** operation also blocks unless prevented by lack of internal resources, while it is waiting for the availability of message blocks in the stream, regardless of priority or whether the **O_NDELAY** flag has been specified. No partial message is sent.

Error Codes

If unsuccessful, the **errno** global variable is set to one of the following values:

EAGAIN	A nonpriority message was specified, the **O_NDELAY** flag is set, and the stream write queue is full due to internal flow–control conditions.
ENOSR	Buffers could not be allocated for the message that was to be created due to insufficient STREAMS memory resources.
EFAULT	The *arg* parameter points to an area outside the allocated address space, or the buffer area specified in the `ctlbuf` or `databuf` field is outside this space.

EINVAL One of the following conditions has occurred:

The `fildes` field in the **strfdinsert** structure is not a valid, open stream file descriptor.

The size of a pointer plus the value of the `offset` field is greater than the `len` field for the buffer specified through the `ctlptr` field.

The *offset* parameter does not specify a properly aligned location in the data buffer.

An undefined value is stored in the *flags* parameter.

ENXIO Hangup received on the *fildes* parameter of the **ioctl** call or the `fildes` field in the **strfdinsert** structure.

ERANGE The `len` field for the buffer specified through the `databuf` field does not fall within the range specified by the maximum and minimum packet sizes of the topmost stream module; or the `len` field for the buffer specified through the `databuf` field is larger than the maximum configured size of the data part of a message; or the `len` field for the buffer specified through the `ctlbuf` field is larger than the maximum configured size of the control part of a message.

The **I_FDINSERT** operation is also unsuccessful if an error message is received by the stream head corresponding to the `fildes` field in the **strfdinsert** structure. In this case, the **errno** global variable is set to the value in the message.

Implementation Specifics

This operation is part of STREAMS Kernel Extensions.

Related Information

The **streamio** operations.

Understanding STREAMS Messages in *AIX Communications Programming Concepts*.

I_FIND streamio Operation

Purpose

Compares the names of all modules currently present in the stream to a specified name.

Description

The **I_FIND** operation compares the names of all modules currently present in the stream to the name pointed to by the *arg* parameter, and returns a value of 1 if the named module is present in the stream. It returns a value of 0 if the named module is not present.

Error Codes

If unsuccessful, the **errno** global variable is set to one of the following values:

EFAULT	The *arg* parameter points outside the allocated address space.
EINVAL	The *arg* parameter does not contain a valid module name.

Implementation Specifics

This operation is part of STREAMS Kernel Extensions.

Related Information

The **streamio** operations.

Understanding STREAMS Drivers and Modules in *AIX Communications Programming Concepts*.

I_FLUSH streamio Operation

Purpose

Flushes all input or output queues.

Description

The **I_FLUSH** operation flushes all input or output queues, depending on the value of the *arg* parameter. Legal values for the *arg* parameter are:

FLUSHR	Flush read queues.
FLUSHW	Flush write queues.
FLUSHRW	Flush read and write queues.

Error Codes

If unsuccessful, the **errno** global variable is set to one of the following values:

ENOSR	Unable to allocate buffers for flush message due to insufficient STREAMS memory resources.
EINVAL	Invalid value for the *arg* parameter.
ENXIO	Hangup received on the *fildes* parameter.

Implementation Specifics

This operation is part of STREAMS Kernel Extensions.

Related Information

The **streamio** operations.

The **I_FLUSHBAND** streamio operation.

The **flushband** utility, **flushq** utility.

I_FLUSHBAND streamio Operation

Purpose

Flushes all messages from a particular band.

Description

The **I_FLUSHBAND** operation flushes all messages of a given priority band from all input or output queues. The *arg* parameter points to a **bandinfo** structure that contains the following elements:

```
unsigned char   bi_pri;
int             bi_flag;
```

The elements are defined as follows:

`bi_pri` Specifies the band to be flushed.

`bi_flag`
Specifies the queues to be pushed. Legal values for the `bi_flag` field are:

FLUSHR	Flush read queues.
FLUSHW	Flush write queues.
FLUSHRW	Flush read and write queues.

Error Codes

If unsuccessful, the **errno** global variable is set to one of the following values:

ENOSR	Unable to allocate buffers for flush message due to insufficient STREAMS memory resources.
EINVAL	Invalid value for the *arg* parameter.
ENXIO	Hangup received on the *fildes* parameter.

Implementation Specifics

This operation is part of STREAMS Kernel Extensions.

Related Information

The **streamio** operations.

The **I_FLUSH** streamio operation.

The **flushband** utility, **flushq** utility.

I_GETBAND streamio Operation

Purpose

Gets the band of the first message on the stream–head read queue.

Description

The **I_GETBAND** operation returns the priority band of the first message on the stream–head read queue in the integer referenced by the *arg* parameter.

Error Codes

If unsuccessful, the **errno** global variable is set to the following value:

ENODATA	No message is on the stream–head read queue.

Implementation Specifics

This operation is part of STREAMS Kernel Extensions.

Related Information

The **streamio** operations.

Understanding STREAMS Messages in *AIX Communications Programming Concepts*.

I_GETCLTIME streamio Operation

Purpose

Returns the delay time.

Description

The **I_GETCLTIME** operation returns the delay time, in milliseconds, that is pointed to by the *arg* parameter.

Implementation Specifics

This operation is part of STREAMS Kernel Extensions.

Related Information

The **streamio** operations.

The **I_SETCLTIME** streamio operation.

I_GETSIG streamio Operation

Purpose

Returns the events for which the calling process is currently registered to be sent a **SIGPOLL** signal.

Description

The **I_GETSIG** operation returns the events for which the calling process is currently registered to be sent a **SIGPOLL** signal. The events are returned as a bitmask pointed to by the *arg* parameter, where the events are those specified in the description of the **I_SETSIG** operation.

Error Codes

If unsuccessful, the **errno** global variable is set to one of the following values:

EINVAL	Process not registered to receive the **SIGPOLL** signal.
EFAULT	The *arg* parameter points outside the allocated address space.

Implementation Specifics

This operation is part of STREAMS Kernel Extensions.

Related Information

The **streamio** operations.

The **I_SETSIG** streamio operation.

I_GRDOPT streamio Operation

Purpose

Returns the current read mode setting.

Description

The **I_GRDOPT** operation returns the current read mode setting in an *int* parameter pointed to by the *arg* parameter. Read modes are described in the **read** subroutine description.

Error Codes

If unsuccessful, the **errno** global variable is set to the following value:

EFAULT	The *arg* parameter points outside the allocated address space.

Implementation Specifics

This operation is part of STREAMS Kernel Extensions.

Related Information

The **streamio** operations.

The **I_SRDOPT** streamio operation.

I_LINK streamio Operation

Purpose

Connects two specified streams.

Description

The **I_LINK** operation is used for connecting multiplexed STREAMS configurations.

The **I_LINK** operation connects two streams, where the *fildes* parameter is the file descriptor of the stream connected to the multiplexing driver, and the *arg* parameter is the file descriptor of the stream connected to another driver. The stream designated by the *arg* parameter gets connected below the multiplexing driver. The **I_LINK** operation requires the multiplexing driver to send an acknowledgment message to the stream head regarding the linking operation. This call returns a multiplexer ID number (an identifier used to disconnect the multiplexer; see the **I_UNLINK** operation) on success, and a value of –1 on failure.

Error Codes

If unsuccessful, the **errno** global variable is set to one of the following values:

ENXIO	Hangup received on the *fildes* field.
ETIME	Time out before acknowledgment message was received at stream head.
EAGAIN	Temporarily unable to allocate storage to perform the **I_LINK** operation.
ENOSR	Unable to allocate storage to perform the **I_LINK** operation due to insufficient STREAMS memory resources.
EBADF	The *arg* parameter is not a valid, open file descriptor.
EINVAL	The specified link operation would cause a cycle in the resulting configuration; that is, if a given stream head is linked into a multiplexing configuration in more than one place.

An **I_LINK** operation can also fail while waiting for the multiplexing driver to acknowledge the link request, if a message indicating an error or a hangup is received at the stream head of the *fildes* parameter. In addition, an error code can be returned in the positive or negative acknowledgment message. For these cases, the **I_LINK** operation fails with the **errno** global variable set to the value in the message.

Implementation Specifics

This operation is part of STREAMS Kernel Extensions.

Related Information

The **streamio** operations.

The **I_UNLINK** streamio operation, **I_PLINK** streamio operation.

I_LIST streamio Operation

Purpose

Lists all the module names on a stream.

Description

The **I_LIST** operation lists all of the modules present on a stream, including the topmost driver name. If the value of the *arg* parameter is null, the **I_LIST** operation returns the number of modules on the stream pointed to by the *fildes* parameter. If the value of the *arg* parameter is nonnull, it points to an **str_list** structure that contains the following elements:

```
int sl_nmods;
struct str_mlist      *sl_modlist;
```

The **str_mlist** structure contains the following element:

```
char l_name[FMNAMESZ+1];
```

The fields are defined as follows:

`sl_nmods`	Specifies the number of entries the user has allocated in the array.
`sl_modlist`	Contains the list of module names (on return).

Error Codes

If unsuccessful, the **errno** global variable is set to one of the following values:

EAGAIN	Unable to allocate buffers.
EINVAL	The `sl_nmods` member is less than 1.

Implementation Specifics

This operation is part of STREAMS Kernel Extensions.

Related Information

The **streamio** operations.

Understanding STREAMS Drivers and Modules in *AIX Communications Programming Concepts*.

I_LOOK streamio Operation

Purpose

Retrieves the name of the module just below the stream head.

Syntax

```
#include <sys/conf.h>
#include <stropts.h>

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

Description

The **I_LOOK** operation retrieves the name of the module just below the stream head of the stream pointed to by the *fildes* parameter and places it in a null terminated character string pointed at by the *arg* parameter. The buffer pointed to by the *arg* parameter should be at least FMNAMESMZ + 1 bytes long.

Error Codes

If unsuccessful, the **errno** global variable is set to one of the following values:

EFAULT	The *arg* parameter points outside the allocated address space.
EINVAL	No module is present in stream.

Implementation Specifics

This operation is part of STREAMS Kernel Extensions.

Related Information

The **streamio** operations.

The **I_FIND** streamio operation, **I_POP** streamio operation, **I_PUSH** streamio operation.

Understanding STREAMS Drivers and Modules in *AIX Communications Programming Concepts*.

I_NREAD streamio Operation

Purpose

Counts the number of data bytes in data blocks in the first message on the stream–head read queue, and places this value in a specified location.

Description

The **I_NREAD** operation counts the number of data bytes in data blocks in the first message on the stream–head read queue, and places this value in the location pointed to by the *arg* parameter.

Return Values

The return value for the operation is the number of messages on the stream–head read queue. For example, if a value of 0 is returned in the *arg* parameter, but the **ioctl** operation return value is greater than 0, this indicates that a zero–length message is next on the queue.

Error Codes

If unsuccessful, the **errno** global variable is set to the following value:

EFAULT	The *arg* parameter points outside the allocated address space.

Implementation Specifics

This operation is part of STREAMS Kernel Extensions.

Related Information

The **streamio** operations.

I_PEEK streamio Operation

Purpose

Allows a user to retrieve the information in the first message on the stream–head read queue without taking the message off the queue.

Description

The **I_PEEK** operation allows a user to retrieve the information in the first message on the stream–head read queue without taking the message off the queue. The *arg* parameter points to a **strpeek** structure that contains the following elements:

```
struct strbuf ctlbuff;
struct strbuf databuf;
long flags;
```

The `maxlen` field in the **strbuf** structures of the `ctlbuf` and `databuf` fields must be set to the number of bytes of control information or data information, respectively, to retrieve. If the user sets the `flags` field to **RS_HIPRI**, the **I_PEEK** operation looks for a priority message only on the stream–head read queue.

The **I_PEEK** operation returns a value of 1 if a message was retrieved, and returns a value of 0 if no message was found on the stream–head read queue, or if the **RS_HIPRI** flag was set in the `flags` field and a priority message was not present on the stream–head read queue. It does not wait for a message to arrive.

On return, the fields contain the following data:

ctlbuf	Specifies information in the control buffer.
databuf	Specifies information in the data buffer.
flags	Contains the value of 0 or **RS_HIPRI**.

Error Codes

If unsuccessful, the **errno** global variable is set to one of the following values:

EFAULT	The *arg* parameter points, or the buffer area specified in the `ctlbuf` or `databuf` field is outside the allocated address space.
EBADMSG	Queued message is not valid for the **I_PEEK** operation.

Implementation Specifics

This operation is part of STREAMS Kernel Extensions.

Related Information

The **streamio** operations.

Understanding STREAMS Messages in *AIX Communications Programming Concepts*.

I_PLINK streamio Operation

Purpose

Connects two specified streams.

Description

The **I_PLINK** operation is used for connecting multiplexed STREAMS configurations with a permanent link.

The **I_PLINK** operation connects two streams, where the *fildes* parameter is the file descriptor of the stream connected to the multiplexing driver, and the *arg* parameter is the file descriptor of the stream connected to another driver. The stream designated by the *arg* parameter gets connected by a permanent link below the multiplexing driver. The **I_PLINK** operation requires the multiplexing driver to send an acknowledgment message to the stream head regarding the linking operation. This call creates a permanent link which can exist even if the file descriptor associated with the upper stream to the multiplexing driver is closed. This call returns a multiplexer ID number (an identifier used to disconnect the multiplexer; see the **I_PUNLINK** operation) on success, and a value of –1 on failure.

Error Codes

If unsuccessful, the **errno** global variable is set to one of the following values:

ENXIO	Hangup received on the *fildes* field.
ETIME	Time out occurred before acknowledgment message was received at stream head.
EAGAIN	Unable to allocate storage to perform the **I_PLINK** operation.
EBADF	The *arg* parameter is not a valid, open file descriptor.
EINVAL	The *fildes* parameter does not support multiplexing. OR The *fildes* parameter is the file descriptor of a pipe or FIFO. OR The *arg* parameter is not a stream or is already linked under a multiplexer. OR The specified link operation would cause a cycle in the resulting configuration; that is, if a given stream head is linked into a multiplexing configuration in more than one place.

An **I_PLINK** operation can also be unsuccessful while waiting for the multiplexing driver to acknowledge the link request, if a message indicating an error or a hangup is received at the stream head of the *fildes* parameter. In addition, an error code can be returned in the positive or negative acknowledgment message. For these cases, the **I_PLINK** operation is unsuccessful with the **errno** global variable set to the value in the message.

Implementation Specifics

This operation is part of STREAMS Kernel Extensions.

Related Information

The **streamio** operations.

I_POP streamio Operation

Purpose

Removes the module just below the stream head.

Description

The **I_POP** operation removes the module just below the stream head of the stream pointed to by the *fildes* parameter. The value of the *arg* parameter should be 0 in an **I_POP** request.

Error Codes

If unsuccessful, the **errno** global variable is set to one of the following values:

EINVAL	No module is present in the stream.
ENXIO	Hangup received on the *fildes* parameter.

Implementation Specifics

This operation is part of STREAMS Kernel Extensions.

Related Information

The **streamio** operations.

The **I_FIND** streamio operation, **I_LIST** streamio operation, **I_LOOK** streamio operation, **I_PUSH** streamio operation.

Building STREAMS in *AIX Communications Programming Concepts*.

I_PUNLINK streamio Operation

Purpose

Disconnects the two specified streams.

Description

The **I_PUNLINK** operation is used for disconnecting Multiplexed STREAMS configurations connected by a permanent link.

The **I_PUNLINK** operation disconnects the two streams specified by the *fildes* parameter and the *arg* parameter that are connected with a permanent link. The *fildes* parameter is the file descriptor of the stream connected to the multiplexing driver. The *arg* parameter is the multiplexer ID number that was returned by the **I_PLINK** operation. If the value of the *arg* parameter is **MUXID_ALL**, then all streams which are permanently linked to the stream specified by the *fildes* parameter are disconnected. As in the **I_PLINK** operation, this operation requires the multiplexing driver to acknowledge the unlink.

Error Codes

If unsuccessful, the **errno** global variable is set to one of the following values:

ENXIO	Hangup received on the *fildes* parameter.
ETIME	Time out occurred before acknowledgment message was received at stream head.
EINVAL	The *arg* parameter is an invalid multiplexer ID number. OR The *fildes* parameter is the file descriptor of a pipe or FIFO.

An **I_PUNLINK** operation can also be unsuccessful while waiting for the multiplexing driver to acknowledge the link request, if a message indicating an error or a hangup is received at the stream head of the *fildes* parameter. In addition, an error code can be returned in the positive or negative acknowledgment message. For these cases, the **I_PUNLINK** operation is unsuccessful and the **errno** global variable is set to the value in the message.

Implementation Specifics

This operation is part of STREAMS Kernel Extensions.

Related Information

The **streamio** operations.

The **I_PLINK** streamio operation, **I_UNLINK** streamio operation.

I_PUSH streamio Operation

Purpose

Pushes a module onto the top of the current stream.

Description

The **I_PUSH** operation pushes the module whose name is pointed to by the *arg* parameter onto the top of the current stream, just below the stream head. It then calls the open routine of the newly–pushed module.

Error Codes

If unsuccessful, the **errno** global variable is set to one of the following values:

EINVAL	Incorrect module name.
EFAULT	The *arg* parameter points outside the allocated address space.
ENXIO	Open routine of new module failed.
ENXIO	Hangup received on the *fildes* parameter.

Implementation Specifics

This operation is part of STREAMS Kernel Extensions.

Related Information

The **autopush** command.

The **streamio** operations.

The **I_FIND** streamio operation, **I_LIST** streamio operation, **I_LOOK** streamio operation, **I_POP** streamio operation.

Building STREAMS in *AIX Communications Programming Concepts*.

I_RECVFD streamio Operation

Purpose

Retrieves the file descriptor associated with the message sent by an **I_SENDFD** operation over a stream pipe.

Description

The **I_RECVFD** operation retrieves the file descriptor associated with the message sent by an **I_SENDFD** operation over a stream pipe. The *arg* parameter is a pointer to a data buffer large enough to hold an **strrecvfd** data structure containing the following elements:

```
int fd;
unsigned short uid;
unsigned short gid;
char fill[8];
```

The fields of the **strrecvfd** structure are defined as follows:

`fd`	Specifies an integer file descriptor.
`uid`	Specifies the user ID of the sending stream.
`gid`	Specifies the group ID of the sending stream.

If the **O_NDELAY** flag is not set, the **I_RECVFD** operation blocks until a message is present at the stream head. If the **O_NDELAY** flag is set, the **I_RECVFD** operation fails with the **errno** global variable set to **EAGAIN** if no message is present at the stream head.

If the message at the stream head is a message sent by an **I_SENDFD** operation, a new user file descriptor is allocated for the file pointer contained in the message. The new file descriptor is place in the `fd` field of the **strrecvfd** structure. The structure is copied into the user data buffer pointed to by the *arg* parameter.

Error Codes

If unsuccessful, the **errno** global variable is set to one of the following values:

EAGAIN	A message was not present at the stream head read queue, and the **O_NDELAY** flag is set.
EBADMSG	The message at the stream head read queue was not a message containing a passed file descriptor.
EFAULT	The *arg* parameter points outside the allocated address space.
EMFILE	The **NOFILES** file descriptor is currently open.
ENXIO	Hangup received on the *fildes* parameter.

Implementation Specifics

This operation is part of STREAMS Kernel Extensions.

Related Information

The **isastream** function.

The **streamio** operations.

The **I_SENDFD** streamio operation.

I_SENDFD streamio Operation

Purpose

Requests a stream to send a message to the stream head at the other end of a stream pipe.

Description

The **I_SENDFD** operation requests the stream associated with the *fildes* field to send a message, containing a file pointer, to the stream head at the other end of a stream pipe. The file pointer corresponds to the *arg* parameter, which must be an integer file descriptor.

The **I_SENDFD** operation converts the *arg* parameter into the corresponding system file pointer. It allocates a message block and inserts the file pointer in the block. The user ID and group ID associated with the sending process are also inserted. This message is placed directly on the read queue of the stream head at the other end of the stream pipe to which it is connected.

Error Codes

If unsuccessful, the **errno** global variable is set to one of the following values:

EAGAIN	The sending stream is unable to allocate a message block to contain the file pointer.
EAGAIN	The read queue of the receiving stream head is full and cannot accept the message sent by the **I_SENDFD** operation.
EBADF	The *arg* parameter is not a valid, open file descriptor.
EINVAL	The *fildes* parameter is not connected to a stream pipe.
ENXIO	Hangup received on the *fildes* parameter.

Implementation Specifics

This operation is part of STREAMS Kernel Extensions.

Related Information

The **streamio** operations.

The **I_RECVFD** streamio operation.

The **putmsg** system call.

I_SETCLTIME streamio Operation

Purpose

Sets the time that the stream head delays when a stream is closing.

Description

The **I_SETCLTIME** operation sets the time that the stream head delays when a stream is closing and there is data on the write queues. Before closing each module and driver, the stream head delays closing for the specified length of time to allow the data to be written. Any data left after the delay is flushed.

The *arg* parameter contains a pointer to the number of milliseconds to delay. This number is rounded up to the nearest legal value on the system. The default delay time is 15 seconds.

Error Codes

If unsuccessful, the **errno** global variable is set to the following value:

EINVAL	The value in the *arg* parameter is invalid.

Implementation Specifics

This operation is part of STREAMS Kernel Extensions.

Related Information

The **streamio** operations.

The **I_GETCLTIME** streamio operation.

I_SETSIG streamio Operation

Purpose

Informs the stream head that the user wishes the kernel to issue the **SIGPOLL** signal when a particular event occurs on the stream.

Description

The **I_SETSIG** operation informs the stream head that the user wishes the kernel to issue the **SIGPOLL** signal (see the **signal** and **sigset** subroutines) when a particular event has occurred on the stream associated with the *fildes* parameter. The **I_SETSIG** operation supports an asynchronous processing capability in STREAMS. The value of the *arg* parameter is a bit mask that specifies the events for which the user should be signaled. It is the bitwise–OR of any combination of the following constants:

S_INPUT	A nonpriority message has arrived on a stream–head read queue, and no other messages existed on that queue before this message was placed there. This is set even if the message is of zero length.
S_HIPRI	A priority message is present on the stream–head read queue. This is set even if the message is of zero length.
S_OUTPUT	The write queue just below the stream head is no longer full. This notifies the user that there is room on the queue for sending (or writing) data downstream.
S_MSG	A STREAMS signal message that contains the **SIGPOLL** signal has reached the front of the stream–head read queue.

A user process may choose to be signaled only by priority messages by setting the *arg* bit mask to the value **S_HIRPI**.

Processes that wish to receive **SIGPOLL** signals must explicitly register to receive them using **I_SETSIG**. If several processes register to receive this signal for the same event on the same stream, each process will be signaled when the event occurs.

If the value of the *arg* parameter is 0, the calling process is unregistered and does not receive further **SIGPOLL** signals.

Error Codes

If unsuccessful, the **errno** global variable is set to one of the following values:

EINVAL	The value for the *arg* parameter is invalid or 0 and process is not registered to receive the **SIGPOLL** signal.
EAGAIN	The allocation of a data structure to store the signal request is unsuccessful.

Implementation Specifics

This operation is part of STREAMS Kernel Extensions.

Related Information

The **poll** subroutine, **signal** subroutine.

The **streamio** operations.

The **I_GETSIG** streamio operation.

Understanding STREAMS Monitoring in *AIX Communications Programming Concepts*.

I_SRDOPT streamio Operation

Purpose

Sets the read mode.

Description

The **I_SRDOPT** operation sets the read mode using the value of the *arg* parameter. Legal values for the *arg* parameter are:

RNORM Byte–stream mode. This is the default mode.

RMSGD Message–discard mode.

RMSGN Message–nondiscard mode.

RFILL

Read mode. This mode prevents completion of any **read** request until one of three conditions occurs:

- The entire user buffer is filled.
- An end of file occurs.
- The stream head receives an **M_MI_READ_END** message.

Several control messages support the **RFILL** mode. They are used by modules to manipulate data being placed in user buffers at the stream head. These messages are multiplexed under a single **M_MI** message type. The message subtype, pointed to by the *b_rptr* parameter, is one of the following:

M_MI_READ_SEEK
Provides random access data retrieval. An application and a cooperating module can gather large data blocks from a slow, high–latency, or unreliable link, while minimizing the number of system calls required, and relieving the protocol modules of large buffering requirements.

The **M_MI_READ_SEEK** message subtype is followed by two long words, as in a standard **seek** call. The first word is an origin indicator as follows:

0 Start of buffer

1 Current position

2 End of buffer

The second word is a signed offset from the specified origin.

M_MI_READ_RESET
Discards any data previously delivered to partially satisfy an **RFILL** mode **read** request.

M_MI_READ_END
Completes the current **RFILL** mode **read** request with whatever data has already been delivered.

In addition, treatment of control messages by the stream head can be changed by setting the following flags in the *arg* parameter:

RPROTNORM Causes the **read** routine to be unsuccessful if a control message is at the front of the stream–head read queue.

RPROTDAT Delivers the control portion of a message as data.

RPROTDIS Discards the control portion of a message, delivering any data portion.

Error Codes

If unsuccessful, the **errno** global variable is set to the following value:

EINVAL The value of the *arg* parameter is not one of the above legal values.

Implementation Specifics

This operation is part of STREAMS Kernel Extensions.

Related Information

The **streamio** operations.

The **I_GRDOPT** streamio operation.

I_STR streamio Operation

Purpose

Constructs an internal STREAMS ioctl message.

Description

The **I_STR** operation constructs an internal STREAMS ioctl message from the data pointed to by the *arg* parameter and sends that message downstream.

This mechanism is provided to send user ioctl requests to downstream modules and drivers. It allows information to be sent with the ioctl and returns to the user any information sent upstream by the downstream recipient. The **I_STR** operation blocks until the system responds with either a positive or negative acknowledgment message or until the request times out after some period of time. If the request times out, it fails with the **errno** global variable set to **ETIME**.

At most, one **I_STR** operation can be active on a stream. Further **I_STR** operation calls block until the active **I_STR** operation completes at the stream head. The default timeout interval for this request is 15 seconds. The **O_NDELAY** flag has no effect on this call.

To send a request downstream, the *arg* parameter must point to a **strioctl** structure that contains the following elements:

```
int  ic_cmd;      /* downstream operation */
int  ic_timeout;   /* ACK/NAK timeout */
int  ic_len:      /* length of data arg */
char *ic_dp;      /* ptr to data arg */
```

The elements of the **strioctl** structure are described as follows:

`ic_cmd` The internal **ioctl** operation intended for a downstream module or driver.

`ic_timout` The number of seconds an **I_STR** request waits for acknowledgment before timing out:

- –1 Waits an infinite number of seconds.
- 0 Uses default value.
- > 0 Waits the specified number of seconds.

`ic_len` The number of bytes in the data argument. The `ic_len` field has two uses:

- On input, it contains the length of the data argument passed in.
- On return from the operation, it contains the number of bytes being returned to the user (the buffer pointed to by the `ic_dp` field should be large enough to contain the maximum amount of data that any module or the driver in the stream can return).

`ic_dp` A pointer to the data parameter.

The stream head converts the information pointed to by the **strioctl** structure to an internal **ioctl** operation message and sends it downstream.

Error Codes

If unsuccessful, the **errno** global variable is set to one of the following values:

EAGAIN	The value of **ic_len** is greater than the maximum size of a message block returned by the STREAMS **allocb** utility, or there is insufficient memory for a message block.
ENOSR	Unable to allocate buffers for the ioctl message due to insufficient STREAMS memory resources.
EFAULT	The area pointed to by the *arg* parameter or the buffer area specified by the `ic_dp` and `ic_len` fields (for data sent and data returned, respectively) is outside of the allocated address space.
EINVAL	The value of the `ic_len` field is less than 0 or greater than the maximum configured size of the data part of a message, or the value of the `ic_timout` field is less than –1.
ENXIO	Hangup received on the `fildes` field.
ETIME	A downstream **streamio** operation timed out before acknowledgment was received.

An **I_STR** operation can also be unsuccessful while waiting for an acknowledgment if a message indicating an error or a hangup is received at the stream head. In addition, an error code can be returned in the positive or negative acknowledgment messages, in the event that the **streamio** operation sent downstream fails. For these cases, the **I_STR** operation is unsuccessful and the **errno** global variable is set to the value in the message.

Implementation Specifics

This operation is part of STREAMS Kernel Extensions.

Related Information

The **timod** Module.

The **streamio** operations.

Understanding STREAMS Messages in *AIX Communications Programming Concepts*.

I_UNLINK streamio Operation

Purpose

Disconnects the two specified streams.

Description

The **I_UNLINK** operation is used for disconnecting multiplexed STREAMS configurations.

The **I_UNLINK** operation disconnects the two streams specified by the *fildes* parameter and the *arg* parameter. The *fildes* parameter is the file descriptor of the stream connected to the multiplexing driver. The *fildes* parameter must correspond to the stream on which the **ioctl I_LINK** operation was issued to link the stream below the multiplexing driver. The *arg* parameter is the multiplexer ID number that was returned by the **I_LINK** operation. If the value of the *arg* parameter is –1, then all streams that were linked to the *fildes* parameter are disconnected. As in the **I_LINK** operation, this operation requires the multiplexing driver to acknowledge the unlink.

Error Codes

If unsuccessful, the **errno** global variable is set to one of the following values:

ENXIO	Hangup received on the *fildes* parameter.
ETIME	Time out before acknowledgment message was received at stream head.
ENOSR	Unable to allocate storage to perform the **I_UNLINK** operation due to insufficient STREAMS memory resources.
EINVAL	The *arg* parameter is an invalid multiplexer ID number or the *fildes* parameter is not the stream on which the **I_LINK** operation that returned the *arg* parameter was performed.

An **I_UNLINK** operation can also fail while waiting for the multiplexing driver to acknowledge the link request, if a message indicating an error or a hangup is received at the stream head of the *fildes* parameter. In addition, an error code can be returned in the positive or negative acknowledgment message. For these cases, the **I_UNLINK** operation fails and the **errno** global variable is set to the value in the message.

Implementation Specifics

This operation is part of STREAMS Kernel Extensions.

Related Information

The **I_LINK** streamio operation, **I_PUNLINK** streamio operation.

List of Streams Programming References, Understanding streamio (STREAMS ioctl) Operations, Understanding STREAMS Drivers and Modules, Understanding STREAMS Messages in *AIX Communications Programming Concepts*.

strlog Utility

Purpose

Generates STREAMS error–logging and event–tracing messages.

Syntax

```
int
strlog(mid, sid, level, flags, fmt, arg1, . . . )
short mid, sid;
char level;
ushort flags;
char *fmt;
unsigned arg1;
```

Description

The **strlog** utility generates log messages within the kernel. Required definitions are contained in the **sys**/**strlog.h** file.

Parameters

mid Specifies the STREAMS module ID number for the module or driver submitting the **log** message.

sid Specifies an internal sub–ID number usually used to identify a particular minor device of a driver.

level Specifies a tracing level that allows for selective screening of low–priority messages from the tracer.

flags

Specifies the destination of the message. This can be any combination of:

SL_ERROR The message is for the error logger.

SL_TRACE The message is for the tracer.

SL_CONSOLE Log the message to the console.

SL_FATAL Advisory notification of a fatal error.

SL_WARN Advisory notification of a nonfatal error.

SL_NOTE Advisory message.

SL_NOTIFY Request that a copy of the message be mailed to the system administrator.

fmt Specifies a print style–format string, except that %s, %f, %e, %E, %g, and %G conversion specifications are not handled.

arg1 Specifies numeric or character arguments. Up to **NLOGARGS** (currently 4) numeric or character arguments can be provided. (The **NLOGARGS** variable specifies the maximum number of arguments allowed. It is defined in the **sys**/**strlog.h** file.)

Implementation Specifics

This utility is part of STREAMS Kernel Extensions.

Related Information

The **streamio** operations.

clone Device Driver in *AIX Communications Programming Concepts*.

List of Streams Programming References, Understanding the log Device Driver, Understanding STREAMS Error and Trace Logging in *AIX Communications Programming Concepts*.

strqget Utility

Purpose

Obtains information about a queue or band of the queue.

Syntax

```
int
strqget(q, what, pri, valp)
register queue_t *q;
qfields_t what;
register unsigned char pri;
long *valp;
```

Description

The **strqget** utility allows modules and drivers to get information about a queue or particular band of the queue. The information is returned in the *valp* parameter. The fields that can be obtained are defined as follows:

```
typedef enum qfileds {
        QHIWAT   = 0,
        QLOWAT   = 1,
        QMAXPSZ  = 2,
        QMINPSZ  = 3,
        QCOUNT   = 4,
        QFIRST   = 5,
        QLAST    = 6,
        QFLAG    = 7,
        QBAD     = 8
} qfields_t;
```

Parameters

q	Specifies the queue about which to get information.
what	Specifies the information to get from the queue.
pri	Specifies the priority band about which to get information.
valp	Contains the requested information on return.

Return Values

On success, the **strqget** utility returns a value of 0. Otherwise, it returns an error number.

Implementation Specifics

This utility is part of STREAMS Kernel Extensions.

Related Information

List of Streams Programming References, Understanding STREAMS Messages in *AIX Communications Programming Concepts*.

t_accept Subroutine for Transport Layer Interface

Purpose

Accepts a connect request.

Library

Transport Layer Interface Library (**libtli.a**)

Syntax

```
#include <tiuser.h>

int t_accept (fd, resfd, call)
int fd;
int resfd;
struct t_call *call;
```

Description

The **t_accept** subroutine is issued by a transport user to accept a connect request. A transport user can accept a connection on either the same local transport end point or on an end point different from the one on which the connect indication arrived.

Parameters

fd Identifies the local transport end point where the connect indication arrived.

resfd Specifies the local transport end point where the connection is to be established.

call Contains information required by the transport provider to complete the connection. The *call* parameter points to a **t_call** structure, which contains the following members:

```
struct netbuf addr;
struct netbuf opt;
struct netbuf udata;
int sequence;
```

The **netbuf** structure is described in the **tiuser.h** file. In the *call* parameter, the `addr` field is the address of the caller, the `opt` field indicates any protocol–specific parameters associated with the connection, the `udata` field points to any user data to be returned to the caller, and the `sequence` field is the value returned by the **t_listen** subroutine which uniquely associates the response with a previously received connect indication.

If the same end point is specified (that is, the *resfd* value equals the *fd* value), the connection can be accepted unless the following condition is true: the user has received other indications on that end point, but has not responded to them (with either the **t_accept** or **t_snddis** subroutine). For this condition, the **t_accept** subroutine fails and sets the **t_errno** variable to **TBADF**.

If a different transport end point is specified (that is, the *resfd* value does not equal the *fd* value), the end point must be bound to a protocol address and must be in the **T_IDLE** state (see the **t_getstate** subroutine) before the **t_accept** subroutine is issued.

For both types of end points, the **t_accept** subroutine fails and sets the **t_errno** variable to **TLOOK** if there are indications (for example, a connect or disconnect) waiting to be received on that end point.

The values of parameters specified by the `opt` field and the syntax of those values are protocol–specific. The `udata` field enables the called transport user to send user data to the caller, the amount of user data must not exceed the limits supported by the transport provider as returned by the **t_open** or **t_getinfo** subroutine. If the value in the `len` field of the `udata` field is 0, no data will be sent to the caller.

Return Values

On successful completion, the **t_connect** subroutine returns a value of 0. Otherwise, it returns a value of –1, and the **t_errno** variable is set to indicate the error.

Error Codes

If unsuccessful, the **t_errno** variable is set to one of the following:

TACCES The user does not have permission to accept a connection on the responding transport end point or use the specified options.

TBADDATA The amount of user data specified was not within the bounds allowed by the transport provider.

TBADDR	The specified protocol address was in an incorrect format or contained illegal information.
TBADF	The specified file descriptor does not refer to a transport end point; or the user is illegally accepting a connection on the same transport end point on which the connect indication arrived.
TBADOPT	The specified options were in an incorrect format or contained illegal information.
TBADSEQ	An incorrect sequence number was specified.
TLOOK	An asynchronous event has occurred on the transport end point referenced by the *fd* parameter and requires immediate attention.
TNOTSUPPORT	This function is not supported by the underlying transport provider.
TOUTSTATE	The function was issued in the wrong sequence on the transport end point referenced by the *fd* parameter, or the transport end point referred to by the *resfd* parameter is not in the **T_IDLE** state.
TSYSERR	A system error has occurred during execution of this function.

Implementation Specifics

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

Related Information

The **t_alloc** subroutine, **t_connect** subroutine, **t_getinfo** subroutine, **t_getstate** subroutine, **t_listen** subroutine, **t_open** subroutine, **t_rcvconnect** subroutine and **t_snddis** subroutine.

List of Streams Programming References and STREAMS Overview in *AIX Communications Programming Concepts*.

t_alloc Subroutine for Transport Layer Interface

Purpose

Allocates a library structure.

Library

Transport Layer Interface Library (**libtli.a**)

Syntax

```
#include <tiuser.h>

char *t_alloc (fd, struct_type, fields)
int fd;
int struct_type;
int fields;
```

Description

The **t_alloc** subroutine dynamically assigns memory for the various transport–function argument structures. This subroutine allocates memory for the specified structure, and also allocates memory for buffers referenced by the structure.

Use of the **t_alloc** subroutine to allocate structures will help ensure the compatibility of user programs with future releases of the transport interface.

Parameters

fd Specifies the transport end point through which the newly allocated structure will be passed.

struct_type

Specifies the structure to be allocated. The structure to allocate is specified by the *struct_type* parameter, and can be one of the following:

T_BIND	**struct t_bind**
T_CALL	**struct t_call**
T_OPTMGMT	**struct t_optmgmt**
T_DIS	**struct t_discon**
T_UNITDATA	**struct t_unitdata**
T_UDERROR	**struct t_uderr**
T_INFO	**struct t_info**

Each of these structures may subsequently be used as a parameter to one or more transport functions.

Each of the above structures, except **T_INFO**, contains at least one field of the **struct netbuf** type. The **netbuf** structure is described in the **tiuser.h** file. For each field of this type, the user may specify that the buffer for that field should be allocated as well. The *fields* parameter specifies this option, where the parameter is the bitwise–OR of any of the following:

T_ADDR	The `addr` field of the **t_bind**, **t_call**, **t_unitdata**, or **t_uderr** structure.
T_OPT	The `opt` field of the **t_optmgmt**, **t_call**, **t_unitdata**, or **t_uderr** structure.
T_UDATA	The `udata` field of the **t_call**, **t_discon**, or **t_unitdata** structure.
T_ALL	All relevant fields of the given structure.

fields Specifies whether the buffer should be allocated for each field type. For each field specified in the *fields* parameter, the **t_alloc** subroutine allocates memory for the buffer associated with the field, initializes the `len` field to zero, and initializes the `buf` pointer and the `maxlen` field accordingly. The length of the buffer allocated is based on the same size information returned to the user from the **t_open** and **t_getinfo** subroutines. Thus, the *fd* parameter must refer to the transport end point through which the newly allocated structure will be passed, so that the appropriate size information can be accessed. If the size value associated with any specified field is –1 or –2, the **t_alloc** subroutine will be unable to determine the size of the buffer to allocate; it then fails, setting the **t_errno** variable to **TSYSERR** and the **errno** global variable to **EINVAL**. For any field not specified in the *fields* parameter, the `buf` field is set to null and the `maxlen` field is set to 0.

Return Values

On successful completion, the **t_alloc** subroutine returns a pointer to the newly allocated structure. Otherwise, it returns a null pointer.

Error Codes

On failure, the **t_errno** variable is set to one of the following:

TBADF	The specified file descriptor does not refer to a transport end point.
TNOSTRUCTYPE	Unsupported structure type requested. This can include a request for a structure type which is inconsistent with the transport provider type specified, for example, connection–oriented or connectionless.
TSYSERR	A system error has occurred during execution of this function.

Implementation Specifics

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

Related Information

The **t_free** subroutine, **t_getinfo** subroutine, **t_open** subroutine.

List of Streams Programming References and STREAMS Overview in *AIX Communications Programming Concepts*.

t_bind Subroutine for Transport Layer Interface

Purpose

Binds an address to a transport end point.

Library

Transport Layer Interface Library (**libtli.a**)

Syntax

```
#include <tiuser.h>

int t_bind(fd, req, ret)
int fd;
struct t_bind *req;
struct t_bind *ret;
```

Description

The **t_bind** subroutine associates a protocol address with the transport end point specified by the *fd* parameter and activates that transport end point. In connection mode, the transport provider may begin accepting or requesting connections on the transport end point. In connectionless mode, the transport user may send or receive data units through the transport end point.

Parameters

fd	Specifies the transport end point.
req	Specifies the address to be bound to the given transport end point.
ret	Specifies the maximum size of the address buffer.

The *req* and *ret* parameters point to a **t_bind** structure containing the following members:

```
struct netbuf addr;
unsigned qlen;
```

The **netbuf** structure is described in the **tiuser.h** file. The `addr` field of the **t_bind** structure specifies a protocol address and the `qlen` field is used to indicate the maximum number of outstanding connect indications.

The *req* parameter is used to request that the address represented by the **netbuf** structure be bound to the given transport end point. In the *req* parameter, the **netbuf** structure fields have the following meanings:

`len`	Specifies the number of bytes in the address.
`buf`	Points to the address buffer.
`maxlen`	Has no meaning for the *req* parameter.

On return, the *ret* parameter contains the address that the transport provider actually bound to the transport end point; this may be different from the address specified by the user in the *req* parameter. In the *ret* parameter, the **netbuf** structure fields have the following meanings:

`maxlen`	Specifies the maximum size of the address buffer.
`buf`	Points to the buffer where the address is to be placed. (On return, this field points to the bound address.)
`len`	Specifies the number of bytes in the bound address.

If the value of the `maxlen` field is not large enough to hold the returned address, an error will result.

If the requested address is not available or if no address is specified in the *req* parameter (that is, the `len` field of the `addr` field in the *req* parameter is 0) the transport provider assigns an appropriate address to be bound and returns that address in the `addr` field of the *ret* parameter. The user can compare the addresses in the *req* parameter to those in the *ret* parameter to determine whether the transport provider has bound the transport end point to a different address than that requested. If the transport provider could not allocate an address, the **t_bind** subroutine fails and **t_errno** is set to **TNOADDR**.

The *req* parameter may be null if the user does not wish to specify an address to be bound. Here, the value of the `qlen` field is assumed to be 0, and the transport provider must assign an address to the transport end point. Similarly, the *ret* parameter may be null if the user does not care which address was bound by the provider and is not interested in the negotiated value of the `qlen` field. It is valid to set the *req* and *ret* parameters to null for the same call, in which case the provider chooses the address to bind to the transport end point and does not return that information to the user.

The `qlen` field has meaning only when initializing a connection–mode service. It specifies the number of outstanding connect indications the transport provider should support for the given transport end point. An outstanding connect indication is one that has been passed to the transport user by the transport provider. A value of the `qlen` field greater than 0 is only meaningful when issued by a passive transport user that expects other users to call it. The value of the `qlen` field is negotiated by the transport provider and can be changed if the transport provider cannot support the specified number of outstanding connect indications. On return, the `qlen` field in the *ret* parameter contains the negotiated value.

This subroutine allows more than one transport end point to be bound to the same protocol address as long as the transport provider also supports this capability. However, it is not allowable to bind more than one protocol address to the same transport end point. If a user binds more than one transport end point to the same protocol address, only one end point can be used to listen for connect indications associated with that protocol address. In other words, only one **t_bind** subroutine for a given protocol address may specify a value greater than 0 for the `qlen` field. In this way, the transport provider can identify which transport end point should be notified of an incoming connect indication. If a user attempts to bind a protocol address to a second transport end point having a `qlen` value greater than 0, the transport provider instead assigns another address to be bound to that end point. If a user accepts a connection on the transport end point that is being used as the listening end point, the bound protocol address is found to be busy for the duration of that connection. No other transport end points may be bound for listening while that initial listening end point is in the data–transfer phase. This prevents more than one transport end point bound to the same protocol address from accepting connect indications.

Return Values

On successful completion, the **t_connect** subroutine returns a value of 0. Otherwise, it returns a value of –1, and the **t_errno** variable is set to indicate the error.

Error Codes

If unsuccessful, the **t_errno** variable is set to one of the following:

TACCES	The user does not have permission to use the specified address.
TADDRBUSY	The requested address is in use.
TBADADDR	The specified protocol address was in an incorrect format or contained illegal information.
TBADF	The specified file descriptor does not refer to a transport end point.

TBUFOVFLW	The number of bytes allowed for an incoming argument is not sufficient to store the value of that argument. The provider's state changes to **T_IDLE** and the information to be returned in the *ret* parameter is discarded.
TNOADDR	The transport provider could not allocate an address.
TOUTSTATE	The function was issued in the wrong sequence.
TSYSERR	A system error has occurred during execution of this function.

Implementation Specifics

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

Related Information

The **t_open** subroutine, **t_optmgmt** subroutine, **t_unbind** subroutine.

List of Streams Programming References and STREAMS Overview in *AIX Communications Programming Concepts*.

t_close Subroutine for Transport Layer Interface

Purpose

Closes a transport end point.

Library

Transport Layer Interface Library (**libtli.a**)

Syntax

```
#include <tiuser.h>

int t_close(fd)
int fd;
```

Description

The **t_close** subroutine informs the transport provider that the user is finished with the transport end point specified by the *fd* parameter and frees any local library resources associated with the end point. In addition, the **t_close** subroutine closes the file associated with the transport end point.

The **t_close** subroutine should be called from the **T_UNBND** state (see the **t_getstate** subroutine). However, this subroutine does not check state information, so it may be called from any state to close a transport end point. If this occurs, the local library resources associated with the end point are freed automatically. In addition, the **close** subroutine is issued for that file descriptor. The **close** subroutine is abortive if no other process has that file open, and will break any transport connection that may be associated with that end point.

Parameter

fd	Specifies the transport end point to be closed.

Return Values

On successful completion, the **t_connect** subroutine returns a value of 0. Otherwise, it returns a value of –1, and the **t_errno** variable is set to indicate the error.

Error Code

If unsuccessful, the **t_errno** variable is set to the following:

TBADF	The specified file descriptor does not refer to a transport end point.

Implementation Specifics

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

Related Information

The **close** subroutine, **t_getstate** subroutine, **t_open** subroutine, **t_unbind** subroutine.

List of Streams Programming References and STREAMS Overview in *AIX Communications Programming Concepts*.

t_connect Subroutine for Transport Layer Interface

Purpose

Establishes a connection with another transport user.

Library

Transport Layer Interface Library (**libtli.a**)

Syntax

```
#include <tiuser.h>

int t_connect(fd, sndcall, rcvcall)
int fd;
struct t_call *sndcall;
struct t_call *rcvcall;
```

Description

The **t_connect** subroutine enables a transport user to request a connection to the specified destination transport user.

Parameters

fd	Identifies the local transport end point where communication will be established.
sndcall	Specifies information needed by the transport provider to establish a connection.
rcvcall	Specifies information associated with the newly established connection.

The *sndcall* and *rcvcall* parameters point to a **t_call** structure that contains the following members:

```
struct netbuf addr;
struct netbuf opt;
struct netbuf udata;
int sequence;
```

The **netbuf** structure is described in the **tiuser.h** file. In the *sndcall* parameter, the `addr` field specifies the protocol address of the destination transport user, the `opt` field presents any protocol–specific information that might be needed by the transport provider, the `udata` field points to optional user data that may be passed to the destination transport user during connection establishment, and the `sequence` field has no meaning for this function.

On return to the *rcvcall* parameter, the `addr` field returns the protocol address associated with the responding transport end point, the `opt` field presents any protocol–specific information associated with the connection, the `udata` field points to optional user data that may be returned by the destination transport user during connection establishment; and the `sequence` field has no meaning for this function.

The `opt` field implies no structure on the options that may be passed to the transport provider. The transport provider is free to specify the structure of any options passed to it. These options are specific to the underlying protocol of the transport provider. The user can choose not to negotiate protocol options by setting the `len` field of the `opt` field to 0. In this case, the provider may use default options.

The `udata` field enables the caller to pass user data to the destination transport user and receive user data from the destination user during connection establishment. However, the amount of user data must not exceed the limits supported by the transport provider as

returned by the **t_open** or **t_getinfo** subroutine. If the `len` field of the `udata` field in the *sndcall* parameter is 0, no data is sent to the destination transport user.

On return, the `addr`, `opt`, and `udata` fields of the **rcvcall** parameter are updated to reflect values associated with the connection. Thus, the `maxlen` field of each parameter must be set before issuing this function to indicate the maximum size of the buffer for each. However, the **rcvcall** parameter may be null, in which case no information is given to the user on return from the **t_connect** subroutine.

By default, the **t_connect** subroutine executes in synchronous mode, and waits for the destination user's response before returning control to the local user. A successful return (that is, a return value of 0) indicates that the requested connection has been established. However, if the **O_NDELAY** flag is set (with the **t_open** subroutine or the **fcntl** command), the **t_connect** subroutine executes in asynchronous mode. In this case, the call does not wait for the remote user's response, but returns control immediately to the local user and returns –1 with the **t_errno** variable set to **TNODATA** to indicate that the connection has not yet been established. In this way, the function simply initiates the connection establishment procedure by sending a connect request to the destination transport user.

Return Values

On successful completion, the **t_connect** subroutine returns a value of 0. Otherwise, it returns a value of –1, and the **t_errno** variable is set to indicate the error.

Error Codes

If unsuccessful, the **t_errno** variable is set to one of the following:

TACCES	The user does not have permission to use the specified address or options.
TBADADDR	The specified protocol address was in an incorrect format or contained illegal information.
TBADDATA	The amount of user data specified was not within the bounds allowed by the transport provider.
TBADF	The specified file descriptor does not refer to a transport end point.
TBADOPT	The specified protocol options were in an incorrect format or contained illegal information.
TBUFOVFLW	The number of bytes allocated for an incoming argument is not sufficient to store the value of that argument. If executed in synchronous mode, the provider's state, as seen by the user, changes to **T_DATAXFER**, and the connect indication information to be returned in the *rcvcall* parameter is discarded.
TLOOK	An asynchronous event has occurred on this transport end point and requires immediate attention.
TNODATA	The **O_NDELAY** or **O_NONBLOCK** flag was set, so the function successfully initiated the connection establishment procedure, but did not wait for a response from the remote user.
TNOTSUPPORT	This function is not supported by the underlying transport provider.
TOUTSTATE	The function was issued in the wrong sequence.
TSYSERR	A system error has occurred during execution of this function.

Implementation Specifics

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

Related Information

The **fcntl** command.

The **t_accept** subroutine, **t_getinfo** subroutine, **t_listen** subroutine, **t_open** subroutine, **t_optmgmt** subroutine, **t_rcvconnect** subroutine.

List of Streams Programming References and STREAMS Overview in *AIX Communications Programming Concepts*.

t_error Subroutine for Transport Layer Interface

Purpose

Produces an error message.

Library

Transport Layer Interface Library (**libtli.a**)

Syntax

```
#include <tiuser.h>

void t_error(errmsg)
char *errmsg;
extern int t_errno;
extern char *t_errno;
extern int t_nerr;
```

Description

The **t_error** subroutine produces a message on the standard error output that describes the last error encountered during a call to a transport function.

The **t_error** subroutine prints the user–supplied error message, followed by a colon and the standard transport–function error message for the current value contained in the **t_errno** variable.

Parameter

errmsg Specifies a user–supplied error message that gives context to the error.

External Variables

t_errno Specifies which standard transport–function error message to print. If the value of the **t_errno** variable is **TSYSERR**, the **t_error** subroutine also prints the standard error message for the current value contained in the **errno** global variable.

The **t_errno** variable is set when an error occurs and is not cleared on subsequent successful calls.

t_nerr Specifies the maximum index value for the **t_errlist** array. The **t_errlist** array is the array of message strings allowing user–message formatting. The **t_errno** variable can be used as an index into this array to retrieve the error message string (without a terminating new–line character).

Examples

A **t_connect** subroutine is unsuccessful on transport end point `fd2` because a bad address was given, and the following call follows the failure:

```
t_error("t_connect failed on fd2")
```

The diagnostic message would print as:

```
t_connect failed on fd2: Incorrect transport address format
```

In this example, `t_connect failed on fd2` tells the user which function was unsuccessful on which transport end point, and `Incorrect transport address format` identifies the specific error that occurred.

Implementation Specifics

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

Related Information

List of Streams Programming References, STREAMS Overview in *AIX Communications Programming Concepts*.

t_free Subroutine for Transport Layer Interface

Purpose

Frees a library structure.

Library

Transport Layer Interface Library (**libtli.a**)

Syntax

```
#include <tiuser.h>

int t_free(ptr, struct_type)
char *ptr;
int struct_type;
```

Description

The **t_free** subroutine frees memory previously allocated by the **t_alloc** subroutine. This subroutine frees memory for the specified structure and also frees memory for buffers referenced by the structure.

The **t_free** subroutine checks the `addr`, `opt`, and `udata` fields of the given structure (as appropriate) and frees the buffers pointed to by the `buf` field of the **netbuf** structure. If the `buf` field is null, the **t_free** subroutine does not attempt to free memory. After all buffers are freed, the **t_free** subroutine frees the memory associated with the structure pointed to by the *ptr* parameter.

Undefined results will occur if the *ptr* parameter or any of the `buf` pointers points to a block of memory that was not previously allocated by the **t_alloc** subroutine.

Parameters

ptr Points to one of the seven structure types described for the **t_alloc** subroutine.

struct_type

Identifies the type of that structure. The type can be one of the following:

Type	Structure
T_BIND	**struct t_bind**
T_CALL	**struct t_call**
T_OPTMGMT	**struct t_optmgmt**
T_DIS	**struct t_discon**
T_UNITDATA	**struct t_unitdata**
T_UDERROR	**struct t_uderr**
T_INFO	**struct t_info**

Each of these structure types is used as a parameter to one or more transport subroutines.

Return Values

On successful completion, the **t_free** subroutine returns a value of 0. Otherwise, it returns a value of –1, and the **t_errno** variable is set to indicate the error.

Error Codes

If unsuccessful, the **t_errno** variable is set to the following:

TNOSTRUCTYPE	Unsupported structure type requested.
TSYSERR	A system error has occurred during execution of this function.

Implementation Specifics

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

Related Information

The **t_alloc** subroutine.

List of Streams Programming References and STREAMS Overview in *AIX Communications Programming Concepts*.

t_getinfo Subroutine for Transport Layer Interface

Purpose

Gets protocol–specific service information.

Library

Transport Layer Interface Library (**libtli.a**)

Syntax

```
#include <tiuser.h>

int t_getinfo(fd, info)
int fd;
struct t_info *info;
```

Description

The **t_getinfo** subroutine returns the current characteristics of the underlying transport protocol associated with *fd* file descriptor. The **t_info** structure is used to return the same information returned by the **t_open** subroutine. This function enables a transport user to access this information during any phase of communication.

Parameters

fd Specifies the file descriptor.

info Points to a **t_info** structure that contains the following members:

```
long addr;
long options;
long tsdu;
long etsdu;
long connect;
long discon;
long servtype;
```

The values of the fields have the following meanings:

`addr` A value greater than or equal to 0 indicates the maximum size of a transport protocol address; a value of –1 specifies that there is no limit on the address size; and a value of –2 specifies that the transport provider does not provide user access to transport protocol addresses.

`options` A value greater than or equal to 0 indicates the maximum number of bytes of protocol–specific options supported by the provider; a value of –1 specifies that there is no limit on the option size; and a value of –2 specifies that the transport provider does not support user–settable options.

`tsdu` A value greater than 0 specifies the maximum size of a transport service data unit (TSDU); a value of 0 specifies that the transport provider does not support the concept of TSDU, although it does support the sending of a data stream having no logical boundaries preserved across a connection; a value of –1 specifies that there is no limit on the size of a TSDU; and a value of –2 specifies that the transfer of normal data is not supported by the transport provider.

`etsdu` A value greater than 0 specifies the maximum size of an expedited transport service data unit (ETSDU); a value of 0 specifies that the transport provider does not support the concept of ETSDU, although it does support the sending of an expedited data stream having no logical boundaries preserved across a connection; a value of –1 specifies that there is no limit

on the size of an ETSDU; and a value of –2 specifies that the transfer of expedited data is not supported by the transport provider.

`connect` A value greater than or equal to 0 specifies the maximum amount of data that may be associated with connection establishment functions; a value of –1 specifies that there is no limit on the amount of data sent during connection establishment; and a value of –2 specifies that the transport provider does not allow data to be sent with connection establishment functions.

`discon` A value greater than or equal to 0 specifies the maximum amount of data that may be associated with the **t_snddis** and **t_rcvdis** subroutines; a value of –1 specifies that there is no limit on the amount of data sent with these abortive release functions; and a value of –2 specifies that the transport provider does not allow data to be sent with the abortive release functions.

`servtype` This field specifies the service type supported by the transport provider.

If a transport user is concerned with protocol independence, the sizes may be accessed to determine how large the buffers must be to hold each piece of information. Alternatively, the **t_alloc** subroutine may be used to allocate these buffers. An error will result if a transport user exceeds the allowed data size on any function. The value of each field can change as a result of option negotiation; the **t_getinfo** subroutine enables a user to retrieve the current characteristics.

Return Values

On successful completion, the **t_getinfo** subroutine returns a value of 0. Otherwise, it returns a value of –1, and the **t_errno** variable is set to indicate the error.

The `servtype` field of the *info* parameter may specify one of the following values on return:

T_COTS The transport provider supports a connection–mode service, but does not support the optional orderly release facility.

T_COTS_ORD The transport provider supports a connection–mode service with the optional orderly release facility.

T_CLTS The transport provider supports a connectionless–mode service. For this service type, the **t_open** subroutine returns –2 for the values in the `etsdu`, `connect`, and `discon` fields.

Error Codes

In unsuccessful, the **t_errno** variable is set to one of the following:

TBADF The specified file descriptor does not refer to a transport end point.

TSYSERR A system error has occurred during execution of this function.

Implementation Specifics

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

Related Information

The **t_alloc** subroutine, **t_open** subroutine, **t_rcvdis** subroutine and **t_snddis** subroutine.

List of Streams Programming Reference and STREAMS Overview in *AIX Communications Programming Concepts*.

t_getstate Subroutine for Transport Layer Interface

Purpose

Gets the current state.

Library

Transport Layer Interface Library (**libtli.a**)

Syntax

```
#include <tiuser.h>

int t_getstate(fd)
int fd;
```

Description

The **t_getstate** subroutine returns the current state of the provider associated with the transport end point specified by the *fd* parameter.

Parameter

fd	Specifies the transport end point.

Return Codes

On successful completion, the **t_getstate** subroutine returns the current state. Otherwise, it returns a value of –1, and the **t_errno** variable is set to indicate the error.

If the provider is undergoing a state transition when the **t_getstate** subroutine is called, the function will fail. The current state is one of the following.

T_DATAXFER	Data transfer.
T_IDLE	Idle.
T_INCON	Incoming connection pending.
T_INREL	Incoming orderly release (waiting to send an orderly release indication).
T_OUTCON	Outgoing connection pending.
T_OUTREL	Outgoing orderly release (waiting for an orderly release indication).
T_UNBND	Unbound.

Error Codes

If unsuccessful, the **t_errno** variable is set to one of the following:

TBADF	The specified file descriptor does not refer to a transport end point.
TSTATECHNG	The transport provider is undergoing a state change.
TSYSERR	A system error has occurred during execution of this function.

Implementation Specifics

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

Related Information

The **t_open** subroutine.

List of Streams Programming References and STREAMS Overview in *AIX Communications Programming Concepts*.

t_listen Subroutine for Transport Layer Interface

Purpose

Listens for a connect request.

Library

Transport Layer Interface Library (**libtli.a**)

Syntax

```
#include <tiuser.h>

int t_listen(fd, call)
int fd;
struct t_call *call;
```

Description

The **t_listen** subroutine listens for a connect request from a calling transport user.

Note: If a user issues a **t_listen** subroutine call in synchronous mode on a transport end point that was not bound for listening (that is, the `qlen` field was 0 on the **t_bind** subroutine), the call will never return because no connect indications will arrive on that endpoint.

Parameters

fd Identifies the local transport endpoint where connect indications arrive.

call

Contains information describing the connect indication.

The *call* parameter points to a **t_call** structure that contains the following members:

```
struct netbuf addr;
struct netbuf opt;
struct netbuf udata;
int sequence;
```

The **netbuf** structure contains the following fields:

`addr` Returns the protocol address of the calling transport user.

`opt` Returns protocol–specific parameters associated with the connect request.

`udata` Returns any user data sent by the caller on the connect request.

`sequence` Uniquely identifies the returned connect indication. The value of `sequence` enables the user to listen for multiple connect indications before responding to any of them.

Since the **t_listen** subroutine returns values for the `addr`, `opt`, and `udata` fields of the *call* parameter, the `maxlen` field of each must be set before issuing the **t_listen** subroutine to indicate the maximum size of the buffer for each.

By default, the **t_listen** subroutine executes in synchronous mode and waits for a connect indication to arrive before returning to the user. However, if the **O_NDELAY** or

O_NONBLOCK flag is set (using the **t_open** subroutine or the **fcntl** command), the **t_listen** subroutine executes asynchronously, reducing to a poll for existing connect indications. If none are available, the **t_listen** subroutine returns –1 and sets the **t_errno** variable to **TNODATA**.

Return Values

On successful completion, the **t_listen** subroutine returns a value of 0. Otherwise, it returns a value of –1, and the **t_errno** variable is set to indicate the error.

Error Codes

If unsuccessful, the **t_errno** variable is set to one of the following:

TBADF	The specified file descriptor does not refer to a transport end point.
TBADQLEN	The transport end point is not bound for listening. The *qlen* is zero.
TBUFOVFLW	The number of bytes allocated for an incoming argument is not sufficient to store the value of that argument. The provider's state, as seen by the user, changes to **T_INCON**, and the connect–indication information to be returned in the *call* parameter is discarded.
TLOOK	An asynchronous event has occurred on this transport end point and requires immediate attention.
TNODATA	The **O_NDELAY** or **O_NONBLOCK** flag was set, but no connect indications had been queued.
TNOTSUPPORT	This function is not supported by the underlying transport provider.
TOUTSTATE	The subroutine was issued in the wrong sequence.
TSYSERR	A system error has occurred during execution of this function.

Implementation Specifics

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

Related Information

The **t_accept** subroutine, **t_alloc** subroutine, **t_bind** subroutine, **t_connect** subroutine, **t_open** subroutine, **t_rcvconnect** subroutine.

List of Streams Programming References and STREAMS Overview in *AIX Communications Programming Concepts*.

t_look Subroutine for Transport Layer Interface

Purpose

Looks at the current event on a transport end point.

Library

Transport Layer Interface Library (**libtli.a**)

Syntax

```
#include <tiuser.h>

int t_look(fd)
int fd;
```

Description

The **t_look** subroutine returns the current event on the transport end point specified by the *fd* parameter. This subroutine enables a transport provider to notify a transport user of an asynchronous event when the user is issuing functions in synchronous mode. Certain events require immediate notification of the user and are indicated by a specific error, **TLOOK**, on the current or next subroutine executed.

This subroutine also enables a transport user to poll a transport end point periodically for asynchronous events.

Parameter

fd	Specifies the transport end point.

Return Values

On successful completion, the **t_look** subroutine returns a value that indicates which of the allowable events has occurred, or returns a value of 0 if no event exists. One of the following events is returned:

T_CONNECT	Indicates connect confirmation received.
T_DATA	Indicates normal data received.
T_DISCONNECT	Indicates disconnect received.
T_ERROR	Indicates fatal error.
T_EXDATA	Indicates expedited data received.
T_LISTEN	Indicates connection indication received.
T_ORDREL	Indicates orderly release.
T_UDERR	Indicates datagram error.

If the **t_look** subroutine is unsuccessful, a value of –1 is returned, and the **t_errno** variable is set to indicate the error.

Error Codes

If unsuccessful, the **t_errno** variable is set to one of the following:

TBADF	The specified file descriptor does not refer to a transport end point.
TSYSERR	A system error has occurred during execution of this function.

Implementation Specifics

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

Related Information

The **t_open** subroutine.

List of Streams Programming References and STREAMS Overview in *AIX Communications Programming Concepts*.

t_open Subroutine for Transport Layer Interface

Purpose

Establishes a transport end point.

Library

Transport Layer Interface Library (**libtli.a**)

Syntax

```
#include <tiuser.h>

int t_open(path, oflag, info)
char *path;
int oflag;
struct t_info *info;
```

Description

The **t_open** subroutine must be called as the first step in the initialization of a transport end point. This subroutine establishes a transport end point, first, by opening a UNIX system file that identifies a particular transport provider (that is, transport protocol) and then returning a file descriptor that identifies that end point. For example, opening the **/dev/dlpi/tr** file identifies an 802.5 data link provider.

Parameters

path Points to the path name of the file to open.

oflag Specifies the open routine flags.

info Points to a **t_info** structure.

The *info* parameter points to a **t_info** structure that contains the following elements:

```
long addr;
long options;
long tsdu;
long etsdu;
long connect;
long discon;
long servtype;
```

The values of the elements have the following meanings:

`addr` A value greater than or equal to 0 indicates the maximum size of a transport protocol address; a value of –1 specifies that there is no limit on the address size; and a value of –2 specifies that the transport provider does not provide user access to transport protocol addresses.

`options` A value greater than or equal to 0 indicates the maximum number of bytes of protocol–specific options supported by the provider; a value of –1 specifies that there is no limit on the option size; and a value of –2 specifies that the transport provider does not support user–settable options.

`tsdu` A value greater than 0 specifies the maximum size of a transport service data unit (TSDU); a value of 0 specifies that the transport provider does not support the concept of TSDU, although it does support the sending of a data stream having no logical boundaries preserved across a connection; a value of –1 specifies that there is no limit on the size of a TSDU; and a value of –2 specifies that the transfer of normal data is not supported by the transport provider.

`etsdu` A value greater than 0 specifies the maximum size of a expedited transport service data unit (ETSDU); a value of 0 specifies that the transport provider does not support the concept of ETSDU, although it does support the sending of an expedited data stream having no logical boundaries preserved across a connection; a value of –1 specifies that there is no limit on the size of an ETSDU; and a value of –2 specifies that the transfer of expedited data is not supported by the transport provider.

`connect` A value greater than or equal to 0 specifies the maximum amount of data that may be associated with connection establishment functions; a value of –1 specifies that there is no limit on the amount of data sent during connection establishment; and a value of –2 specifies that the transport provider does not allow data to be sent with connection establishment functions.

`discon` A value greater than or equal to 0 specifies the maximum amount of data that may be associated with the **t_snddis** and **t_rcvdis** functions; a value of –1 specifies that there is no limit on the amount of data sent with these abortive release functions; and a value of –2 specifies that the transport provider does not allow data to be sent with the abortive release functions.

`servtype` This field specifies the service type supported by the transport provider, as described in the Return Values section.

If a transport user is concerned with protocol independence, these sizes may be accessed to determine how large the buffers must be to hold each piece of information. Alternatively, the **t_alloc** subroutine can be used to allocate these buffers. An error results if a transport user exceeds the allowed data size on any function.

Return Values

On successful completion, the **t_open** subroutine returns a valid file descriptor. Otherwise, it returns a value of –1, and the **t_errno** variable is set to indicate the error.

The `servtype` field of the *info* parameter can specify one of the following values on return:

T_COTS The transport provider supports a connection–mode service but does not support the optional orderly release facility.

T_COTS_ORD The transport provider supports a connection–mode service with the optional orderly release facility.

T_CLTS The transport provider supports a connectionless–mode service. For this service type, the **t_open** subroutine returns –2 for the values in the `etsdu`, `connect`, and `discon` fields.

A single transport end point can support only one of the above services at one time.

If the *info* parameter is set to null by the transport user, no protocol information is returned by the **t_open** subroutine.

Error Codes

If unsuccessful, the **t_errno** variable is set to the following:

TSYSERR A system error has occurred during the startup of this function.

Implementation Specifics

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

Related Information

The **open** subroutine, **t_close** subroutine.

List of Streams Programming References and STREAMS Overview in *AIX Communications Programming Concepts*.

t_optmgmt Subroutine for Transport Layer Interface

Purpose

Manages options for a transport end point.

Library

Transport Layer Interface Library (**libtli.a**)

Syntax

```
#include <tiuser.h>

int t_optmgmt(fd, req, ret)
int fd;
struct t_optmgmt *req;
struct t_optmgmt *ret;
```

Description

The **t_optmgmt** subroutine enables a transport user to retrieve, verify, or negotiate protocol options with the transport provider.

Parameters

fd	Identifies a bound transport end point.
req	Requests a specific action of the provider.
ret	Returns options and flag values to the user.

Both the *req* and *ret* parameters point to a **t_optmgmt** structure containing the following members:

```
struct netbuf opt;
long flags;
```

The `opt` field identifies protocol options, and the `flags` field specifies the action to take with those options.

The options are represented by a **netbuf** structure in a manner similar to the address in the **t_bind** subroutine. The *req* parameter is used to send options to the provider. This **netbuf** structure contains the following fields:

`len`	Specifies the number of bytes in the options.
`buf`	Points to the options buffer.
`maxlen`	Has no meaning for the *req* parameter.

The *ret* parameter is used to return information to the user from the transport provider. On return, this **netbuf** structure contains the following fields:

`len`	Specifies the number of bytes of options returned.
`buf`	Points to the buffer where the options are to be placed.
`maxlen`	Specifies the maximum size of the options buffer. The `maxlen` field has no meaning for the *req* parameter, but must be set in the *ret* parameter to specify the maximum number of bytes the options buffer can hold. The actual structure and content of the options is imposed by the transport provider.

The `flags` field of the *req* parameter can specify one of the following actions:

T_NEGOTIATE Enables the user to negotiate the values of the options specified in the *req* parameter with the transport provider. The provider evaluates the requested options and negotiates the values, returning the negotiated values through the *ret* parameter.

T_CHECK Enables the user to verify if the options specified in the *req* parameter are supported by the transport provider. On return, the `flags` field of the *ret* parameter has either **T_SUCCESS** or **T_FAILURE** set to indicate to the user whether the options are supported or not. These flags are only meaningful for the **T_CHECK** request.

T_DEFAULT Enables a user to retrieve the default options supported by the transport provider into the `opt` field of the *ret* parameter. In the *req* parameter, the `len` field of the `opt` field must be zero, and the `buf` field can be NULL.

If issued as part of the connectionless–mode service, the **t_optmgmt** subroutine may become blocked due to flow control constraints. The subroutine does not complete until the transport provider has processed all previously sent data units.

Return Values

On successful completion, the **t_optmgmt** subroutine returns a value of 0. Otherwise, it returns a value of –1, and the **t_errno** variable is set to indicate the error.

Error Codes

If unsuccessful, the **t_errno** variable is set to one of the following:

TACCES User does not have permission to negotiate the specified options.

TBADF Specified file descriptor does not refer to a transport endpoint.

TBADFLAG Unusable flag was specified.

TBADOPT Specified protocol options were in an incorrect format or contained unusable information.

TBUFOVFLW Number of bytes allowed for an incoming parameter is not sufficient to store the value of that parameter. Information to be returned in the *ret* parameter will be discarded.

TOUTSTATE Function was issued in the wrong sequence.

TSYSERR A system error has occurred during operation of this subroutine.

Implementation Specifics

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

Related Information

The **t_getinfo** subroutine, **t_open** subroutine.

List of Streams Programming References and STREAMS Overview in *AIX Communications Programming Concepts*.

t_rcv Subroutine for Transport Layer Interface

Purpose

Receives normal data or expedited data sent over a connection.

Library

Transport Layer Interface Library (**libtli.a**)

Syntax

```
int t_rcv(fd, buf, nbytes, flags)
int fd;
char *buf;
unsigned nbytes;
int *flags;
```

Description

The **t_rcv** subroutine receives either normal or expedited data. By default, the **t_rcv** subroutine operates in synchronous mode and will wait for data to arrive if none is currently available. However, if the **O_NDELAY** flag is set (using the **t_open** subroutine or the **fcntl** command), the **t_rcv** subroutine runs in asynchronous mode and will stop if no data is available.

On return from the call, if the **T_MORE** flag is set in the *flags* parameter, this indicates that there is more data. This means that the current transport service data unit (TSDU) or expedited transport service data unit (ETSDU) must be received in multiple **t_rcv** subroutine calls. Each **t_rcv** subroutine with the **T_MORE** flag set indicates that another **t_rcv** subroutine must follow immediately to get more data for the current TSDU. The end of the TSDU is identified by the return of a **t_rcv** subroutine call with the **T_MORE** flag not set. If the transport provider does not support the concept of a TSDU as indicated in the *info* parameter on return from a **t_open** or **t_getinfo** subroutine, the **T_MORE** flag is not meaningful and should be ignored.

On return, the data returned is expedited data if the **T_EXPEDITED** flag is set in the *flags* parameter. If the number of bytes of expedited data exceeds the value in the *nbytes* parameter, the **t_rcv** subroutine will set the **T_EXPEDITED** and **T_MORE** flags on return from the initial call. Subsequent calls to retrieve the remaining ETSDU not have the **T_EXPEDITED** flag set on return. The end of the ETSDU is identified by the return of a **t_rcv** subroutine call with the **T_MORE** flag not set.

If expedited data arrives after part of a TSDU has been retrieved, receipt of the remainder of the TSDU will be suspended until the ETSDU has been processed. Only after the full ETSDU has been retrieved (the **T_MORE** flag is not set) will the remainder of the TSDU be available to the user.

Parameters

fd	Identifies the local transport end point through which data will arrive.
buf	Points to a receive buffer where user data will be placed.
nbytes	Specifies the size of the receiving buffer.
flags	Specifies optional flags.

Return Values

On successful completion, the **t_rcv** subroutine returns the number of bytes it received. Otherwise, it returns a value of –1 and sets the **t_errno** variable to indicate the error.

Error Codes

If unsuccessful, the **t_errno** variable may be set to one of the following:

TBADF	The specified file descriptor does not refer to a transport end point.
TLOOK	An asynchronous event has occurred on this transport end point and requires immediate attention.
TNODATA	The **O_NDELAY** flag was set, but no data is currently available from the transport provider.
TNOTSUPPORT	This subroutine is not supported by the underlying transport provider.
TOUTSTATE	The subroutine was issued in the wrong sequence.
TSYSERR	A system error has occurred during operation of this subroutine.

Implementation Specifics

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

Related Information

The **t_getinfo** subroutine, **t_look** subroutine, **t_open** subroutine, **t_snd** subroutine.

List of Streams Programming References and STREAMS Overview in *AIX Communications Programming Concepts*.

t_rcvconnect Subroutine for Transport Layer Interface

Purpose

Receives the confirmation from a connect request.

Library

Transport Layer Interface Library (**libtli.a**)

Syntax

```
#include <tiuser.h>

int t_rcvconnect(fd, call)
int fd;
struct t_call *call;
```

Description

The **t_rcvconnect** subroutine enables a calling transport user to determine the status of a previously sent connect request and is used in conjunction with **t_connect** to establish a connection in asynchronous mode. The connection will be established on successful completion of this function.

Parameters

fd Identifies the local transport end point where communication will be established.

call

Contains information associated with the newly established connection.

The *call* parameter points to a **t_call** structure that contains the following elements:

```
struct netbuf addr;
struct netbuf opt;
struct netbuf udata;
int sequence;
```

The **netbuf** structure contains the following elements:

`addr` Returns the protocol address associated with the responding transport end point.

`opt` Presents protocol–specific information associated with the connection.

`udata` Points to optional user data that may be returned by the destination transport user during connection establishment.

`sequence` Has no meaning for this function.

The `maxlen` field of each parameter must be set before issuing this function to indicate the maximum size of the buffer for each. However, the *call* parameter may be null, in which case no information is given to the user on return from the **t_rcvconnect** subroutine. By default, the **t_rcvconnect** subroutine runs in synchronous mode and waits for the connection to be established before returning. On return, the `addr`, `opt`, and `udata` fields reflect values associated with the connection.

If the **O_NDELAY** flag is set (using the **t_open** subroutine or **fcntl** command), the **t_rcvconnect** subroutine runs in asynchronous mode and reduces to a poll for existing

connect confirmations. If none are available, the **t_rcvconnect** subroutine stops and returns immediately without waiting for the connection to be established. The **t_rcvconnect** subroutine must be re–issued at a later time to complete the connection establishment phase and retrieve the information returned in the *call* parameter.

Return Values

On successful completion, the **t_rcvconnect** subroutine returns a value of 0. Otherwise, it returns a value of –1 and sets the **t_errno** variable to indicate the error.

Error Codes

If unsuccessful, the **t_errno** variable may be set to one of the following:

TBADF	The specified file descriptor does not refer to a transport end point.
TBUFOVFLW	The number of bytes allocated for an incoming parameter is not sufficient to store the value of that parameter and the connect information to be returned in the *call* parameter will be discarded. The state of the provider, as seen by the user, will be changed to **DATAXFER**.
TLOOK	An asynchronous event has occurred on this transport connection and requires immediate attention.
TNODATA	The **O_NDELAY** flag was set, but a connect confirmation has not yet arrived.
TNOTSUPPORT	This subroutine is not supported by the underlying transport provider.
TOUTSTATE	This subroutine was issued in the wrong sequence.
TSYSERR	A system error has occurred during operation of this subroutine.

Implementation Specifics

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

Related Information

The **t_accept** subroutine, **t_alloc** subroutine, **t_bind** subroutine, **t_connect** subroutine, **t_listen** subroutine, **t_look** subroutine, **t_open** subroutine.

List of Streams Programming References and STREAMS Overview in *AIX Communications Programming Concepts*.

t_rcvdis Subroutine for Transport Layer Interface

Purpose

Retrieves information from disconnect.

Library

Transport Layer Interface Library (**libtli.a**)

Syntax

```
#include <tiuser.h>

t_rcvdis(fd, discon)
int fd;
struct t_discon *discon
```

Description

The **t_rcvdis** subroutine is used to identify the cause of a disconnect, and to retrieve any user data sent with the disconnect.

Parameters

fd Identifies the local transport end point where the connection existed.

discon

Points to a **t_discon** structure that contains the reason for the disconnect and contains any user data that was sent with the disconnect.

The **t_discon** structure contains the following members:

```
struct netbuf udata;
int reason;
int sequence;
```

These fields are defined as follows:

`reason` Specifies the reason for the disconnect through a protocol–dependent reason code.

`udata` Identifies any user data that was sent with the disconnect.

`sequence` Identifies an outstanding connect indication with which the disconnect is associated. The `sequence` field is only meaningful when the **t_rcvdis** subroutine is issued by a passive transport user that has called one or more **t_listen** subroutines and is processing the resulting connect indications. If a disconnect indication occurs, the `sequence` field can be used to identify which of the outstanding connect indications is associated with the disconnect.

If a user does not care if there is incoming data and does not need to know the value of the `reason` or `sequence` fields, the *discon* parameter may be null and any user data associated with the disconnect will be discarded. However, if a user has retrieved more than one outstanding connect indication (using the **t_listen** subroutine) and the *discon* parameter is null, the user will be unable to identify with which connect indication the disconnect is associated.

Return Values

On successful completion, the **t_rcvdis** subroutine returns a value of 0. Otherwise, it returns a value of –1 and sets the **t_errno** variable to indicate the error.

Error Codes

If unsuccessful, the **t_errno** variable may be set to one of the following:

TBADF	The specified file descriptor does not refer to a transport end point.
TBUFOVFLW	The number of bytes allocated for incoming data is not sufficient to store the data. (The state of the provider, as seen by the user, will change to **T_IDLE**, and the disconnect indication information to be returned in the *discon* parameter will be discarded.)
TLOOK	An asynchronous event has occurred on this transport endpoint and requires immediate attention.
TNODIS	No disconnect indication currently exists on the specified transport end point.
TNOTSUPPORT	This function is not supported by the underlying transport provider.
TOUTSTATE	This subroutine was issued in the wrong sequence.
TSYSERR	A system error has occurred during execution of this subroutine.

Implementation Specifics

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

Related Information

The **t_alloc** subroutine, **t_connect** subroutine, **t_listen** subroutine, **t_open** subroutine, **t_snddis** subroutine.

List of Streams Programming References and STREAMS Overview in *AIX Communications Programming Concepts*.

t_rcvrel Subroutine for Transport Layer Interface

Purpose

Acknowledges receipt of an orderly release indication.

Library

Transport Layer Interface Library (**libtli.a**)

Syntax

```
#include <tiuser.h>

t_rcvrel(fd)
int fd;
```

Description

The **t_rcvrel** subroutine is used to acknowledge receipt of an orderly release indication. After receipt of this indication, the user may not attempt to receive more data because such an attempt will block forever. However, the user may continue to send data over the connection if the **t_sndrel** subroutine has not been issued by the user. The subroutine is an optional service of the transport provider, and is only supported if the transport provider returned service type **T_COTS_ORD** on the **t_open** or **t_getinfo** subroutine.

Parameter

fd	Identifies the local transport end point where the connection exists.

Return Values

On successful completion, the **t_rcvrel** subroutine returns a value of 0. Otherwise, it returns a value of –1 and sets the **t_errno** variable to indicate the error.

Error Codes

If unsuccessful, the **t_errno** variable is set to one of the following:

TBADF	The specified file descriptor does not refer to a transport end point.
TLOOK	An asynchronous event has occurred on this transport end point and requires immediate attention.
TNOREL	No orderly release indication currently exists on the specified transport end point.
TNOTSUPPORT	This subroutine is not supported by the underlying transport provider.
TOUTSTATE	This subroutine was issued in the wrong sequence.
TSYSERR	A system error has occurred during execution of this function.

Implementation Specifics

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

Related Information

The **t_getinfo** subroutine, **t_look** subroutine, **t_open** subroutine, **t_sndrel** subroutine.

List of Streams Programming References and STREAMS Overview in *AIX Communications Programming Concepts*.

t_rcvudata Subroutine for Transport Layer Interface

Purpose

Receives a data unit.

Library

Transport Layer Interface Library (**libtli.a**)

Syntax

```
#include <tiuser.h>

int t_rcvudata(fd, unitdata, flags)
int fd;
struct t_unitdata *unitdata;
int *flags;
```

Description

The **t_rcvudata** subroutine is used in connectionless mode to receive a data unit from another transport user.

Parameters

fd Identifies the local transport end point through which data will be received.

unitdata

Holds information associated with the received data unit.

The *unitdata* parameter points to a **t_unitdata** structure containing the following members:

```
struct netbuf addr;
struct netbuf opt;
struct netbuf udata;
```

On return from this call:

`addr` Specifies the protocol address of the sending user.

`opt` Identifies protocol–specific options that were associated with this data unit.

`udata` Specifies the user data that was received.

Note: The `maxlen` field of the `addr`, `opt`, and `udata` fields must be set before issuing this function to indicate the maximum size of the buffer for each.

flags Indicates that the complete data unit was not received.

By default, the **t_rcvudata** subroutine operates in synchronous mode and will wait for a data unit to arrive if none is currently available. However, if the **O_NDELAY** or **O_NONBLOCK** flag is set (using the **t_open** subroutine or **fcntl** command), the **t_rcvudata** subroutine will run in asynchronous mode and will stop if no data units are available.

If the buffer defined in the `udata` field of *unitdata* is not large enough to hold the current data unit, the buffer will be filled and the **T_MORE** flag will be set in *flags* on return to indicate that another **t_rcvudata** subroutine should be issued to retrieve the rest of the data unit. Subsequent **t_rcvudata** subroutine calls will return 0 for the length of the address and options until the full data unit has been received.

Return Values

On successful completion, the **t_rcvudata** subroutine returns a value of 0. Otherwise, it returns a value of –1 and sets the **t_errno** variable to indicate the error.

Error Codes

If unsuccessful, the **t_errno** variable is set to one of the following:

TBADF	The specified file descriptor does not refer to a transport end point.
TBUFOVFLW	The number of bytes allocated for the incoming protocol address of options is not sufficient to store the information. (The unit data information to be returned in the *unitdata* parameter will be discarded.)
TLOOK	An asynchronous event has occurred on this transport endpoint and requires immediate attention.
TNODATA	The **O_DELAY** or **O_NONBLOCK** flag was set, but no data units are currently available from the transport provider.
TNOTSUPPORT	This subroutine is not supported by the underlying transport provider.
TOUTSTATE	The subroutine was issued in the wrong sequence.
TSYSERR	A system error has occurred during operation of this subroutine.

Implementation Specifics

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

Related Information

The **t_alloc** subroutine, **t_open** subroutine, **t_rcvuderr** subroutine, **t_sndudata** subroutine.

List of Streams Programming References and STREAMS Overview in *AIX Communications Programming Concepts*.

t_rcvuderr Subroutine for Transport Layer Interface

Purpose

Receives a unit data error indication.

Library

Transport Layer Interface Library (**libtli.a**)

Syntax

```
#include <tiuser.h>

int t_rcvuderr(fd, uderr)
int fd;
struct t_uderr *uderr;
```

Description

The **t_rcvuderr** subroutine is used in connectionless mode to receive information concerning an error on a previously sent data unit, and should only be issued following a unit data error indication. It informs the transport user that a data unit with a specific destination address and protocol options produced an error.

Parameters

fd Identifies the local transport endpoint through which the error report will be received.

uderr

Points to a **t_uderr** structure containing the following members:

```
struct netbuf addr;
struct netbuf opt;
long error;
```

The `maxlen` field of the `addr` and `opt` fields must be set before issuing this function to indicate the maximum size of the buffer for each.

On return from this call, the **t_uderr** structure contains:

`addr` Specifies the destination protocol address of the erroneous data unit.

`opt` Identifies protocol–specific options that were associated with the data unit.

`error` Specifies a protocol–dependent error code.

If the user decides not to identify the data unit that produced an error, the *uderr* parameter can be set to null and the **t_rcvuderr** subroutine will clear the error indication without reporting any information to the user.

Return Values

On successful completion, the **t_rcvuderr** subroutine returns a value of 0. Otherwise, it returns a value of –1 and sets the **t_errno** variable to indicate the error.

Error Codes

If unsuccessful, the **t_errno** variable is set to one of the following:

TBADF	The specified file descriptor does not refer to a transport end point.
TNOUDERR	No unit data error indication currently exists on the specified transport end point.
TBUFOVFLW	The number of bytes allocated for the incoming protocol address or options is not sufficient to store the information. (The unit data error information to be returned in the *uderr* parameter will be discarded.)
TNOTSUPPORT	This subroutine is not supported by the underlying transport provider.
TSYSERR	A system error has occurred during execution of this subroutine.

Implementation Specifics

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

Related Information

The **t_look** subroutine, **t_rcvudata** subroutine, **t_sndudata** subroutine.

List of Streams Programming References and STREAMS Overview in *AIX Communications Programming Concepts*.

t_snd Subroutine for Transport Layer Interface

Purpose

Sends data or expedited data over a connection.

Library

Transport Layer Interface Library (**libtli.a**)

Syntax

```
#include <tiuser.h>

int t_snd(fd, buf, nbytes, flags)
int fd;
char *buf;
unsigned nbytes;
int flags;
```

Description

The **t_snd** subroutine is used to send either normal or expedited data.

By default, the **t_snd** subroutine operates in synchronous mode and may wait if flow–control restrictions prevent the data from being accepted by the local transport provider at the time the call is made. However, if the **O_NDELAY** or **O_NONBLOCK** flag is set (using the **t_open** subroutine or the **fcntl** command), the **t_snd** subroutine runs in asynchronous mode and stops immediately if there are flow–control restrictions.

Even when there are no flow–control restrictions, the **t_snd** subroutine will wait if STREAMS internal resources are not available, regardless of the state of the **O_NDELAY** or **O_NONBLOCK** flag.

On successful completion, the **t_snd** subroutine returns the number of bytes accepted by the transport provider. Normally this equals the number of bytes specified in the *nbytes* parameter. However, if the **O_NDELAY** or **O_NONBLOCK** flag is set, it is possible that only part of the data will be accepted by the transport provider. In this case, the **t_snd** subroutine sets the **T_MORE** flag for the data that was sent and returns a value less than the value of the *nbytes* parameter. If the value of the *nbytes* parameter is 0, no data is passed to the provider and the **t_snd** subroutine returns a value of 0.

Parameters

fd	Identifies the local transport end point through which data is sent.
buf	Points to the user data.

nbytes Specifies the number of bytes of user data to be sent.

flags Specifies any optional flags.

If the **T_EXPEDITED** flag is set in the *flags* parameter, the data is sent as expedited data and is subject to the interpretations of the transport provider.

If the **T_MORE** flag is set in the *flags* parameter, or as described above, an indication is sent to the transport provider that the transport service data unit (TSDU) or expedited transport service data unit (ETSDU) is being sent through multiple **t_snd** subroutine calls. Each **t_snd** subroutine with the **T_MORE** flag set indicates that another **t_snd** subroutine will follow with more data for the current TSDU. The end of the TSDU or ETSDU is identified by a **t_snd** subroutine call with the **T_MORE** flag not set. Use of the **T_MORE** flag enables a user to break up large logical data units without losing the boundaries of those units at the other end of the connection. The flag implies nothing about how the data is packaged for transfer below the transport interface. If the transport provider does not support the concept of a TSDU as indicated in the *info* parameter on return from the **t_open** or **t_getinfo** subroutine, the **T_MORE** flag is not meaningful and should be ignored.

The size of each TSDU or ETSDU must not exceed the limits of the transport provider as returned by the **t_open** or **t_getinfo** subroutine. If the size is exceeded, a **TSYSERR** error with system error **EPROTO** occurs. However, the **t_snd** subroutine may not fail because **EPROTO** errors may not be reported immediately. In this case, a subsequent call that accesses the transport endpoint fails with the associated **TSYSERR** error.

If the call to the **t_snd** subroutine is issued from the **T_IDLE** state, the provider may silently discard the data. If the call to the **t_snd** subroutine is issued from any state other than **T_DATAXFER**, **T_INREL**, or **T_IDLE**, the provider generates a **TSYSERR** error with system error **EPROTO** (which can be reported in the manner described above).

Return Values

On successful completion, the **t_snd** subroutine returns the number of bytes accepted by the transport provider. Otherwise, it returns a value of –1 and sets the **t_errno** variable to indicate the error.

Error Codes

If unsuccessful, the **t_errno** variable is set to one of the following:

TBADDATA The amount of user data specified was not within the bounds allowed by the transport provider.

TBADF The specified file descriptor does not refer to a transport end point.

TBADFLAG The value specified in the *flags* parameter is invalid.

TFLOW The **O_NDELAY** or **O_NONBLOCK** flag was set, but the flow–control mechanism prevented the transport provider from accepting data at this time.

TLOOK An asynchronous event has occurred on the transport end point reference by the *fd* parameter and requires immediate attention.

TNOTSUPPORT This subroutine is not supported by the underlying transport provider.

TOUTSTATE The subroutine was issued in the wrong sequence.

TSYSERR A system error has been detected during execution of this subroutine.

Implementation Specifics

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

Related Information

The **t_getinfo** subroutine, **t_getstate** subroutine, **t_open** subroutine, **t_rcv** subroutine.

List of Streams Programming References and STREAMS Overview in *AIX Communications Programming Concepts*.

t_snddis Subroutine for Transport Layer Interface

Purpose

Sends a user–initiated disconnect request.

Library

Transport Layer Interface Library (**libtli.a**)

Syntax

```
#include <tiuser.h>

int t_snddis(fd, call)
int fd;
struct t_call *call;
```

Description

The **t_snddis** subroutine is used to initiate an abortive release on an already established connection or to reject a connect request.

Parameters

fd Identifies the local transport end point of the connection.

call Specifies information associated with the abortive release.

The *call* parameter points to a **t_call** structure containing the following fields:

```
struct netbuf addr;
struct netbuf opt;
struct netbuf udata;
int sequence;
```

The values in the *call* parameter have different semantics, depending on the context of the call to the **t_snddis** subroutine. When rejecting a connect request, the *call* parameter must not be null and must contain a valid value in the `sequence` field to uniquely identify the rejected connect indication to the transport provider. The `addr` and `opt` fields of the *call* parameter are ignored. In all other cases, the *call* parameter need only be used when data is being sent with the disconnect request. The `addr`, `opt`, and `sequence` fields of the **t_call** structure are ignored. If the user does not wish to send data to the remote user, the value of the *call* parameter can be null.

The `udata` field specifies the user data to be sent to the remote user. The amount of user data must not exceed the limits supported by the transport provider as returned by the **t_open** or **t_getinfo** subroutine. If the `len` field of the `udata` field is 0, no data will be sent to the remote user.

Return Values

On successful completion, the **t_snddis** subroutine returns a value of 0. Otherwise, it returns a value of –1 and sets the **t_errno** variable to indicate the error.

Error Codes

If unsuccessful, the **t_errno** variable is set to one of the following:

TBADF	The specified file descriptor does not refer to a transport end point.
TOUTSTATE	The function was issued in the wrong sequence. The transport provider's outgoing queue may be flushed, so data may be lost.
TBADDATA	The amount of user data specified was not within the bounds allowed by the transport provider. The transport provider's outgoing queue will be flushed, so data may be lost.
TBADSEQ	An incorrect sequence number was specified, or a null call structure was specified when rejecting a connect request. The transport provider's outgoing queue will be flushed, so data may be lost.
TLOOK	An asynchronous event has occurred on this transport endpoint and requires immediate attention.
TNOTSUPPORT	This subroutine is not supported by the underlying transport provider.
TSYSERR	A system error has occurred during execution of this subroutine.

Implementation Specifics

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

Related Information

The **t_connect** subroutine, **t_getinfo** subroutine, **t_listen** subroutine, **t_look** subroutine, **t_open** subroutine.

List of Streams Programming References and STREAMS Overview in *AIX Communications Programming Concepts*.

t_sndrel Subroutine for Transport Layer Interface

Purpose

Initiates an orderly release of a transport connection.

Library

Transport Layer Interface Library (**libtli.a**)

Syntax

```
#include <tiuser.h>

int t_sndrel(fd)
int fd;
```

Description

The **t_sndrel** subroutine is used to initiate an orderly release of a transport connection and indicates to the transport provider that the transport user has no more data to send.

After issuing a **t_sndrel** subroutine call, the user cannot send any more data over the connection. However, a user can continue to receive data if an orderly release indication has been received.

The **t_sndrel** subroutine is an optional service of the transport provider and is only supported if the transport provider returned service type **T_COTS_ORD** in the **t_open** or **t_getinfo** subroutine.

Parameter

fd	Identifies the local transport endpoint where the connection exists.

Return Values

On successful completion, the **t_sndrel** subroutine returns a value of 0. Otherwise, it returns a value of –1 and sets the **t_errno** variable to indicate the error.

Error Codes

If unsuccessful, the **t_errno** variable is set to one of the following:

TBADF	The specified file descriptor does not refer to a transport endpoint.
TFLOW	The **O_NDELAY** or **O_NONBLOCK** flag was set, but the flow–control mechanism prevented the transport provider from accepting the function at this time.
TLOOK	An asynchronous event has occurred on the transport end point reference by the *fd* parameter and requires immediate attention.
TNOTSUPPORT	This subroutine is not supported by the underlying transport provider.
TOUTSTATE	The subroutine was issued in the wrong sequence.
TSYSERR	A system error has occurred during execution of this subroutine.

Implementation Specifics

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

Related Information

The **t_getinfo** subroutine, **t_open** subroutine, **t_rcvrel** subroutine.

List of Streams Programming References and STREAMS Overview in *AIX Communications Programming Concepts*.

t_sndudata Subroutine for Transport Layer Interface

Purpose

Sends a data unit to another transport user.

Library

Transport Layer Interface Library (**libtli.a**)

Syntax

```
#include <tiuser.h>

int t_sndudata(fd, unitdata)
int fd;
struct t_unitdata *unitdata;
```

Description

The **t_sndudata** subroutine is used in connectionless mode to send a data unit to another transport user.

By default, the **t_sndudata** subroutine operates in synchronous mode and may wait if flow–control restrictions prevent the data from being accepted by the local transport provider at the time the call is made. However, if the **O_NDELAY** or **O_NONBLOCK** flag is set (using the **t_open** subroutine or the **fcntl** command), the **t_sndudata** subroutine runs in asynchronous mode and fails under such conditions.

Parameters

fd Identifies the local transport endpoint through which data is sent.

unitdata

Points to a **t_unitdata** structure containing the following elements:

```
struct netbuf addr;
struct netbuf opt;
struct netbuf udata;
```

The elements are defined as follows:

`addr` Specifies the protocol address of the destination user.

`opt` Identifies protocol–specific options that the user wants associated with this request.

`udata` Specifies the user data to be sent. The user can choose not to specify what protocol options are associated with the transfer by setting the `len` field of the `opt` field to 0. In this case, the provider can use default options.

If the `len` field of the `udata` field is 0, no data unit is passed to the transport provider; the **t_sndudata** subroutine does not send zero–length data units.

If the **t_sndudata** subroutine is issued from an invalid state, or if the amount of data specified in the `udata` field exceeds the TSDU size as returned by the **t_open** or **t_getinfo** subroutine, the provider generates an **EPROTO** protocol error.

Return Values

On successful completion, the **t_sndudata** subroutine returns a value of 0. Otherwise, it returns a value of –1 and sets the **t_errno** variable to indicate the error.

Error Codes

If unsuccessful, the **t_errno** variable is set to one of the following:

TBADDATA	The amount of user data specified was not within the bounds allowed by the transport provider.
TBADF	The specified file descriptor does not refer to a transport endpoint.
TFLOW	The **O_NDELAY** or **O_NONBLOCK** flag was set, but the flow–control mechanism prevented the transport provider from accepting data at this time.
TLOOK	An asynchronous event has occurred on this transport end point and requires immediate attention.
TNOTSUPPORT	This subroutine is not supported by the underlying transport provider.
TOUTSTATE	This subroutine was issued in the wrong sequence.
TSYSERR	A system error has occurred during execution of this subroutine.

Implementation Specifics

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

Related Information

The **t_alloc** subroutine, **t_open** subroutine, **t_rcvudata** subroutine, **t_rcvuderr** subroutine.

List of Streams Programming References and STREAMS Overview in *AIX Communications Programming Concepts*.

t_sync Subroutine for Transport Layer Interface

Purpose

Synchronizes transport library.

Library

Transport Layer Interface Library (**libtli.a**)

Syntax

```
#include <tiuser.h>

int t_sync(fd)
int fd;
```

Description

The **t_sync** subroutine synchronizes the data structures managed by the transport library with information from the underlying transport provider. In doing so, this subroutine can convert a raw file descriptor (obtained using the **open** or **dup** subroutine, or as a result of a **fork** operation and an **exec** operation) to an initialized transport endpoint, assuming that the file descriptor referenced a transport provider. This subroutine also allows two cooperating processes to synchronize their interaction with a transport provider.

For example, a process creates a new process with the **fork** subroutine and issues an **exec** subroutine call. The new process must issue a **t_sync** subroutine call to build the private library data structure associated with a transport endpoint and to synchronize the data structure with the relevant provider information.

> **Note:** The transport provider treats all users of a transport endpoint as a single user. If multiple processes are using the same endpoint, they should coordinate their activities so as not to violate the state of the provider. The **t_sync** subroutine returns the current state of the provider to the user, thereby enabling the user to verify the state before taking further action. This coordination is only valid among cooperating processes; a process or an incoming event may change the provider's state *after* a **t_sync** subroutine call is issued.

If the provider is undergoing a state transition when the **t_sync** subroutine is called, the subroutine will be unsuccessful.

Parameters

fd	Specifies the transport end point.

Return Values

On successful completion, the **t_sync** subroutine returns the state of the transport provider. Otherwise, it returns a value of –1 and sets the **t_errno** variable to indicate the error. The state returned can be one of the following:

T_UNBIND	Unbound
T_IDLE	Idle
T_OUTCON	Outgoing connection pending
T_INCON	Incoming connection pending
T_DATAXFER	Data transfer
T_OUTREL	Outgoing orderly release (waiting for an orderly release indication)
T_INREL	Incoming orderly release (waiting for an orderly release request)

Error Codes

If unsuccessful, the **t_errno** variable is set to one of the following:

TBADF	The specified file descriptor is a valid open file descriptor, but does not refer to a transport endpoint.
TSTATECHNG	The transport provider is undergoing a state change.
TSYSERR	A system error has occurred during execution of this function.

Implementation Specifics

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

Related Information

The **dup** subroutine, **exec** subroutine, **fork** subroutine, **open** subroutine.

List of Streams Programming References and STREAMS Overview in *AIX Communications Programming Concepts*.

t_unbind Subroutine for Transport Layer Interface

Purpose

Disables a transport endpoint.

Library

Transport Layer Interface Library (**libtli.a**)

Syntax

```
#include <tiuser.h>

int t_unbind(fd)
int fd;
```

Description

The **t_unbind** subroutine disables a transport endpoint, which was previously bound by the **t_bind** subroutine. On completion of this call, no further data or events destined for this transport endpoint are accepted by the transport provider.

Parameter

fd	Specifies the transport endpoint.

Return Values

On successful completion, the **t_unbind** subroutine returns a value of 0. Otherwise, it returns a value of –1 and sets the **t_errno** variable to indicate the error.

Error Codes

If unsuccessful, the **t_errno** variable is set to one of the following:

TBADF	The specified file descriptor does not refer to a transport endpoint.
TOUTSTATE	The function was issued in the wrong sequence.
TLOOK	An asynchronous event has occurred on this transport endpoint.
TSYSERR	A system error has occurred during execution of this function.

Implementation Specifics

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

Related Information

The **t_bind** subroutine.

List of Streams Programming References and STREAMS Overview in *AIX Communications Programming Concepts*.

testb Utility

Purpose

Checks for an available buffer.

Syntax

```
int
testb(size, pri)
register size;
uint pri;
```

Description

The **testb** utility checks for the availability of a message buffer of the size specified in the *size* parameter without actually retrieving the buffer. A successful return value from the **testb** utility does not guarantee that a subsequent call to the **allocb** utility will succeed; for example, when an interrupt routine takes the buffers.

Parameters

size Specifies the buffer size.

pri Specifies the relative importance of the allocated blocks to the module. The possible values are:

- **BPRI_LO**
- **BPRI_MED**
- **BPRI_HI**

The *pri* parameter is currently unused and is maintained only for compatibility with applications developed prior to UNIX System V Release 4.0.

Return Values

If the buffer is available, the **testb** utility returns a value of 1. Otherwise, it returns a value of 0.

Implementation Specifics

This utility is part of STREAMS Kernel Extensions.

Related Information

The **allocb** utility.

List of Streams Programming References and Understanding STREAMS Flow Control in *AIX Communications Programming Concepts*.

timeout Utility

Purpose

Schedules a function to be called after a specified interval.

Syntax

```
int
timeout(func, arg, ticks)
int (*func)();
caddr_t arg;
long ticks;
```

Description

The **timeout** utility schedules the function pointed to by the *func* parameter to be called with the *arg* parameter after the number of timer ticks specified by the *ticks* parameter. Multiple pending calls to the **timeout** utility with the same *func* and *arg* parameters are allowed. The function called by the **timeout** utility must adhere to the same restrictions as a driver interrupt handler. It must not sleep.

On multiprocessor systems, the function called by the **timeout** utility should be interrupt–safe. Otherwise, the **STR_QSAFETY** flag must be set when installing the module or driver with the **str_install** utility.

Note: This utility must not be confused with the kernel service of the same name in the **libsys.a** library. STREAMS modules and drivers inherently use this version, not the **libsys.a** library version. No special action is required to use this version in the STREAMS environment.

Parameters

func Indicates the function to be called. The function is declared as follows:

```
void (*func)(arg)
void *arg;
```

arg Indicates the parameter to supply to the function specified by the *func* parameter.

ticks Specifies the number of timer ticks that must occur before the function specified by the *func* parameter is called. Many timer ticks can occur every second.

Return Values

The **timeout** utility returns an integer that identifies the request. This value may be used to withdraw the time–out request by using the **untimeout** utility. If the timeout table is full, the **timeout** utility returns a value of –1 and the request is not registered.

Execution Environment

The **timeout** utility may be called from either the process or interrupt environment.

Implementation Specifics

This utility is part of STREAMS Kernel Extensions.

Related Information

The **untimeout** utility.

List of Streams Programming References in *AIX Communications Programming Concepts*.

Understanding STREAMS Drivers and Modules in *AIX Communications Programming Concepts*.

Understanding STREAMS Synchronization in *AIX Communications Programming Concepts*.

timod Module

Purpose

Converts a set of **streamio** operations into STREAMS messages.

Description

The **timod** module is a STREAMS module for use with the Transport Interface (TI) functions of the Network Services Library. The **timod** module converts a set of **streamio** operations into STREAMS messages that may be consumed by a transport protocol provider that supports the Transport Interface. This allows a user to initiate certain TI functions as atomic operations.

The **timod** module must only be pushed (see ”Pushable Modules” in *AIX Communications Programming Concepts*) onto a stream terminated by a transport protocol provider that supports the TI.

All STREAMS messages, with the exception of the message types generated from the **streamio** operations described below as values for the `cmd` field, will be transparently passed to the neighboring STREAMS module or driver. The messages generated from the following **streamio** operations are recognized and processed by the **timod** module.

Fields

The fields are described as follows:

`cmd`

Specifies the command to be carried out. The possible values for this field are:

TI_BIND Binds an address to the underlying transport protocol provider. The message issued to the **TI_BIND** operation is equivalent to the TI message type **T_BIND_REQ**, and the message returned by the successful completion of the operation is equivalent to the TI message type **T_BIND_ACK**.

TI_UNBIND Unbinds an address from the underlying transport protocol provider. The message issued to the **TI_UNBIND** operation is equivalent to the TI message type **T_UNBIND_REQ**, and the message returned by the successful completion of the operation is equivalent to the TI message type **T_OK_ACK**.

TI_GETINFO Gets the TI protocol–specific information from the transport protocol provider. The message issued to the **TI_GETINFO** operation is equivalent to the TI message type **T_INFO_REQ**, and the message returned by the successful completion of the operation is equivalent to the TI message type **T_INFO_ACK**.

TI_OPTMGMT Gets, sets, or negotiates protocol–specific options with the transport protocol provider. The message issued to the **TI_OPTMGMT** ioctl operation is equivalent to the TI message type **T_OPTMGMT_REQ**, and the message returned by the successful completion of the ioctl operation is equivalent to the TI message type **T_OPTMGMT_ACK**.

`len`	(On issuance) Specifies the size of the appropriate TI message to be sent to the transport provider. (On return) Specifies the size of the appropriate TI message from the transport provider in response to the issued TI message.
`dp`	Specifies a pointer to a buffer large enough to hold the contents of the appropriate TI messages. The TI message types are defined in the **sys/tihdr.h** file.

Examples

The following is an example of how to use the **timod** module:

```
#include <sys/stropts.h>
          _
          _
struct strioctl strioctl;
strucu t_info info;
          _
          _
strioctl.ic_cmd = TI_GETINFO;
strioctl.ic_timeout = INFTIM;
strioctl.ic_len = sizeof (info);
strioctl.ic_dp = (char *)&info;
ioctl(fildes, I_STR, &strioctl);
```

Implementation Specifics

This module is part of STREAMS Kernel Extensions.

Related Information

The **tirdwr** module.

The **streamio** operations.

Benefits and Features of STREAMS, Building STREAMS, Pushable Modules, Understanding STREAMS Drivers and Modules, Understanding STREAMS Messages, Using STREAMS in *AIX Communications Programming Concepts*.

tirdwr Module

Purpose

Supports the Transport Interface functions of the Network Services library.

Description

The **tirdwr** module is a STREAMS module that provides an alternate interface to a transport provider that supports the Transport Interface (TI) functions of the Network Services library. This alternate interface allows a user to communicate with the transport protocol provider by using the **read** and **write** subroutines. The **putmsg** and **getmsg** system calls can also be used. However, the **putmsg** and **getmsg** system calls can only transfer data messages between user and stream.

The **tirdwr** module must only be pushed (see the **I_PUSH** operation) onto a stream terminated by a transport protocol provider that supports the TI. After the **tirdwr** module has been pushed onto a stream, none of the TI functions can be used. Subsequent calls to TI functions will cause an error on the stream. Once the error is detected, subsequent system calls on the stream will return an error with the **errno** global variable set to EPROTO.

The following list describes actions taken by the **tirdwr** module when it is pushed or popped or when data passes through it:

push
Checks any existing data to ensure that only regular data messages are present. It ignores any messages on the stream that relate to process management. If any other messages are present, the **I_PUSH** operation returns an error and sets the **errno** global variable to **EPROTO**.

write
Takes the following actions on data that originated from a **write** subroutine:

Messages with no control portions
Passes the message on downstream.

Zero length data messages
Frees the message and does not pass downstream.

Messages with control portions
Generates an error, fails any further system calls, and sets the **errno** global variable to **EPROTO**.

read

Takes the following actions on data that originated from the transport protocol provider:

Messages with no control portions
Passes the message on upstream.

Zero length data messages
Frees the message and does not pass upstream.

Messages with control portions will produce the following actions**:**

- Messages that represent expedited data generate an error. All further calls associated with the stream fail with the **errno** global variable set to **EPROTO**.
- Any data messages with control portions have the control portions removed from the message prior to passing the message to the upstream neighbor.
- Messages that represent an orderly release indication from the transport provider generate a zero length data message, indicating the end of file, which is sent to the reader of the stream. The orderly release message itself is freed by the module.
- Messages that represent an abortive disconnect indication from the transport provider cause all further **write** and **putmsg** calls to fail with the **errno** global variable set to **ENXIO**. All further **read** and **getmsg** calls return zero length data (indicating end of file) once all previous data has been read.
- With the exception of the above rules, all other messages with control portions generate an error, and all further system calls associated with the stream fail with the **errno** global variable set to **EPROTO**.

pop Sends an orderly release request to the remote side of the transport connection if an orderly release indication has been previously received.

Implementation Specifics

This module is part of STREAMS Kernel Extensions.

Related Information

The **timod** module.

The **streamio** operations.

The **read** subroutine, **write** subroutine.

The **getmsg** system call, **putmsg** system call.

Benefits and Features of STREAMS, Building STREAMS, Pushable Modules, STREAMS Overview, Understanding STREAMS Drivers and Modules, Understanding STREAMS Messages, Using STREAMS in *AIX Communications Programming Concepts*.

unbufcall Utility

Purpose

Cancels a **bufcall** request.

Syntax

```
void unbufcall(id)
register int id;
```

Description

The **unbufcall** utility cancels a **bufcall** request.

Parameters

id	Identifies an event in the **bufcall** request.

Implementation Specifics

This utility is part of STREAMS Kernel Extensions.

Related Information

The **bufcall** utility.

List of Streams Programming References and Understanding STREAMS Messages in *AIX Communications Programming Concepts*.

unlinkb Utility

Purpose

Removes a message block from the head of a message.

Syntax

```
mblk_t *
unlinkb(bp)
register mblk_t *bp;
```

Description

The **unlinkb** utility removes the first message block pointed to by the *bp* parameter and returns a pointer to the head of the resulting message. The **unlinkb** utility returns a null pointer if there are no more message blocks in the message.

Parameters

bp	Specifies which message block to unlink.

Implementation Specifics

This utility is part of STREAMS Kernel Extensions.

Related Information

The **linkb** utility.

List of Streams Programming References and Understanding STREAMS Messages in *AIX Communications Programming Concepts*.

untimeout Utility

Purpose

Cancels a pending timeout request.

Syntax

```
int
untimeout(id)
int id;
```

Description

The **untimeout** utility cancels the specific request made with the **timeout** utility.

Note: This utility must not be confused with the kernel service of the same name in the **libsys.a** library. STREAMS modules and drivers inherently use this version, not the **libsys.a** library version. No special action is required to use this version in the STREAMS environment.

Parameters

id	Specifies the identifier returned from the corresponding timeout request.

Execution Environment

The **untimeout** utility can be called from either the process or interrupt environment.

Implementation Specifics

This utility is part of STREAMS Kernel Extensions.

Related Information

The **timeout** utility.

List of Streams Programming References and Understanding STREAMS Drivers and Modules in *AIX Communications Programming Concepts*.

unweldq Utility

Purpose

Removes a previously established weld connection between STREAMS queues.

Syntax

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

int unweldq (q1, q2, q3, q4, func, arg, protect_q)
queue_t *q1;
queue_t *q2;
queue_t *q3;
queue_t *q4;
weld_fcn_t func;
weld_arg_t arg;
queue_t *protect_q;
```

Description

The **unweldq** utility removes a weld connection previously established with the **weld** utility between two STREAMS queues (*q1* and *q2*). The **unweldq** utility can be used to unweld two pairs of queues in one call (*q1* and *q2*, *q3* and *q4*).

The unwelding operation is performed by changing the first queue's **q_next** pointer so that it does not point to any queue. The **unweldq** utility does not actually perform the operation. Instead, it creates an unwelding request which STREAMS performs asynchronously. STREAMS acquires the appropriate synchronization queues before performing the operation.

Callers that need to know when the unwelding operation has actually taken place should specify a callback function (*func* parameter) when calling the **unweldq** utility. If the caller also specifies a synchronization queue (*protect_q* parameter) , STREAMS acquires the synchronization associated with that queue when calling *func*. If the callback function is not a protected STREAMS utility, such as the **qenable** utility, the caller should always specify a *protect_q* parameter. The caller can also use this parameter to synchronize callbacks with protected STREAMS utilities.

Note: The **stream.h** header file must be the last included header file of each source file using the stream library.

Parameters

q1	Specifies the queue whose **q_next** pointer must be nulled.
q2	Specifies the queue that will be unwelded to *q1*.
q3	Specifies the second queue whose **q_next** pointer must be nulled. If the **unweldq** utility is used to unweld only one pair of queues, this parameter should be set to **NULL**.
q4	Specifies the queue that will be unwelded to *q3*.
func	Specifies an optional callback function that will execute when the unwelding operation has completed.
arg	Specifies the parameter for *func*.
protect_q	Specifies an optional synchronization queue that protects *func*.

Return Values

Upon successful completion, **0** (zero) is returned. Otherwise, an error code is returned.

Error Codes

The **unweldq** utility fails if the following is true:

EAGAIN	The weld record could not be allocated. The caller may try again.
EINVAL	One or more parameters are not valid.
ENXIO	The weld mechanism is not installed.

Implementation Specifics

This utility is part of STREAMS Kernel Extensions.

Related Information

List of Streams Programming References in *AIX Communications Programming Concepts*.

STREAMS Overview in *AIX Communications Programming Concepts*.

Welding Mechanism in *AIX Communications Programming Concepts*.

The **weldq** utility.

wantio Utility

Purpose

Register direct I/O entry points with the stream head.

Syntax

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

int wantio(queue_t *q, struct wantio *w)
```

Parameters

q	Pointer to the **queue** structure.
w	Pointer to the **wantio** structure.

Description

The **wantio** STREAMS routine can be used by a STREAMS module or driver to register input/output (read/write/select) entry points with the stream head. The stream head then calls these entry points directly, by–passing all normal STREAMS processing, when an I/O request is detected. This service may be useful to increase STREAMS performance in cases where normal module processing is not required or where STREAMS processing is to be performed outside of AIX.

STREAMS modules and drivers should precede a **wantio** call by sending a high priority M_LETSPLAY message upstream. The M_LETSPLAY message format is a message block containing an integer followed by a pointer to the write queue of the module or driver originating the M_LETSPLAY message. The integer counts the number of modules that can permit direct I/O. Each module passes this message to its neighbor after incrementing the count if direct I/O is possible. When this message reaches the stream head, the stream head compares the count field with the number of modules and drivers in the stream. If the count is not equal to the number of modules, then a M_DONTPLAY message is sent downstream indicating direct I/O will not be permitted on the stream. If the count is equal, then queued messages are cleared by sending them downstream as M_BACKWASH messages. When all messages are cleared, then an M_BACKDONE message is sent downstream. This process starts at the stream head and is repeated in every module in the stream. Modules will wait to receive an M_BACKDONE message from upstream. Upon receipt of this message, the module will send all queued data downstream as M_BACKWASH messages. When all data is cleared, the module will send an M_BACKDONE message to its downstream neighbor indicating that all data has been cleared from the stream to this point. wantio registration is cleared from a stream by issuing a **wantio** call with a NULL pointer to the wantio structure.

Multiprocessor serialization is the responsibility of the driver or module requesting direct I/O. The stream head acquires no STREAMS locks before calling the wantio entry point.

Currently, the write entry point of the **wantio** structure is ignored.

Return Values

Returns 0 always.

Related Information

The **wantmsg** utility.

The **queue** and **wantio** structures in **/usr/include/sys/stream.h**.

The STREAMS Entry Points article in InfoExplorer.

wantmsg Utility

Purpose

Allows a STREAMS message to bypass a STREAMS module if the module is not interested in the message.

Syntax

```
int wantmsg(q, f)
queue_t * q;
int (*f)();
```

Description

The **wantmsg** utility allows a STREAMS message to bypass a STREAMS module if the module is not interested in the message, resulting in performance improvements.

The module registers filter functions with the read and write queues of the module with the **wantmsg** utility. A filter function takes as input a message pointer and returns 1 if the respective queue is interested in receiving the message. Otherwise it returns 0. The **putnext** and **qreply** subroutines call a queue's filter function before putting a message on that queue. If the filter function returns 1, then **putnext** or **qreply** put the message on that queue. Otherwise, **putnext** or **qreply** bypass the module by putting the message on the next module's queue.

The filter functions must be defined so that a message bypasses a module only when the module does not need to see the message.

The **wantmsg** utility cannot be used if the module has a service routine associated with the queue specified by the *q* parameter. If **wantmsg** is called for a module that has a service routine associated with *q*, **wantmsg** returns a value of 0 without registering the filter function with *q*.

Parameters

q	Specifies the read or write queue to which the filter function is to be registered.
f	Specifies the module's filter function that is called at the **putnext** or **qreply** time.

Return Values

Upon successful completion, the **wantmsg** utility returns a 1, indicating that the filter function specified by the *f* parameter has been registered for the queue specified by the *q* parameter. In this case, the filter function is called from **putnext** or **qreply**. The **wantmsg** utility returns a value of 0 if the module has a service routine associated with the queue *q*, indicating that the filter function is not registered with *q*.

Example

```
wantmsg(q, tioc_is_r_interesting);
        wantmsg(WR(q), tioc_is_w_interesting);

/*
 * read queue filter function.
 * queue is only interested in IOCNAK, IOCACK, and
 * CTL messages.
 */

static int
tioc_is_r_interesting(mblk_t *mp)
{
        if (mp->b_datap->db_type == M_DATA)
                /* fast path for data messages */
                return 0;
        else if (mp->b_datap->db_type == M_IOCNAK ||
                 mp->b_datap->db_type == M_IOCACK ||
                 mp->b_datap->db_type == M_CTL)
                return 1;
        else
                return 0;
}

/*
 * write queue filter function.
 * queue is only interested in IOCTL and IOCDATA
 * messages.
 */

static int
tioc_is_w_interesting(mblk_t *mp)
{
        if (mp->b_datap->db_type == M_DATA)
                /* fast path for data messages */
                return 0;
        else if (mp->b_datap->db_type == M_IOCTL ||
                 mp->b_datap->db_type == M_IOCDATA)
                return 1;
        else
                return 0;
}
```

Implementation Specifics

This utility is part of STREAMS Kernel Extensions.

Related Information

The **putnext** utility, the **qreply** utility.

List of Streams Programming References, STREAMS Messages in *AIX Communications Programming Concepts*.

weldq Utility

Purpose

Establishes an uni–directional connection between STREAMS queues.

Syntax

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

int weldq (q1, q2, q3, q4, func, arg, protect_q)
queue_t *q1;
queue_t *q2;
queue_t *q3;
queue_t *q4;
weld_fcn_t func;
weld_arg_t arg;
queue_t *protect_q;
```

Description

The **weldq** utility establishes an uni–directionnal connection (weld connection) between two STREAMS queues (*q1* and *q2*). The **weldq** utility can be used to weld two pairs of queues in one call (*q1* and *q2*, *q3* and *q4*).

The welding operation is performed by changing the first queue's **q_next** pointer to point to the second queue. The **weldq** utility does not actually perform the operation. Instead, it creates a welding request which STREAMS performs asynchronously. STREAMS acquires the appropriate synchronization queues before performing the operation.

Callers that need to know when the welding operation has actually taken place should specify a callback function (*func* parameter) when calling the **weldq** utility. If the caller also specifies a synchronization queue (*protect_q* parameter), STREAMS acquires the synchronization associated with that queue when calling *func*. If the callback function is not a protected STREAMS utility, such as the **qenable** utility, the caller should always specify a *protect_q* parameter. The caller can also use this parameter to synchronize callbacks with protected STREAMS utilities.

Note: The **stream.h** header file must be the last included header file of each source file using the stream library.

Parameters

q1	Specifies the queue whose **q_next** pointer must be modified.
q2	Specifies the queue that will be welded to *q1*.
q3	Specifies the second queue whose **q_next** pointer must be modified. If the **weldq** utility is used to weld only one pair of queues, this parameter should be set to **NULL**.
q4	Specifies the queue that will be welded to *q3*.
func	Specifies an optional callback function that will execute when the welding operation has completed.
arg	Specifies the parameter for *func*.
protect_q	Specifies an optional synchronization queue that protects *func*.

Return Values

Upon successful completion, **0** (zero) is returned. Otherwise, an error code is returned.

Error Codes

The **weldq** utility fails if the following is true:

EAGAIN	The weld record could not be allocated. The caller may try again.
EINVAL	One or more parameters are not valid.
ENXIO	The weld mechanism is not installed.

Implementation Specifics

This utility is part of STREAMS Kernel Extensions.

Related Information

List of Streams Programming References in *AIX Communications Programming Concepts*.

STREAMS Overview in *AIX Communications Programming Concepts*.

Welding Mechanism in *AIX Communications Programming Concepts*.

The **unweldq** utility.

WR Utility

Purpose

Retrieves a pointer to the write queue.

Syntax

```
#define WR(q) ((q)+1)
```

Description

The **WR** utility accepts a read queue pointer, the *q* parameter, as an argument and returns a pointer to the write queue for the same module.

Parameters

q	Specifies the read queue.

Implementation Specifics

This utility is part of STREAMS Kernel Extensions.

Related Information

The **OTHERQ** utility, **RD** utility.

List of Streams Programming References and Understanding STREAMS Messages in *AIX Communications Programming Concepts*.

xtiso STREAMS Driver

Purpose

Provides access to sockets–based protocols to STREAMS applications.

Description

The **xtiso** driver (XTI over SOckets) is a STREAMS–based pseudo–driver that provides a Transport Layer Interface (TLI) to the socket–based protocols. The only supported use of the **xtiso** driver is by the TLI and XTI libraries.

The TLI and XTI specifications do not describe the name of the transport provider and how to address local and remote hosts, two important items required for use.

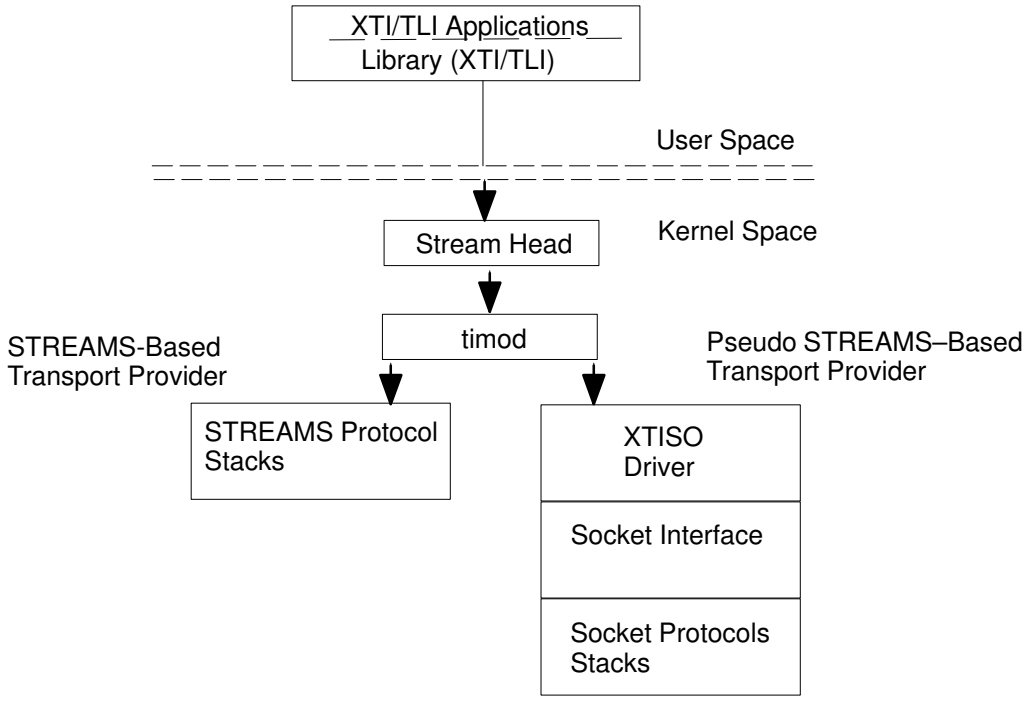
The **xtiso** driver supports most of the protocols available through the socket interface. Each protocol has a /**dev** entry, which must be used as the *name* parameter in the **t_open** subroutine. The currently supported names (as configured by the **strload** subroutine) are:

Name	Socket Equivalent
/dev/xti/unixdg	AF_UNIX, SOCK_DGRAM
/dev/xti/unixst	AF_UNIX, SOCK_STREAM
/dev/xti/udp	AF_INET, SOCK_DGRAM
/dev/xti/tcp	AF_INET, SOCK_STREAM

Each of these protocols has a **sockaddr** structure that is used to specify addresses. These structures are also used by the TLI and XTI functions that require host addresses. The **netbuf** structure associated with the address for a particular function should refer to one of the **sockaddr** structure types. For instance, the TCP socket protocol uses a **sockaddr_in** structure; so a corresponding **netbuf** structure would be:

```
struct netbuf addr;
struct sockaddr_in sin;
/* initialize sockaddr here */
sin.sin_family = AF_INET;
sin.sin_port = 0;
sin.sin_addr.s_addr = inet_addr(”127.0.0.1”);
addr.maxlen = sizeof(sin);
addr.len = sizeof(sin);
addr.buf = (char *)&sin;
```

The following figure shows the implementation of XTI/TLI with STREAMS.



The X/Open Transport Interface (XTI) Stream always consists of a Stream head and the transport interface module, **timod**. Depending on the transport provider specified by the application, **timod** accesses either the STREAMS–based protocol stack natively or a socket–based protocol through the pseudo–driver, **xtiso**.

The XTI library, **libxti.a** assumes a STREAMS–based transport provider. The routines of this library perform various operations for sending transport Provider Interface, TPI, messages down the XTI streams to the transport provider and receives them back.

The transport interface module, **timod**, is a STREAMS module that completes the translation of the TPI messages in the downstream and upstream directions.

The **xtiso** driver is a pseudo–driver that acts as the transport provider for socket–based communications. It interprets back and forth between the the TPI messages it receives from upstream and the socket interface.

AIX also provides the transport interface read/write module, **tirdwr**, which applications can push on to the XTI/TLI Stream for accessing the socket layer with standard UNIX read and write calls.

Files

/dev/xti/*	Contains names of supported protocols.

Implementation Specifics

This driver is part of STREAMS Kernel Extensions.

Related Information

The **strload** command.

The **t_bind** subroutine for Transport Layer Interface, **t_connect** subroutine for Transport Layer Interface, **t_open** subroutine for Transport Layer Interface.

The **t_bind** subroutine for X/Open Transport Layer Interface, **t_connect** subroutine for X/Open Transport Layer Interface, **t_open** subroutine for X/Open Transport Layer Interface.

Internet Transport–Level Protocols in *AIX 4.3 System Management Guide: Communications and Networks*.

UNIX System V Release 4 Programmer's Guide: Networking Interfaces.

Understanding STREAMS Drivers and Modules in *AIX Communications Programming Concepts*.

t_accept Subroutine for X/Open Transport Interface

Purpose

Accept a connect request.

Library

X/Open Transport Interface Library (**libxti.a**)

Syntax

```
#include <xti.h>

int t_accept (fd, resfd, call)
int fd;
int resfd;
const struct t_call *call;
```

Description

The **t_accept** subroutine is issued by a transport user to accept a command request. A transport user may accept a connection on either the same local transport endpoint or on an endpoint different than the one on which the connect indication arrived.

Before the connection can be accepted on the same endpoint, the user must have responded to any previous connect indications received on that transport endpoint via the **t_accept** subroutine or the **t_snddis** subroutine. Otherwise, the **t_accept** subroutine will fail and set **t_errno** to **TINDOUT**.

If a different transport endpoint is specified, the user may or may not choose to bind the endpoint before the **t_accept** subroutine is issued. If the endpoint is not bound prior to the **t_accept** subroutine, the transport provider will automatically bind the endpoint to the same protocol address specified in the *fd* parameter. If the transport user chooses to bind the endpoint, it must be bound to a protocol address with a *qlen* field of zero (see the **t_bind** subroutine) and must be in the **T_IDLE** state before the **t_accept** subroutine is issued.

The call to the **t_accept** subroutine fails with **t_errno** set to **TLOOK** if there are indications (for example, connect or disconnect) waiting to be received on the endpoint specified by the *fd* parameter.

The value specfied in the *udata* field enables the called transport user to send user data to the caller. The amount of user data sent must not exceed the limits supported by the transport provider. This limit is specified in the *connect* field of the **t_info** structure of the **t_open** or **t_getinfo** subroutines. If the *len* field of *udata* is zero, no data is sent to the caller. All the *maxlen* fields are meaningless.

When the user does not indicate any option, it is assumed that the connection is to be accepted unconditionally. The transport provider may choose options other than the defaults to ensure that the connection is accepted successfully.

Parameters

fd Identifies the local transport endpoint where the connect indication arrived.

resfd Specifies the local transport endpoint where the connection is to be established.

call

Contains information required by the transport provider to complete the connection. The *call* parameter points to a **t_call** structure which contains the following members:

```
struct netbuf addr;
struct netbuf opt;
struct netbuf udata;
int sequence;
```

The fields within the structure have the following meanings:

addr Specifies the protocol address of the calling transport user. The address of the caller may be null (length zero). When this field is not null, the field may be optionally checked by the X/Open Transport Interface.

opt Indicates any options associated with the connection.

udata Points to any user data to be returned to the caller.

sequence Specifies the value returned by the **t_listen** subroutine which uniquely associates the response with a previously received connect indication.

Valid States

fd: T_INCON
resfd (*Fd != resfd***): T_IDLE**

Return Values

0 Successful completion.

–1 Unsuccessful completion, **t_errno** is set to indicate an error.

Error Codes

On failure, **t_errno** is set to one of the following:

TACCES The user does not have permission to accept a connection on the responding transport endpoint or to use the specified options.

TBADADDR The specified protocol address was in an incorrect format or contained illegal information.

TBADDATA The amount of user data specified was not within the bounds allowed by the transport provider.

TBADF The file descriptor *fd* or *resfd* does not refer to a transport endpoint.

TBADOPT The specified options were in an incorrect format or contained illegal information.

TBADSEQ An invalid sequence number was specified.

TINDOUT	The subroutine was called with the same endpoint, but there are outstanding connection indications on the endpoint. Those other connection indications must be handled either by rejecting them via the **t_snddis** subroutine or accepting them on a different endpoint via the **t_accept** subroutine.
TLOOK	An asynchronous event has occurred on the transport endpoint referenced by *fd* and requires immediate attention.
TNOTSUPPORT	This subroutine is not supported by the underlying transport provider.
TOUTSTATE	The subroutine was called in the wrong sequence on the transport endpoint referenced by *fd,* or the transport endpoint referred to by *resfd* is not in the appropriate state.
TPROTO	This error indicates that a communication problem has been detected between X/Open Transport Interface and the transport provider for which there is no other suitable X/Open Transport Interface(**t_errno**).
TPROVMISMATCH	The file descriptors *fd* and *resfd* do not refer to the same transport provider.
TRESADDR	This transport provider requires both *fd* and *resfd* to be bound to the same address. This error results if they are not.
TRESQLEN	The endpoint referenced by *resfd* (where *resfd* is a different transport endpoint) was bound to a protocol address with a *qlen* field value that is greater than zero.
TSYSERR	A system error has occurred during execution of this subroutine.

Implementation Specifics

There may be transport provider–specific restrictions on address binding. See Appendix A, ISO Transport Protocol Information and Appendix B, Internet Protocol–specific Information.

Some transport providers do not differentiate between a connect indication and the connection itself. If the connection has already been established after a successful return of the **t_listen** subroutine, the **t_accept** subroutine will assign the existing connection to the transport endpoint specified by *resfd* (see Appendix B, Internet Protocol–specific Information).

Related Information

The **t_connect** subroutine, **t_getstate** subroutine, **t_open** subroutine, **t_optmgmt** subroutine, **t_rcvconnect** subroutine.

t_alloc Subroutine for X/Open Transport Interface

Purpose

Allocate a library structure.

Library

X/Open Transport Interface Library (**libxti.a**)

Syntax

```
#include <xti.h>

void *t_alloc (
int fd
int struct_type,
int fields)
```

Description

The **t_alloc** subroutine dynamically allocates memory for the various transport function parameter structures. This subroutine allocates memory for the specified structure, and also allocates memory for buffers referenced by the structure.

Use of the **t_alloc** subroutine to allocate structures helps ensure the compatibility of user programs with future releases of the transport interface functions.

Parameters

fd Specifies the transport endpoint through which the newly allocated structure will be passed.

struc_type

Specifies the structure to be allocated. The possible values are:

T_BIND	struct t_bind
T_CALL	struct t_call
T_OPTMGMT	struct t_optmgmt
T_DIS	struct t_discon
T_UNITDATA	struct t_unitdata
T_UDERROR	struct t_uderr
T_INFO	struct t_info

Each of these structures may subsequently be used as a parameter to one or more transport functions. Each of the above structures, except **T_INFO**, contains at least one field of the **struct netbuf** type. For each field of this type, the user may specify that the buffer for that field should be allocated as well. The length of the buffer allocated will be equal to or greater than the appropriate size as returned in the *info* paramenter of the **t_open** or **t_getinfo** subroutines.

fields

Specfies whether the buffer should be allocated for each field type. The *fields* parameter specifies which buffers to allocate, where the parameter is the bitwise–OR of any of the following:

T_ADDR The *addr* field of the **t_bind**, **t_call**, **t_unitdata** or **t_underr** structures.

T_OPT The *opt* field of the t_**optmgmt**, **t_call**, **t_unitdata** or **t_underr** structures.

T_UDATA The *udata* field of the **t_call**, **t_discon** or **t_unitdata** structures.

T_ALL All relevant fields of the given structure. Fields which are not supported by the transport provider specified by the *fd* parameter are not allocated.

For each relevant field specified in the *fields* parameter, the **t_alloc** subroutine allocates memory for the buffer associated with the field and initializes the *len* field to zero and initializes the *buf* pointer and *maxlen* field accordingly. Irrelevant or unknown values passed in fields are ignored. The length of the buffer allocated is based on the same size information returned to the user on a call to the **t_open** and **t_getinfo** subroutines. Thus, the *fd* paramenter must refer to the transport endpoint through which the newly allocated structure is passed so that the appropriate size information is accessed. If the size value associated with any specified field is –1 or –2, (see the **t_open** or **t_getinfo** subroutines), the **t_alloc** subroutine is unable to determine the size of the buffer to allocate and fails, setting **t_errno** to **TSYSERR** and *errno* to **EINVAL**. For any field not specified in *fields, buf* will be set to the null pointer and *len* and *maxlen* will be set to zero.

Valid States

ALL – apart from T_UNINIT.

Return Values

On successful completion, the **t_alloc** subroutinereturns a pointer to the newly allocated structure. On failure, a null pointer is returned.

Error Codes

On failure, **t_errno** is set to one of the following:

TBADF The specified file descriptor does not refer to a transport endpoint.

TSYSERR A system error has occurred during execution of this function.

TNOSTRUCTYPE Unsupported structure type (*struct_type*) requested. This can include a request for a structure type which is inconsistent with the transport provider type specified, for example, connection–oriented or connectionless.

TPROTO This error indicates that a communication problem has been detected between the X/Open Transport Interface and the transport provider for which there is no other suitable X/Open Transport Interface (**t_errno**).

Related Information

The **t_free** subroutine, **t_getinfo** subroutine, **t_open** subroutine.

t_bind Subroutine for X/Open Transport Interface

Purpose

Bind an address to a transport endpoint.

Library

X/Open Transport Interface Library (**libxti.a**)

Syntax

```
#include <xti.h>

int t_bind (fd, req, ret)
int fd;
const struct t_bind *req;
struct t_bind *ret;
```

Description

The **t_bind** subroutine associates a protocol address with the transport endpoint specified by the *fd* parameter and activates that transport endpoint. In connection mode, the transport provider may begin enqueuing incoming connect indications or servicing a connection request on the transport endpoint. In connectionless mode, the transport user may send or receive data units through the transport endpoint.

The *req* and *ret* parameters point to a **t_bind** structure containing the following members:

```
struct netbuf addr;
unsigned qlen;
```

Within this structure, the fields have the following meaning:

addr	Specifies a protocol address.
qlen	Indicates the maximum number of outstanding connect indications.

If the requested address is not available, the **t_bind** subroutine returns **–1** with **t_errno** set as appropriate. If no address is specified in the *req* parameter, (that is, the *len* field of the *addr* field in the *req* parameter is zero or the *req* parameter is NULL), the transport provider assigns an appropriate address to be bound, and returns that address in the *addr* field of the *ret* parameter. If the transport provider could not allocate an address, the **t_bind** subroutine fails with **t_errno** set to **TNOADDR**.

The *qlen* field has meaning only when initializing a connection–mode service. This field specifies the number of outstanding connect indications that the transport provider should support for the given transport endpoint. An outstanding connect indication is one that has been passed to the transport user by the transport provider but which has not been accepted or rejected. A *qlen* field value of greater than zero is only meaningful when issued by a passive transport user that expects other users to call it. The value of the *qlen* field is negotiated by the transport provider and may be changed if the transport provider cannot support the specified number of outstanding connect indications. However, this value of the *qlen* field is never negotiated from a requested value greater than zero to zero. This is a requirement on transport providers. See ”Implementation Specifics” for more information. On return, the *qlen* field in the *ret* parameter contains the negotiated value.

Parameters

fd Specifies the transport endpoint. If the *fd* parameter refers to a connection–mode service, this function allows more than one transport endpoint to be bound to the same protocol address. However, the transport provider must also support this capability and it is not possible to bind more than one protocol address to the same transport endpoint. If a user binds more than one transport endpoint to the same protocol address, only one endpoint can be used to listen for connect indications associated with that protocol address. In other words, only one **t_bind** for a given protocol address may specify a *qlen* field value greater than zero. In this way, the transport provider can identify which transport endpoint should be notified of an incoming connect indication. If a user attempts to bind a protocol address to a second transport endpoint with a a *qlen* field value greater than zero, **t_bind** will return **–1** and set **t_errno** to **TADDRBUSY**. When a user accepts a connection on the transport endpoint that is being used as the listening endpoint, the bound protocol address will be found to be busy for the duration of the connection, until a **t_unbind** or **t_close** call has been issued. No other transport endpoints may be bound for listening on that same protocol address while that initial listening endpoint is active (in the data transfer phase or in the **T_IDLE** state). This will prevent more than one transport endpoint bound to the same protocol address from accepting connect indications.

If the *fd* parameter refers to a connectionless–mode service, only one endpoint may be associated with a protocol address. If a user attempts to bind a second transport endpoint to an already bound protocol address, **t_bind** will return –1 and set **t_errno** to **TADDRBUSY**.

req

Specifies the address to be bound to the given transport endpoint. The *req* parameter is used to request that an address, represented by the **netbuf** structure, be bound to the given transport endpoint. The **netbuf** structure is described in the **xti.h** file. In the *req* parameter, the **netbuf** structure *addr* fields have the following meanings:

buf	Points to the address buffer.
len	Specifies the number of bytes in the address.
maxlen	Has no meaning for the *req* parameter.

The *req* parameter may be a null pointer if the user does not specify an address to be bound. Here, the value of the *qlen* field is assumed to be zero, and the transport provider assigns an address to the transport endpoint. Similarly, the *ret* parameter may be a null pointer if the user does not care what address was bound by the provider and is not interested in the negotiated value of the *qlen* field. It is valid to set the *req* and *ret* parameters to the null pointer for the same call, in which case the provider chooses the address to bind to the transport endpoint and does not return that information to the user.

ret

Specifies the maximum size of the address buffer. On return, the *ret* parameter contains the address that the transport provider actually bound to the transport endpoint; this is the same as the address specified by the user in the *req* parameter. In the *ret* parameter, the **netbuf** structure fields have the following meanings:

buf	Points to the buffer where the address is to be placed. On return, this points to the bound address.
len	Specifies the number of bytes in the bound address on return.
maxlen	Specifies the the maximum size of the address buffer. If the value of the *maxlen* field is not large enough to hold the returned address, an error will result.

Valid States

T_UNBIND.

Return Values

0	Successful completion.
–1	**t_errno** is set to indicate an error.

Error Codes

On failure, **t_errno** is set to one of the following:

TACCES	The user does not have permission to use the specified address.
TADDRBUSY	The requested address is in use.
TBADADDR	The specified protocol address was in an incorrect format or contained illegal information.
TBADF	The specified file descriptor does not refer to a transport endpoint.
TBUFOVLW	The number of bytes allowed for an incoming argument (*maxlen*) is greater than 0 but not sufficient to store the value of that argument. The provider's state will change to **T_IDLE** and the information to be returned in *ret* will be discarded.
TNOADDR	The transport provider could not allocate an address.
TOUTSTATE	The function was issued in the wrong sequence.
TPROTO	This error indicates that a communication problem has been detected between the X/Open Transport Interface and the transport provider for which there is no other suitable X/Open Transport Interface (**t_errno**).
TSYSERR	A system error has occurred during execution of this function.

Implementation Specifics

The requirement that the value of the *qlen* field never be negotiated from a requested value greater than zero to zero implies that transport providers, rather than the X/Open Transport Interface implementation itself, accept this restriction.

A transport provider may not allow an explicit binding of more than one transport endpoint to the same protocol address, although it allows more than one connection to be accepted for the same protocol address. To ensure portability, it is, therefore, recommended not to bind transport endpoints that are used as responding endpoints, (those specified in the *resfd* parameter), in a call to the **t_accept** subroutine, if the responding address is to be the same as the called address.

Related Information

The **t_alloc** subroutine, **t_close** subroutine, **t_open** subroutine, **t_optmgmt** subroutine, **t_unbind** subroutine.

t_close Subroutine for X/Open Transport Interface

Purpose

Close a transport endpoint.

Library

X/Open Transport Interface Library (**libxti.a**)

Syntax

```
#include <xti.h>
int t_close (fd)
int fd;
```

Description

The **t_close** subroutine informs the transport provider that the user is finished with the transport endpoint specified by the *fd* parameter and frees any local library resources associated with the endpoint. In addition, the **t_close** subroutine closes the file associated with the transport endpoint.

The **t_close** subroutine should be called from the **T_UNBND** state (see the **t_getstate** subroutine). However, this subroutine does not check state information, so it may be called from any state to close a transport endpoint. If this occurs, the local library resources associated with the endpoint will be freed automatically. In addition, the **close** subroutine is issued for that file descriptor. The **close** subroutine is abortive if there are no other descriptors in this process or if there are no other descriptors in another process which references the transport endpoint, and in this case, will break any transport connection that may be associated with that endpoint.

A **t_close** subroutine issued on a connection endpoint may cause data previously sent, or data not yet received, to be lost. It is the responsibility of the transport user to ensure that data is received by the remote peer.

Parameter

fd	Specfies the transport endpoint to be closed.

Valid States

ALL – apart from T_UNINIT.

Return Values

0	Successful completion.
–1	**t_errno** is set to indicate an error.

Errors

On failure, **t_errno** is set to one of the following:

TBADF	The specified file descriptor does not refer to a transport endpoint.
TPROTO	This error indicates that a communication problem has been detected between the X/Open Transport Interface and the transport provider for which there is no other suitable X/Open Transport Interface (**t_errno**).

Related Information

The **t_getstate** subroutine, **t_open** subroutine, **t_unbind** subroutine.

t_connect Subroutine for X/Open Transport Interface

Purpose

Establish a connection with another transport user.

Library

X/Open Transport Interface Library (**libxti.a**)

Syntax

```
#include <xti.h>

int t_connect (fd, sndcall, rcvcall)
int fd;
const struct t_call *sndcall;
struct t_call *rcvcall;
```

Description

The **t_connect** subroutine enables a transport user to request a connection to the specified destination transport user. This subroutine can only be issued in the **T_IDLE** state.

The *sndcall* and *rcvcall* parameters both point to a **t_call** structure which contains the following members:

```
struct netbuf addr;
struct netbuf opt;
struct netbuf udata;
int sequence;
```

In the *sndcall* parameter, the fields of the structure have the following meanings:

addr	Specifies the protocol address of the destination transport user.
opt	Presents any protocol–specific information that might be needed by the transport provider.
sequence	Has no meaning for this subroutine.
udata	Points to optional user data that may be passed to the destination transport user during connection establishment.

On return, the fields of the structure pointed to by the *rcvcall* parameter have the following meanings:

addr	Specifies the protocol address associated with the responding transport endpoint.
opt	Represents any protocol–specific information associated with the connection.
sequence	Has no meaning for this subroutine.
udata	Points to optional user data that may be returned by the destination transport user during connection establishment.

The *opt* field permits users to define the options that may be passed to the transport provider. These options are specific to the underlying protocol of the transport provider and are described for ISO and TCP protocols in Appendix A, ISO Transport Protocol Information, Appendix B, Internet Protocol–specific Information and Appendix F, Headers and Definitions. The user may choose not to negotiate protocol options by setting the *len* field of *opt* to zero. In this case, the provider may use default options.

If used, the value of the *opt.buf* field of the *sndcall* parameter **netbuf** structure must point to a buffer with the corresponding options; the *maxlen* and *buf* values of the *addr* and *opt* fields of the *rcvcall* parameter **netbuf** structure must be set before the call.

The *udata* field of the structure enables the caller to pass user data to the destination transport user and receive user data from the destination user during connection establishment. However, the amount of user data must not exceed the limits supported by the transport provider as returned in the *connect* field of the *info* parameter of the **t_open** or **t_getinfo** subroutines. If the value of *udata.len* field is zero in the *sndcall* parameter **netbuf** structure, no data will be sent to the destination transport user.

On return, the *addr, opt*, and *udata* fields of *rcvcall* are updated to reflect values associated with the connection. Thus, the *maxlen* value of each field must be set before issuing this subroutine to indicate the maximum size of the buffer for each. However, the value of the *rcvcall* parameter may be a null pointer, in which case no information is given to the user on return from the **t_connect** subroutine.

By default, the **t_connect** subroutine executes in synchronous mode, and waits for the destination user's response before returning control to the local user. A successful return (for example, return value of zero) indicates that the requested connection has been established. However, if **O_NONBLOCK** is set via the **t_open** subroutine or the *fcntl* parameter, the **t_connect** subroutine executes in asynchronous mode. In this case, the call will not wait for the remote user's response, but returns control immediately to the local user and returns –1 with **t_errno** set to **TNODATA** to indicate that the connection has not yet been established. In this way, the subroutine initiates the connection establishment procedure by sending a connect request to the destination transport user. The **t_rcvconnect** subroutine is used in conjunction with the **t_connect** subroutine to determine the status of the requested connection.

When a synchronous **t_connect** call is interrupted by the arrival of a signal, the state of the corresponding transport endpoint is **T_OUTCON**, allowing a further call to either the **t_rcvconnect**, **t_rcvdis** or **t_snddis** subroutines.

Parameters

fd	Identifies the local transport endpoint where communication will be established.
sndcall	Specifies information needed by the transport provider to establish a connection.
rcvcall	Specifies information associated with the newly establisehd connection.

Valid States

T_IDLE.

Return Values

0	Successful completion.
–1	**t_errno** is set to indicate an error.

Error Codes

On failure, **t_errno** is set to one of the following:

TACCES	The user does not have permission to use the specified address or options.
TADDRBUSY	This transport provider does not support multiple connections with the same local and remote addresses. This error indicates that a connection already exists.

TBADADDR The specified protocol address was in an incorrect format or contained illegal information.

TBADDATA The amount of user data specified was not within the bounds allowed by the transport provider.

TBADF The specified file descriptor does not refer to a transport endpoint.

TBADOPT The specified protocol options were in an incorrect format or contained illegal information.

TBUFOVFLW The number of bytes allocated for an incoming parameter (*maxlen*) is greater than 0 but not sufficient to store the value of that argument. If executed in synchronous mode, the provider's state, as seen by the user, changes to **T_DAXAXFER**, and the information to be returned in the *rcvcall* parameter is discarded.

TLOOK An asynchronous event has occurred on this transport endpoint and requires immediate attention.,

TNODATA **O_NONBLOCK** was set, so the subroutine successfully initiated the connection establishment procedure, but did not wait for a response from the remote user.

TNOTSUPPORT This subroutine is not supported by the underlying transport provider.

TOUTSTATE The subroutine was issued in the wrong sequence.

TPROTO This error indicates that a communication problem has been detected between the X/Open Transport Interface and the transport provider for which there is no other suitable X/Open Transport Interface (**t_errno**).

TSYSERR A system error has occurred during execution of this subroutine.

Related Information

The **t_accept** subroutine, **t_alloc** subroutine, **t_getinfo** subroutine, **t_listen** subroutine, **t_open** subroutine, **t_optmgmt** subroutine, **t_rcvconnect** subroutine.

t_error Subroutine for X/Open Transport Interface

Purpose

Produce error message.

Library

X/Open Transport Interface Library (**libxti.a**)

Syntax

```
#include <xti.h>

int t_error (
        const char *errmsg)
```

Description

The **t_error** subroutine produces a language–dependent message on the standard error output which describes the last error encountered during a call to a transport subroutine.

If the *errmsg* parameter is not a null pointer and the character pointed to be the *errmsg* parameter is not the null character, the error message is written as follows: the string pointed to by the *errmsg* parameter followed by a colon and a space and a standard error message string for the current error defined in **t_errno**. If **t_errno** has a value different from **TSYSERR**, the standard error message string is followed by a newline character. If, however, **t_errno** is equal to **TSYSERR**, the **t_errno** string is followed by the standard error message string for the current error defined in the *errno* global variable followed by a newline.

The language for error message strings written by the **t_error** subroutine is implementation–defined. If it is in English, the error message string describing the value in **t_errno** is identical to the comments following the **t_errno** codes defined in the **xti.h** header file. The contents of the error message strings describing the value in the *errno* global variable are the same as those returned by the **strerror** subroutine with an parameter of *errno.*

The error number, **t_errno**, is only set when an error occurs and it is not cleared on successful calls.

Parameter

errmsg Specifies a user–supplied error message that gives the context to the error.

Valid States

ALL – apart from T_UNINIT.

Return Values

Upon completion, a value of 0 is returned.

Errors Codes

No errors are defined for the **t_error** subroutine.

Examples

If a **t_connect** subroutine fails on transport endpoint `fd2` because a bad address was given, the following call might follow the failure:

```
t_error(”t_connect failed on fd2”);
```

The diagnostic message to be printed would look like:

```
t_connect failed on fd2: incorrect addr format
```

where `incorrect addr format` identifies the specific error that occurred, and `t_connect failed on fd2` tells the user which function failed on which transport endpoint.

Related Information

The **strerror** subroutine, **t_connect** subroutine.

t_free Subroutine for X/Open Transport Interface

Purpose

Free a library structure.

Library

X/Open Transport Interface Library (**libxti.a**)

Syntax

```
#include <xti.h>

int t_free (
        void *ptr;
        int struct_type)
```

Description

The **t_free** subroutine frees memory previously allocated by the **t_alloc** subroutine. This subroutine frees memory for the specified structure and buffers referenced by the structure.

The **t_free** subroutine checks the *addr, opt*, and *udata* fields of the given structure, as appropriate, and frees the buffers pointed to by the *buf* field of the **netbuf** structure. If *buf* is a null pointer, the **t_free** subroutine does not attempt to free memory. After all buffers are free, the **t_free** subroutine frees the memory associated with the structure pointed to by the *ptr* parameter.

Undefined results occur if the *ptr* parameter or any of the *buf* pointers points to a block of memory that was not previously allocated by the **t_alloc** subroutine.

Parameters

ptr Points to one of the seven structure types described for the **t_alloc** subroutine.

struct_type

Identifies the type of the structure specified by the *ptr* parameter. The type can be one of the following:

T_BIND	struct t_bind
T_CALL	struct t_call
T_OPTMGMT	struct t_optmgmt
T_DIS	struct t_discon
T_UNITDATA	struct t_unitdata
T_UDERROR	struct t_uderr
T_INFO	struct t_info

Each of these structures may subsequently be used as a parameter to one or more transport functions.

Valid States

ALL – apart from T_UNINIT.

Return Values

0	Successful completion.
–1	**t_errno** is set to indicate an error.

Error Codes

On failure, **t_errno** is set to one of the following:

TSYSERR	A system error has occurred during execution of this function.
TNOSTRUCTYPE	Unsupported *struct_type* parameter value requested.
TPROTO	This error indicates that a communication problem has been detected between the X/Open Transport Interface and the transport provider for which there is no other suitable X/Open Transport Interface (**t_errno**).

Related Information

The **t_alloc** subroutine.

t_getinfo Subroutine for X/Open Transport Interface

Purpose

Get protocol–specific service information.

Library

X/Open Transport Interface Library (**libxti.a**)

Syntax

```
#include <xti.h>

int t_getinfo (fd, info)
int fd;
struct t_info *info;
```

Description

The **t_getinfo** subroutine returns the current characteristics of the underlying transport protocol and/or transport connection associated with the file descriptor specified by the *fd* parameter. The pointer specified by the *info* parameter returns the same information returned by the **t_open** subroutine, although not necessarily precisely the same values. This subroutine enables a transport user to access this information during any phase of communication.

Parameters

fd Specifies the file descriptor.

info Points to a **t_info** structure which contains the following members:

```
long addr;     /* max size of the transport protocol       */
               /* address                                  */
long options;  /* max number of bytes of protocol-specific */
               /* options                                  */
long tsdu;     /* max size of a transport service data     */
               /* unit (TSDU)                              */
long etsdu;    /* max size of an expedited transport       */
               /* service data unit (ETSDU)                */
long connect;  /* max amount of data allowed on connection */
               /* establishment functions                  */
long discon;   /* max amount of data allowed on t_snddis   */
               /* and t_rcvdis functions                   */
long servtype; /* service type supported by the transport  */
               /* provider                                 */
long flags;    /* other info about the transport provider  */
```

The values of the fields have the following meanings:

`addr` A value greater than zero indicates the maximum size of a transport protocol address and a value of –2 specifies that the transport provider does not provide user access to transport protocol addresses.

`options` A value greater than zero indicates the maximum number of bytes of protocol–specific options supported by the provider, and a value of –2 specifies that the transport provider does not support options set by users.

`tsdu` A value greater than zero specifies the maximum size of a transport service data unit (TSDU); a value of zero specifies that the transport provider does not support the concept of TSDU, although it does support the sending of a datastream with no logical boundaries preserved across a connection; a value of –1 specifies that there is no limit on the size of a TSDU; and a value of –2 specifies that the transfer of normal data is not supported by the transport provider.

`etsdu` A value greater than zero specifies the maximum size of an expedited transport service data unit (ETSDU); a value of zero specifies that the transport provider does not support the concept of ETSDU, although it does support the sending of an expedited data stream with no logical boundaries preserved across a connection; a value of –1 specifies that there is no limit on the size of an ETSDU; and a value of –2 specifies that the transfer of expedited data is not supported by the transport provider. Note that the semantics of expedited data may be quite different for different transport providers (see Appendix A, ISO Transport Protocol Information and Appendix B, Internet Protocol–specific Information) .

`connect` A value greater than zero specifies the maximum amount of data that may be associated with connection establishment functions and a value of –2 specifies that the transport provider does not allow data to be sent with connection establishment functions.

`discon` A value greater than zero specifies the maximum amount of data that may be associated with the **t_snddis** and **t_rcvdis** subroutines and a value of –2 specifies that the transport provider does not allow data to be sent with the abortive release functions.

`servtype`
This field specifies the service type supported by the transport provider on return. The possible values are:

T_COTS The transport provider supports a connection–mode service but does not support the optional orderly release facility.

T_COTS_ORD The transport provider supports a connection–mode service with the optional orderly release facility.

T_CLTS The transport provider supports a connectionless–mode service. For this service type, the **t_open** subroutine will return –2 for `etsdu`, `connect` and `discon`.

`flags` This is a bit field used to specify other information about the transport provider. If the **T_SENDZERO** bit is set in flags, this indicates that the underlying transport provider supports the sending of zero–length TSDUs. See Appendix A, ISO Transport Protocol Information for a discussion of the separate issue of zero–length fragments within a TSDU.

If a transport user is concerned with protocol independence, the above sizes may be accessed to determine how large the buffers must be to hold each piece of information. Alternatively, the **t_alloc** subroutine may be used to allocate these buffers. An error results if a transport user exceeds the allowed data size on any subroutine. The value of each field may change as a result of protocol option negotiation during connection establishment (the **t_optmgmt** call has no affect on the values returned by the **t_getinfo** subroutine). These values will only change from the values presented to the **t_open** subroutine after the endpoint enters the **T_DATAXFER** state.

Valid States

ALL – apart from T_UNINIT.

Return Values

0	Successful completion.
–1	**t_errno** is set to indicate an error.

Error Codes

On failure, **t_errno** is set to one of the following:

TBADF	The specified file descriptor does not refer to a transport endpoint.
TSYSERR	A system error has occurred during execution of this subroutine.
TPROTO	This error indicates that a communication problem has been detected between the X/Open Transport Interface and the transport provider for which there is no other suitable X/Open Transport Interface (**t_errno**).

Related Information

The **t_alloc** subroutine, **t_open** subroutine.

t_getprotaddr Subroutine for X/Open Transport Interface

Purpose

Get the protocol addresses.

Library

X/Open Transport Interface Library (**libxti.a**)

Syntax

```
#include <xti.h>

int t_getprotaddr (fd, boundaddr, peeraddr)
int fd;
struct t_bind *boundaddr;
struct t_bind *peeraddr;
```

Description

The **t_getproaddr** subroutine returns local and remote protocol addresses currently associated with the transport endpoint specified by the *fd* parameter.

Parameters

fd Specifies the transport endpoint.

boundaddr

Specifies the local address to which the transport endpoint is to be bound. The *boundaddr* parameter has the following fields:

maxlen Specifies the maximum size of the address buffer.

buf Points to the buffer where the address is to be placed. On return, the *buf* field of *boundaddr* points to the address, if any, currently bound to *fd*.

len Specifies the length of the address. If the transport endpoint is in the **T_UNBND** state, zero is returned in the *len* field of *boundaddr.*

peeraddr

Specifies the remote protocol address associated with the transport endpoint.

maxlen Specifies the maximum size of the address buffer.

buf Points to the address, if any, currently connected to *fd.*

len Specifies the length of the address. If the transport endpoint is not in the **T_DATAXFER** state, zero is returned in the *len* field of *peeraddr.*

Valid States

ALL – apart from T_UNINIT.

Return Values

0 Successful completion.

–1 **t_errno** is set to indicate an error.

Error Codes

On failure, **t_errno** is set to one of the following:

TBADF	The specified file descriptor does not refer to a transport endpoint.
TBUFOVIEW	The number of bytes allocated for an incoming parameter (*maxlen*) is greater than 0 but not sufficient to store the value of that parameter.
TSYSERR	A system error has occurred during execution of this subroutine.
TPROTO	This error indicates that a communication problem has been detected between the X/Open Transport Interface and the transport provider for which there is no other suitable X/Open Transport Interface (**t_errno**).

Related Information

The **t_bind** subroutine.

t_getstate Subroutine for X/Open Transport Interface

Purpose

Get the current state.

Library

X/Open Transport Interface Library (**libxti.a**)

Syntax

```
#include <xti.h>

int t_getstate (fd)
int fd;
```

Description

The **t_getstate** subroutine returns the current state of the provider associated with the transport endpoint specified by the *fd* parameter.

Parameter

fd	Specifies the transport endpoint.

Valid States

ALL – apart from T_UNINIT.

Return Values

0	Successful completion.
–1	**t_errno** is set to indicate an error. The current state is one of the following:

T_UNBND	Unbound
T_IDLE	Idle
T_OUTCON	Outgoing connection pending
T_INCON	Incoming connection pending
T_DATAXFER	Data transfer
T_OUTREL	Outgoing orderly release (waiting for an orderly release indication)
T_INREL	Incoming orderly release (waiting to send an orderly release request)

If the provider is undergoing a state transition when the **t_getstate** subroutine is called, the subroutine will fail.

Error Codes

On failure, **t_errno** is set to one of the following:

TBADF	The specified file descriptor does not refer to a transport endpoint.
TSTATECHNG	The transport provider is undergoing a transient state change.

TSYSERR A system error has occurred during execution of this subroutine.

TPROTO This error indicates that a communication problem has been detected between the X/Open Transport Interface and the transport provider for which there is no other suitable X/Open Transport Interface (**t_errno**).

Related Information

The **t_open** subroutine.

t_listen Subroutine for X/Open Transport Interface

Purpose

Listen for a connect indication.

Library

X/Open Transport Interface Library (**libxti.a**)

Syntax

```
#include <xti.h>

int t_listen (fd, call)
int fd;
struct t_call *call;
```

Description

The **t_listen** subroutine listens for a connect request from a calling transport user.

By default, the **t_listen** subroutine executes in synchronous mode and waits for a connect indication to arrive before returning to the user. However, if **O_NONBLOCK** is set via the **t_open** subroutine or with the **fcntl** subroutine (**F_SETFL**), the **t_listen** subroutine executes asynchronously, reducing to a poll for existing connect indications. If none are available, the subroutine returns –1 and sets **t_errno** to **TNODATA**.

Parameters

fd Identifies the local transport endpoint where connect indications arrive.

call

Contains information describing the connect indication. The parameter *call* points to a **t_call** structure which contains the following members:

```
struct netbuf addr;
struct netbuf opt;
struct netbuf udata;
int sequence;
```

In this structure, the fields have the following meanings:

addr Returns the protocol address of the calling transport user. This address is in a format usable in future calls to the **t_connect** subroutine. Note, however that **t_connect** may fail for other reasons, for example, **TADDRBUSY**.

opt Returns options associated with the connect request.

udata Returns any user data sent by the caller on the connect request.

sequence A number that uniquely identifies the returned connect indication. The value of *sequence* enables the user to listen for multiple connect indications before responding to any of them.

Since this subroutine returns values for the *addr, opt* and *udata* fields of the *call* parameter, the *maxlen* field of each must be set before issuing the **t_listen** subroutine to indicate the maximum size of the buffer for each.

Valid States

T_IDLE, **T_INCON**.

Return Values

0	Successful completion.
–1	**t_errno** is set to indicate an error.

Error Codes

On failure, **t_errno** is set to one of the following:

TBADF	The specified file descriptor does not refer to a transport endpoint.
TBADQLEN	The *qlen* parameter of the endpoint referenced by the *fd* parameter is zero.
TBODATA	**O_NONBLOCK** was set, but no connect indications had been queued.
TBUFOVFLW	The number of bytes allocated for an incoming parameter (*maxlen*) is greater than 0 but not sufficient to store the value of that parameter. The provider's state, as seen by the user, changes to **T_INCON**, and the connect indication information to be returned in the *call* parameter is discarded. The value of the *sequence* parameter returned can be used to do a **t_snddis**.
TLOOK	An asynchronous event has occurred on the transport endpoint and requires immediate attention.
TNOTSUPPORT	This subroutine is not supported by the underlying transport provider.
TOUTSTATE	The subroutine was issued in the wrong sequence on the transport endpoint referenced by the *fd* parameter.
TPROTO	This error indicates that a communication problem has been detected between the X/Open Transport Interface and the transport provider for which there is no other suitable X/Open Transport Interface (**t_errno**).
TQFULL	The maximum number of outstanding indications has been reached for the endpoint referenced by the *fd* parameter.
TSYSERR	A system error has occurred during execution of this subroutine.

Implementation Specifics

Some transport providers do not differentiate between a connect indication and the connection itself. If this is the case, a successful return of **t_listen** indicates an existing connection (see Appendix B, Internet Protocol–specific Information).

Related Information

The **fcntl** subroutine, **t_accept** subroutine, **t_alloc** subroutine, **t_bind** subroutine, **t_connect** subroutine, **t_open** subroutine, **t_optmgmt** subroutine, **t_rcvconnect** subroutine.

t_look Subroutine for X/Open Transport Interface

Purpose

Look at the current event on a transport endpoint.

Library

X/Open Transport Interface Library (**libxti.a**)

Syntax

```
#include <xti.h>

int t_look (fd)
int fd;
```

Description

The **t_look** subroutine returns the current event on the transport endpoint specified by the *fd* parameter. This subroutine enables a transport provider to notify a transport user of an asynchronous event when the user is calling subroutines in synchronous mode. Certain events require immediate notification of the user and are indicated by a specific error, **TLOOK**, on the current or next subroutine to be executed. Details on events which cause subroutines to fail, **T_LOOK**, may be found in Section 4.6, Events and TLOOK Error Indication.

This subroutine also enables a transport user to poll a transport endpoint periodically for asynchronous events.

Parameter

fd	Specifies the transport endpoint.

Valid States

ALL – apart from **T_UNINIT**.

Return Values

Upon success, the **t_look** subroutine returns a value that indicates which of the allowable events has occurred, or returns zero if no event exists. One of the following events is returned:

T_LISTEN	Connection indication received.
T_CONNECT	Connect confirmation received.
T_DATA	Normal data received.
T_EXDATA	Expedited data received.
T_DISCONNECT	Disconnect received.
T_UDERR	Datagram error indication.
T_ORDREL	Orderly release indication.
T_GODATA	Flow control restrictions on normal data flow that led to a **TFLOW** error have been lifted. Normal data may be sent again.
T_GOEXDATA	Flow control restrictions on expedited data flow that led to a **TFLOW** error have been lifted. Expedited data may be sent again.

On failure, –1 is returned and **t_errno** is set to indicate the error.

Error Codes

On failure, **t_errno** is set to one of the following:

TBADF	The specified file descriptor does not refer to a transport endpoint.
TSYSERR	A system error has occurred during execution of this subroutine.
TPROTO	This error indicates that a communication problem has been detected between the X/Open Transport Interface and the transport provider for which there is no other suitable X/Open Transport Interface (**t_errno**).

Implementation Specifics

Additional functionality is provided through the Event Management (EM) interface.

Related Information

The **t_open** subroutine, **t_snd** subroutine, **t_sndudata** subroutine.

t_open Subroutine for X/Open Transport Interface

Purpose

Establish a transport endpoint.

Library

X/Open Transport Interface Library (**libxti.a**)

Syntax

```
#include <xti.h>

#include <fcntl.h>

int t_open (
        const char *name;
        int oflag;
        struct t_info *info)
```

Description

The **t_open** subroutine must be called as the first step in the initialization of a transport endpoint. This subroutine establishes a transport endpoint by supplying a transport provider identifier that indicates a particular transport provider (for example, transport protocol) and returning a file descriptor that identifies that endpoint.

This subroutine also returns various default characteristics of the underlying transport protocol by setting fields in the **t_info** structure.

Parameters

name Points to a transport provider identifier.

oflag Identifies any open flags (as in the **open** exec) . The *oflag* parameter is constructed from **O_RDWR** optionally bitwise inclusive–OR–ed with **O_NONBLOCK**. These flags are defined by the **fcntl.h** header file. The file descriptor returned by the **t_open** subroutine is used by all subsequent subroutines to identify the particular local transport endpoint.

info

Points to a **t_info** structure which contains the following members:

```
long addr;      /* max size of the transport protocol      */
                /* address                                 */
long options;   /* max number of bytes of                  */
                /* protocol-specific options               */
long tsdu;      /* max size of a transport service data    */
                /* unit (TSDU)                             */
long etsdu;     /* max size of an expedited transport      */
                /* service data unit (ETSDU)               */
long connect;   /* max amount of data allowed on           */
                /* connection establishment subroutines    */
long discon;    /* max amount of data allowed on           */
                /* t_snddis and t_rcvdis subroutines       */
long servtype;  /* service type supported by the           */
                /* transport provider                      */
long flags;     /* other info about the transport provider */
```

The values of the fields have the following meanings:

`addr` A value greater than zero indicates the maximum size of a transport protocol address and a value of –2 specifies that the transport provider does not provide user access to transport protocol addresses.

`options` A value greater than zero indicates the maximum number of bytes of protocol–specific options supported by the provider, and a value of –2 specifies that the transport provider does not support user–settable options.

`tsdu` A value greater than zero specifies the maximum size of a transport service data unit (TSDU); a value of zero specifies that the transport provider does not support the concept of TSDU, although it does support the sending of a data stream with no logical boundaries preserved across a connection; a value of –1 specifies that there is no limit on the size of a TSDU; and a value of –2 specifies that the transfer of normal data is not supported by the transport provider.

`etsdu` A value greater than zero specifies the maximum size of an expedited transport service data unit (ETSDU); a value of zero specifies that the transport provider does not support the concept of ETSDU, although it does support the sending of an expedited data stream with no logical boundaries preserved across a connection; a value of –1 specifies that there is no limit on the size of an ETSDU; and a value of –2 specifies that the transfer of expedited data is not supported by the transport provider. Note that the semantics of expedited data may be quite different for different transport providers.

`connect` A value greater than zero specifies the maximum amount of data that may be associated with connection establishment subroutines and a value of –2 specifies that the transport provider does not allow data to be sent with connection establishment subroutines.

`discon` A value greater than zero specifies the maximum amount of data that may be associated with the **t_synddis** and **t_rcvdis** subroutines and a value of –2 specifies that the transport provider does not allow data to be sent with the abortive release subroutines.

`servtype` This field specifies the service type supported by the transport provider. The valid values on return are:

T_COTS The transport provider supports a connection–mode service but does not support the optional orderly release facility.

T_COTS_ORD The transport provider supports a connection–mode service with the optional orderly release facility.

T_CLTS The transport provider supports a connectionless–mode service. For this service type, **t_open** will return –2 for `etsdu`, `connect` and `discon`.

A single transport endpoint may support only one of the above services at one time.

`flags` This is a bit field used to specify other information about the transport provider. If the **T_SENDZERO** bit is set in flags, this indicates the underlying transport provider supports the sending of zero–length TSDUs.

If a transport user is concerned with protocol independence, the above sizes may be accessed to determine how large the buffers must be to hold each piece of information.

Alternatively, the **t_alloc** subroutine may be used to allocate these buffers. An error will result if a transport user exceeds the allowed data size on any subroutine.

If the *info* parameter is set to a null pointer by the transport user, no protocol information is returned by the **t_open** subroutine.

Valid States

T_UNINIT

Return Values

Valid file descriptor	Successful completion.
–1	**t_errno** is set to indicate an error.

Error Codes

On failure, **t_errno** is set to one of the following:

TBADFLAG	An invalid flag is specified.
TBADNAME	Invalid transport provider name.
TSYSERR	A system error has occurred during execution of this subroutine.
TPROTO	This error indicates that a communication problem has been detected between the X/Open Transport Interface and the transport provider for which there is no other suitable X/Open Transport Interface (**t_errno**).

Related Information

The **t_open** subroutine.

t_optmgmt Subroutine for X/Open Transport Interface

Purpose

Manage options for a transport endpoint.

Library

X/Open Transport Interface Library (**libxti.a**)

Syntax

```
#include <xti.h>

int t_optmgmt (fd, req, ret)
int fd;
struct t_optmgmt *req;
struct t_optmgmt *ret;
```

Description

The **t_optmgmt** subroutine enables a transport user to retrieve, verify, or negotiate protocol options with the transport provider.

The *req* and *ret* parameters both point to a **t_optmgmt** structure containing the following members:

```
struct netbuf opt;
long flags;
```

Within this structure, the fields have the following meaning:

opt

Identifies protocol options. The options are represented by a **netbuf** structure in a manner similar to the address in the **t_bind** subroutine:

len Specifies the number of bytes in the options and on return, specifies the number of bytes of options returned.

buf Points to the options buffer. For the *ret* parameter, *buf* points to the buffer where the options are to be placed. Each option in the options buffer is of the form **struct t_opthdr** possibly followed by an option value. The fields of this structure and the values are:

level Identifies the X/Open Transport Interface level or a protocol of the transport provider.

name Identifies the option within the level.

len Contains its total length, for example, the length of the option header **t_opthdr** plus the length of the option value. If **t_optmgmt** is called with the action **T_NEGOTIATE** set.

status Contains information about the success or failure of a negotiation.

Each option in the input or output option buffer must start at a long–word boundary. The macro **OPT_NEXTHDR (***pbuf, buflen, poption***)** can be used for that purpose. The macro parameters are as follows:

pbuf Specifies a pointer to an option buffer *opt.buf.*

buflen The length of the option buffer pointed to by *pbuf*.

poption Points to the current option in the option buffer. **OPT_NEXTHDR** returns a pointer to the position of the next option or returns a null pointer if the option buffer is exhausted. The macro is helpful for writing and reading. See the **xti.h** header file for the exact definition of this structure.

If the transport user specifies several options on input, all options must address the same level.

If any option in the options buffer does not indicate the same level as the first option, or the level specified is unsupported, then the **t_optmgmt** request fails with **TBADOPT**. If the error is detected, some options may have successfully negotiated. The transport user can check the current status by calling the **t_optmgmt** subroutine with the **T_CURRENT** flag set.

Note: ”The Use of Options” contains a detailed description about the use of options and should be read before using this subroutine.

maxlen Has no meaning for the *req* parameter, but must be set in the *ret* parameter to specify the maximum size of the options buffer. On return, *len* specifies the number of bytes of options returned. The value in *maxlen* has no meaning for the *req* argument,

flags

Specifies the action to take with those options. The *flags* field of *req* must specify one of the following actions:

T_CHECK
This action enables the user to verify whether the options specified in the *req* parameter are supported by the transport provider. If an option is specified with no option value, (that is, it consists only of a **t_opthdr** structure), the option is returned with its *status* field set to one of the following:

- **T_SUCCESS** – if it is supported.
- **T_NOTSUPPORT** – if it is not or needs additional user privileges.
- **T_READONLY** – if it is read–only (in the current X/Open Transport Interface state).

No option value is returned. If an option is specified with an option value, the *status* field of the returned option has the same value, as if the user had tried to negotiate this value with **T_NEGOTIATE**. If the status is **T_SUCCESS**, **T_FAILURE**, **T_NOTSUPPORT**, or **T_READONLY**, the returned option value is the same as the one requested on input.

The overall result of the option checks is returned in the *flags* field of the **netbuf** structure pointed to by the *ret* parameter. This field contains the worst single result of the option checks, where the rating is the same as for **T_NEGOTIATE**.

Note, that no negotiation takes place. All currently effective option values remain unchanged.

T_CURRENT
This action enables the transport user to retrieve the currently effective option values. The user specifies the options of interest in the *opt* fields in the **netbuf** structure pointed to by the *req* parameter. The option values are irrelevant and will be ignored; it is sufficient to specify the **t_opthdr** part of an option only. The currently effective values are then returned in *opt* fields in the **netbuf** structure pointed to by the *ret* parameter.

The *status* field returned is on of the following:

- **T_NOTSUPPORT** if the protocol level does not support this option or the transport user illegally requested a privileged option.
- **T_READONLY** if the option is read–only.
- **T_SUCCESS** in all other cases.

The overall result of the option checks is returned in the *flags* field of the **netbuf** structure pointed to by the *ret* parameter. This field contains the worst single result of the option checks, where the rating is the same as for **T_NEGOTIATE**.

For each level, the **T_ALLOPT** option (see below) can be requested on input. All supported options of this level with their default values are then returned.

T_DEFAULT
This action enables the transport user to retrieve the default option values. The user specifies the options of interest in the *opt* fields in the **netbuf** structure pointed to by the *req* parameter. The option values are irrelevant and will be ignored; it is sufficient to specify the **t_opthdr** part of an option only. The default values are then returned in the *opt* field of the **netbuf** structure pointed to by the *ret* parameter.

The *status* field returned is one of the following:

- **T_NOTSUPPORT** if the protocol level does not support this option or the transport user illegally requested a privileged option.
- **T_READONLY** if the option is read–only.
- **T_SUCCESS** in all other cases.

The overall result of the option checks is returned in the *flags* field of the *ret* parameter **netbuf** structure. This field contains the worst single result of the option checks, where the rating is the same as for **T_NEGOTIATE**.

For each level, the **T_ALLOPT** option (see below) can be requested on input. All supported options of this level with their default values are then returned. In this case, the *maxlen* value of the *opt* field in the *ret* parameter **netbuf** structure must be given at least the value of the *options* field of the *info* parameter (see the **t_getinfo** or **t_open subroutines**) before the call.

T_NEGOTIATE
This action enables the transport user to negotiate option values. The user specifies the options of interest and their values in the buffer specified in the *req* parameter **netbuf** structure. The negotiated option values are returned in the buffer pointed to by the *opt* field of the *ret* parameter **netbuf** structure. The *status* field of each returned option is set to indicate the result of the negotiation. The value is one of the following:

- **T_SUCCESS** if the proposed value was negotiated.
- **T_PARTSUCCESS** if a degraded value was negotiated.
- **T_FAILURE** is the negotiation failed (according to the negotiation rules).
- **T_NOTSUPPORT** if the transport provider does not support this option or illegally requests negotiation of a privileged option
- **T_READONLY** if modification of a read–only option was requested.

If the status is **T_SUCCESS**, **T_FAILURE**, **T_NOTSUPPORT** or **T_READONLY**, the returned option value is the same as the one requested on input.

The overall result of the negotiation is returned in the *flags* field of the *ret* parameter **netbuf** structure. This field contains the worst single result, whereby the rating is done according to the following order, where **T_NOTSUPPORT** is the worst result and **T_SUCCESS** is the best:

- **T_NOTSUPPORT**
- **T_READONLY**
- **T_FAILURE**
- **T_PARTSUCCESS**
- **T_SUCCESS.**

  For each level, the **T_ALLOPT** option (see below) can be requested on input. This option has no value and consists of a **t_opthdr** only. This input requests negotiation of all supported options of this level to their default values. The result is returned option by option in the *opt* field of the structure pointed to in the *ret* parameter. Depending on the state of the transport endpoint, not all requests to negotiate the default value may be successful.

The **T_ALLOPT** option can only be used with the **t_optmgmt** structure and the actions **T_NEGOTIATE**, **T_DEFAULT** and **T_CURRENT**. This option can be used with any supported level and addresses all supported options of this level. The option has no value and consists of a **t_opthdr** only. Since only options of one level may be addressed in a **t_optmgmt** call, this option should not be requested together with other options. The subroutine returns as soon as this option has been processed.

Options are independently processed in the order they appear in the input option buffer. If an option is multiply input, it depends on the implementation whether it is multiply output or whether it is returned only once.

Transport providers may not be able to provide an interface capable of supporting **T_NEGOTIATE** and/or **T_CHECK** functionalities. When this is the case, the error **TNOTSUPPORT** is returned.

The subroutine **t_optmgmt** may block under various circumstances and depending on the implementation. For example, the subroutine will block if the protocol addressed by the call resides on a separate controller. It may also block due to flow control constraints, if data previously sent across this transport endpoint has not yet been fully processed. If the subroutine is interrupted by a signal, the option negotiations that have been done so far may remain valid. The behavior of the subroutine is not changed if **O_NONBLOCK** is set.

Parameters

fd	Identifies a transport endpoint.
req	Requests a specific action of the provider.
ret	Returns options and flag values to the user.

X/Open Transport Interface–Level Options

X/Open Transport Interface (XTI) level options are not specific for a particular transport provider. An XTI implementation supports none, all, or any subset of the options defined below. An implementation may restrict the use of any of these options by offering them only in the privileged or read–only mode, or if the bound transport endpoint identified by the *fd* parameter relates to specific transport providers.

The subsequent options are not association–related (see Chapter 5, The Use of Options) . They may be negotiated in all XTI states except **T_UNINIT**.

The protocol level is **XTI_GENERIC**. For this level, the following options are defined:

XTI–Level Options			
Option Name	**Type of Option Value**	**Legal Option Value**	**Meaning**
XTI_DEBUG	array of unsigned longs	see text	enable debugging
XTI_LINGER	struct linger	see text	linger on close if data is present
XTI_RCVBUF	unsigned long	size in octets	receive buffer size
XTI_RCVLOWAT	unsigned long	size in octets	receive low–water mark
XTI_SNDBUF0	unsigned long	size in octets	send buffer size
XTI_SNDLOWAT	unsigned long	size in octets	send low–water mark

A request for **XTI_DEBUG** is an absolute requirement. A request to activate **XTI_LINGER** is an absolute requirement; the timeout value to this option is not. **XTI_RCVBUF**, **XTI_RCVLOWAT**, **XTI_SNDBUF** and **XTI_SNDLOWAT** are not absolute requirements.

XTI_DEBUG Enables debugging. The values of this option are implementation–defined. Debugging is disabled if the option is specified with no value (for example, with an option header only).

The system supplies utilities to process the traces. An implementation may also provide other means for debugging.

XTI_LINGER

Lingers the execution of a **t_close** subroutine or the **close** exec if send data is still queued in the send buffer. The option value specifies the linger period. If a **close** exec or **t_close** subroutine is issued and the send buffer is not empty, the system attempts to send the pending data within the linger period before closing the endpoint. Data still pending after the linger period has elapsed is discarded.

Depending on the implementation, the **t_close** subroutine or **close** exec either, at a maximum, block the linger period, or immediately return, whereupon, at most, the system holds the connection in existence for the linger period.

The option value consists of a **structure t_linger** declared as:

```
struct t_linger {
   long l_onoff;
   long l_linger;
}
```

The fields of the structure and the legal values are:

l_onoff Switches the option on or off. The value *l_onoff* is an absolute requirement. The possible values are:

T_NO switch option off

T_YES activate option

l_linger Determines the linger period in seconds. The transport user can request the default value by setting the field to **T_UNSPEC**. The default timeout value depends on the underlying transport provider (it is often **T_INFINITE**). Legal values for this field are **T_UNSPEC**, **T_INFINITE** and all non–negative numbers.

The *l_linger* value is not an absolute requirement. The implementation may place upper and lower limits to this value. Requests that fall short of the lower limit are negotiated to the lower limit.

Note: Note that this option does not linger the execution of the **t_snddis** subroutine.

XTI_RCVBUF Adjusts the internal buffer size allocated for the receive buffer. The buffer size may be increased for high–volume connections, or decreased to limit the possible backlog of incoming data.

This request is not an absolute requirement. The implementation may place upper and lower limits on the option value. Requests that fall short of the lower limit are negotiated to the lower limit.

Legal values are all positive numbers.

XTI_RCVLOWAT Sets a low–water mark in the receive buffer. The option value gives the minimal number of bytes that must have accumulated in the receive buffer before they become visible to the transport user. If and when the amount of accumulated receive data exceeds the low–water mark, a **T_DATA** event is created, an event mechanism (for example, the **poll** or **select** subroutines) indicates the data, and the data can be read by the **t_rcv** or **t_rcvudata** subroutines.

This request is not an absolute requirement. The implementation may place upper and lower limits on the option value. Requests that fall short of the lower limit are negotiated to the lower limit.

Legal values are all positive numbers.

XTI_SNDBUF Adjusts the internal buffer size allocated for the send buffer.

This request is not an absolute requirement. The implementation may place upper and lower limits on the option value. Requests that fall short of the lower limit are negotiated to the lower limit.

Legal values are all positive numbers.

XTI_SNDLOWAT Sets a low–water mark in the send buffer. The option value gives the minimal number of bytes that must have accumulated in the send buffer before they are sent.

This request is not an absolute requirement. The implementation may place upper and lower limits on the option value. Requests that fall short of the lower limit are negotiated to the lower limit.

Legal values are all positive numbers.

Valid States

ALL – except from **T_UNINIT**.

Return Values

0	Successful completion.
–1	**t_errno** is set to indicate an error.

Error Codes

On failure, **t_errno** is set to one of the following:

TACCES	The user does not have permission to negotiate the specified options.
TBADF	The specified file descriptor does not refer to a transport endpoint.
TBADFLAG	An invalid flag was specified.
TBADOPT	The specified options were in an incorrect format or contained illegal information.
TBUFOVFLW	The number of bytes allowed for an incoming argument (*maxlen*) is greater than 0 but not sufficient to store the value of that argument. The information to be returned in *ret* will be discarded.
TOUTSTATE	The subroutine was issued in the wrong sequence.
TPROTO	This error indicates that a communication problem has been detected between the X/Open Transport Interface and the transport provider for which there is no other suitable X/Open Transport Interface (**t_errno**).
TSYSERR	A system error has occurred during execution of this subroutine.

Related Information

The **t_accept** subroutine, **t_alloc** subroutine, **t_connect** subroutine, **t_getinfo** subroutine, **t_listen** subroutine, **t_open** subroutine, **t_rcvconnect** subroutine.

t_rcv Subroutine for X/Open Transport Interface

Purpose

Receive data or expedited data sent over a connection.

Library

X/Open Transport Interface Library (**libxti.a**)

Syntax

```
#include <xti.h>

int t_rcv (
        int fd;
        void *buf;
        unsigned int nbytes;
        int *flags)
```

Description

The **t_rcv** subroutine receives either normal or expedited data. By default, the **t_rcv** subroutine operates in synchronous mode and waits for data to arrive if none is currently available. However, if **O_NONBLOCK** is set via the **t_open** subroutine or the *fcntl* parameter, the, **t_rcv** subroutine executes in asynchronous mode and fails if no data is available. (See the **TNODATA** error in ”Error Codes” below.)

Parameters

fd	Identifies the local transport endpoint through which data will arrive.
buf	Points to a receive buffer where user data will be placed.

nbytes Specifies the size of the receive buffer.

flags

Specifies optional flags. This parameter may be set on return from the **t_rcv** subroutine. The possible values are:

T_MORE If set, on return from the call, indicates that there is more data, and the current transport service data unit (TSDU) or expedited transport service data unit (ETSDU) must be received in multiple **t_rcv** calls. In the asynchronous mode, the **T_MORE** flag may be set on return from the **t_rcv** call even when the number of bytes received is less than the size of the receive buffer specified. Each **t_rcv** call with the **T_MORE** flag set, indicates that another **t_rcv** call must follow to get more data for the current TSDU. The end of the TSDU is identified by the return of a **t_rcv** call with the **T_MORE** flag not set. If the transport provider does not support the concept of a TSDU as indicated in the *info* parameter on return from the **t_open** or **t_getinfo** subroutines, the **T_MORE** flag is not meaningful and should be ignored. If the *nbytes* parameter is greater than zero on the call to **t_rcv**, **t_rcv** returns 0 only if the end of a TSDU is being returned to the user.

T_EXPEDITED If set, the data returned is expedited data. If the number of bytes of expedited data exceeds the value of the *nbytes* parameter, **t_rcv** will set **T_EXPEDITED** and **T_MORE** on return from the initial call. Subsequent calls to retrieve the remaining ETSDU will have **T_EXPEDITED** set on return. The end of the ETSDU is identified by the return of a **t_rcv** call with the **T_MORE** flag not set.

In synchronous mode, the only way to notify the user of the arrival of normal or expedited data is to issue this subroutine or check for the **T_DATA** or **T_EXDATA** events using the **t_look** subroutine. Additionally, the process can arrange to be notified via the Event Management interface.

Valid States

T_DATAXFER, T_OUTREL.

Return Values

On successful completion, the **t_rcv** subroutine returns the number of bytes received. Otherwise, it returns –1 on failure and **t_errno** is set to indicate the error.

Error Codes

On failure, **t_errno** is set to one of the following:

TBADF The specified file descriptor does not refer to a transport endpoint.

TLOOK An asynchronous event has occurred on this transport endpoint and requires immediate attention.

TNODATA **O_NONBLOCK** was set, but no data is currently available from the transport provider.

TNOTSUPPORT This subroutine is not supported by the underlying transport provider.

TOUTSTATE The subroutine was issued in the wrong sequence on the transport endpoint referenced by the *fd* parameter.

TPROTO This error indicates that a communication problem has been detected between the X/Open Transport Interface and the transport provider for which there is no other suitable X/Open Transport Interface (**t_errno**).

TSYSERR A system error has occurred during execution of this subroutine.

Related Information

The **fcntl** subroutine, **t_getinfo** subroutine, **t_look** subroutine, **t_open** subroutine, **t_snd** subroutine.

t_rcvconnect Subroutine for X/Open Transport Interface

Purpose

Receive the confirmation from a connect request.

Library

X/Open Transport Interface Library (**libxti.a**)

Syntax

```
#include <xti.h>

int t_rcvconnect (fd, call)
int fd;
struct t_call *call;
```

Description

The **t_rcvconnect** subroutine enables a calling transport user to determine the status of a previously sent connect request and is used in conjunction with the **t_connect** subroutine to establish a connection in asynchronous mode. The connection is established on successful completion of this subroutine.

Parameters

fd Identifies the local transport endpoint where communication will be established.

call

Contains information associated with the newly established connection. The *call* parameter points to a **t_call** structure which contains the following members:

```
struct netbuf addr;
struct netbuf opt;
struct netbuf udata;
int sequence;
```

The fields of the **t_call** structure are:

addr Returns the protocol address associated with the responding transport endpoint.

opt Presents any options associated with the connection.

udata Points to optional user data that may be returned by the destination transport user during connection establishment.

sequence Has no meaning for this subroutine.

The *maxlen* field of each **t_call** member must be set before issuing this subroutine to indicate the maximum size of the buffer for each. However, the vale of the *call* parameter may be a null pointer, in which case no information is given to the user on return from the **t_rcvconnect** subroutine. By default, the **t_rcvconnect** subroutine executes in synchronous mode and waits for the connection to be established before returning. On return, the *addr, opt* and *udata* fields reflect values associated with the connection.

If **O_NONBLOCK** is set (via the **t_open** subroutine or **fcntl**), the **t_rcvconnect** subroutine executes in asynchronous mode, and reduces to a poll for existing connect confirmations. If none are available, the **t_rcvconnect** subroutine fails and returns immediately without waiting for the connection to be established. (See **TNODATA** in ”Error Codes” below.) In this case, the **t_rcvconnect** subroutine must be called again to complete the connection establishment phase and retrieve the information returned in the *call* parameter.

Valid States

T_OUTCON

Return Values

0 Successful completion.

–1 **t_errno** is set to indicate an error.

Error Codes

On failure, **t_errno** is set to one of the following:

TBADF	The specified file descriptor does not refer to a transport endpoint.
TBUFOVFLW	The number of bytes allocated for an incoming argument (*maxlen*) is greater than 0 but not sufficient to store the value of that argument, and the connect information to be returned in *call* will be discarded. The provider's state, as seen by the user, will be changed to **T_DATAFER**.
TLOOK	An asynchronous event has occurred on the transport connection and requires immediate attention.
TNODATA	**O_NONBLOCK** was set, but a connect confirmation has not yet arrived.
TNOTSUPPORT	This subroutine is not supported by the underlying transport provider.
TOUTSTATE	The subroutine was issued in the wrong sequence on the transport endpoint referenced by the *fd* parameter.
TPROTO	This error indicates that a communication problem has been detected between the X/Open Transport Interface and the transport provider for which there is no other suitable X/Open Transport Interface (**t_errno**).
TSYSERR	A system error has occurred during execution of this subroutine.

Related Information

The **t_accept** subroutine, **t_alloc** subroutine, **t_bind** subroutine, **t_connect** subroutine, **t_listen** subroutine, **t_open** subroutine, **t_optmgmt** subroutine.

t_rcvdis Subroutine for X/Open Transport Interface

Purpose

Retrieve information from disconnect.

Library

X/Open Transport Interface Library (**libxti.a**)

Syntax

```
#include <xti.h>

int t_rcvdis (fd, discon)
int fd;
struct t_discon *discon;
```

Description

The **t_rcvdis** subroutine identifies the cause of a disconnect and retrieves any user data sent with the disconnect.

Parameters

fd Identifies the local transport endpoint where the connection existed.

discon

Points to a **t_discon** structure containing the following members:

```
struct netbuf udata;
int reason;
int sequence;
```

The **t_discon** structure fields are:

reason Specifies the reason for the disconnect through a protocol–dependent reason code.

udata Identifies any user data that was sent with the disconnect.

sequence May identify an outstanding connect indication with which the disconnect is associated. The *sequence* field is only meaningful when the **t_rcvdis** subroutine is issued by a passive transport user who has executed one or more **t_listen** subroutines and is processing the resulting connect indications. If a disconnect indication occurs, the *sequence* field can be used to identify which of the outstanding connect indications is associated with the disconnect.

If a user does not care if there is incoming data and does not need to know the value of the *reason* or *sequence* fields, the *discon* field value may be a null pointer and any user data associated with the disconnect will be discarded. However, if a user has retrieved more than one outstanding connect indication (via the **t_listen** subroutine) and the *discon* field value is a null pointer, the user will be unable to identify with which connect indication the disconnect is associated.

Valid States

T_DATAXFER, **T_OUTCON**, **T_OUTREL**, **T_INREL**, **T_INCON**(*ocnt* > 0).

Return Values

0	Successful completion.
–1	**t_errno** is set to indicate an error.

Error Codes

On failure, **t_errno** is set to one of the following:

TBADF	The specified file descriptor does not refer to a transport endpoint.
TBUFOVFLW	The number of bytes allocated for incoming data (*maxlen*) is greater than 0 but not sufficient to store the data. If the *fd* parameter is a passive endpoint with *ocnt* > 1, it remains in state **T_INCON**; otherwise, the endpoint state is set to **T_IDLE**.
TNODIS	No disconnect indication currently exists on the specified transport endpoint.
TNOTSUPPORT	This subroutine is not supported by the underlying transport provider.
TOUTSTATE	The subroutine was issued in the wrong sequence on the transport endpoint referenced by the *fd* parameter.
TPROTO	This error indicates that a communication problem has been detected between the X/Open Transport Interface and the transport provider for which there is no other suitable X/Open Transport Interface (**t_errno**).
TSYSERR	A system error has occurred during execution of this subroutine.

Related Information

The **t_alloc** subroutine, **t_connect** subroutine, **t_listen** subroutine, **t_open** subroutine, **t_snddis** subroutine.

t_rcvrel Subroutine for X/Open Transport Interface

Purpose

Acknowledging receipt of an orderly release indication.

Library

X/Open Transport Interface Library (**libxti.a**)

Syntax

```
#include <xti.h>

int t_rcvrel (fd)
int fd;
```

Description

The **t_rcvrel** subroutine is used to acknowledge receipt of an orderly release indication. After receipt of this indication, the user may not attempt to receive more data because such an attempt will block forever. However, the user may continue to send data over the connection if the **t_sndrel** subroutine has not been called by the user. This function is an optional service of the transport provider, and is only supported if the transport provider returned the **T_COTS_ORD** service type on **t_open** or **t_getinfo** calls.

Parameter

fd	Identifies the local transport endpoint where the connection exists.

Valid States

T_DATAXFER, **T_OUTREL**.

Return Values

0	Successful completion.
–1	**t_errno** is set to indicate an error.

Error Codes

On failure, **t_errno** is set to one of the following:

TBADF	The specified file descriptor does not refer to a transport endpoint.
TLOOK	An asynchronous event has occurred on this transport endpoint and requires immediate attention.
TNOREL	No orderly release indication currently exists on the specified transport endpoint.
TNOTSUPPORT	This subroutine is not supported by the underlying transport provider.
TOUTSTATE	The subroutine was issued in the wrong sequence on the transport endpoint referenced by the *fd* parameter.
TPROTO	This error indicates that a communication problem has been detected between the X/Open Transport Interface and the transport provider for which there is no other suitable X/Open Transport Interface (**t_errno**).
TSYSERR	A system error has occurred during execution of this subroutine.

Related Information

The **t_getinfo** subroutine, **t_open** subroutine, **t_sndrel** subroutine.

t_rcvudata Subroutine for X/Open Transport Interface

Purpose

Receive a data unit.

Library

X/Open Transport Interface Library (**libxti.a**)

Syntax

```
#include <xti.h>

int t_rcvudata (fd, unitdata, flags)
int fd;
struct t_unitdata *unitdata;
int *flags;
```

Description

The **t_rcvudata** subroutine is used in connectionless mode to receive a data unit from another transport user.

By default, the **t_rcvudata** subroutine operates in synchronous mode and waits for a data unit to arrive if none is currently available. However, if **O_NONBLOCK** is set (via the **t_open** subroutine or *fcntl*), the **t_rcvudata** subroutine executes in asynchronous mode and fails if no data units are available.

If the buffer defined in the *udata* field of the *unitdata* parameter is not large enough to hold the current data unit, the buffer is filled and **T_MORE** is set in the *flags* parameter on return to indicate that another **t_rcvudata** subroutine should be called to retrieve the rest of the data unit. Subsequent calls to the **t_rcvudata** subroutine return zero for the length and options until the full data unit is received.

Parameters

fd Identifies the local transport endpoint through which data will be received.

unitdata

Holds information associated with the received data unit. The *unitdata* parameter points to a **t_unitdata** structure containing the following members:

```
struct netbuf addr;
struct netbuf opt;
struct netbuf udata;
```

On return from this call:

addr Specifies the protocol address of the sending user.

opt Identifies options that were associated with this data unit.

udata Specifies the user data that was received.

The *maxlen* field of *addr, opt,* and *udata* must be set before calling this subroutine to indicate the maximum size of the buffer for each.

flags Indicates that the complete data unit was not received.

Valid States

T_IDLE

Return Values

Upon successful completion, a value of 0 is returned. Otherwise, a value of –1 is returned and **t_errno** is set to indicate an error.

Error Codes

On failure, **t_errno** is set to one of the following:

TBADF	The specified file descriptor does not refer to a transport endpoint.
TBODATA	**O_NONBLOCK** was set, but no data units are currently available from the transport provider.
TBUFOVFLW	The number of bytes allocated for the incoming protocol address or options (*maxlen*) is greater than 0 but not sufficient to store the information. The unit data information to be returned in the *unitdata* parameter is discarded.
TLOOK	An asynchronous event has occurred on the transport endpoint and requires immediate attention.
TNOTSUPPORT	This subroutine is not supported by the underlying transport provider.
TOUTSTATE	The subroutine was issued in the wrong sequence on the transport endpoint referenced by the *fd* parameter.
TPROTO	This error indicates that a communication problem has been detected between the X/Open Transport Interface and the transport provider for which there is no other suitable X/Open Transport Interface (**t_errno**).
TSYSERR	A system error has occurred during execution of this subroutine.

Related Information

The **fcntl** subroutine, **t_alloc** subroutine, **t_open** subroutine, **t_rcvuderr** subroutine, **t_sndudata** subroutine.

t_rcvuderr Subroutine for X/Open Transport Interface

Purpose

Receive a unit data error indication.

Library

X/Open Transport Interface Library (**libxti.a**)

Syntax

```
#include <xti.h>

int t_rcvuderr (fd, uderr)
int fd;
struct t_uderr *uderr;
```

Description

The **t_rcvuderr** subroutine is used in connectionless mode to receive information concerning an error on a previously sent data unit, and should only be issued following a unit data error indication. It informs the transport user that a data unit with a specific destination address and protocol options produced an error.

Parameters

fd Identifies the local transport endpoint through which the error report will be received.

uderr

Points to a **t_uderr** structure containing the following members:

```
struct netbuf addr;
struct netbuf opt;
long error;
```

The *maxlen* field of *add* and *opt* must be set before calling this subroutine to indicate the maximum size of the buffer for each.

On return from this call:

addr Specifies the destination protocol address of the erroneous data unit.

opt Identifies options that were associated with the data unit.

error Specifies a protocol–dependent error code.

If the user does not care to identify the data unit that produced an error, *uderr* may be set to a null pointer, and the **t_rcvuderr** subroutine simply clears the error indication without reporting any information to the user.

Valid States

T_IDLE

Return Values

0 Successful completion.

–1 **t_errno** is set to indicate an error.

Error Codes

On failure, **t_errno** is set to one of the following:

TBADF	The specified file descriptor does not refer to a transport endpoint.
TBUFOVFLW	The number of bytes allocated for the incoming protocol address or options (*maxlen*) is greater than 0 but not sufficient to store the information. The unit data information to be returned in the *uderr* parameter is discarded.
TNOTSUPPORT	This subroutine is not supported by the underlying transport provider.
TNOUDERR	No unit data error indication currently exists on the specified transport endpoint.
TPROTO	This error indicates that a communication problem has been detected between the X/Open Transport Interface and the transport provider for which there is no other suitable X/Open Transport Interface (**t_errno**).
TSYSERR	A system error has occurred during execution of this subroutine.

Related Information

The **t_rcvudata** subroutine, **t_sndudata** subroutine.

t_snd Subroutine for X/Open Transport Interface

Purpose

Send data or expedited data over a connection.

Library

X/Open Transport Interface Library (**libxti.a**)

Syntax

```
#include <xti.h>

int t_snd (
    int fd,
    void *buf,
    unsigned int nbytes,
    int *flags)
```

Description

The **t_snd** subroutine is used to send either normal or expedited data. By default, the **t_snd** subroutine operates in synchronous mode and may wait if flow control restrictions prevents the data from being accepted by the local transport provider at the time the call is made. However, if **O_NONBLOCK** is set (via the **t_open** subroutine or *fcntl*), the **t_snd** subroutine executes in asynchronous mode, and fails immediately if there are flow control restrictions. The process can arrange to be informed when the flow control restrictions are cleared via either the **t_look** subroutine or the Event Management interface.

On successful completion, the **t_snd** subroutine returns the number of bytes accepted by the transport provider. Normally this equals the number of bytes specified in the *nbytes* parameter. However, if **O_NONBLOCK** is set, it is possible that only part of the data is actually accepted by the transport provider. In this case, the **t_snd** subroutine returns a value that is less than the value of the *nbytes* parameter. If the value of the *nbytes* parameter is zero and sending of zero octets is not supported by the underlying transport service, the **t_snd** subroutine returns –1 with **t_errno** set to **TBADDATA**.

Parameters

fd	Identifies the local transport endpoint over which data should be sent.
buf	Points to the user data.

nbytes Specifies the number of bytes of user data to be sent.

flags

Specifies any optional flags described below:

T_EXPEDITED If set in the *flags* parameter, the data is sent as expedited data and is subject to the interpretations of the transport provider.

T_MORE If set in the *flags* parameter, indicates to the transport provider that the transport service data unit (TSDU) (or expedited transport service data unit – ETSDU) is being sent through multiple **t_snd** calls. Each **t_snd** call with the **T_MORE** flag set indicates that another **t_snd** call will follow with more data for the current TSDU (or ETSDU).

The end of the TSDU (or ETSDU) is identified by a **t_snd** call with the **T_MORE** flag not set. Use of **T_MORE** enables a user to break up large logical data units without losing the boundaries of those units at the other end of the connection. The flag implies nothing about how the data is packaged for transfer below the transport interface. If the transport provider does not support the concept of a TSDU, as indicated in the *info* parameter on return from the **t_open** or **t_getinfo** subroutines, the **T_MORE** flag is not meaningful and is ignored if set.

The sending of a zero–length fragment of a TSDU or ETSDU is only permitted where this is used to indicate the end of a TSDU or ETSDU, for example, when the **T_MORE** flag is not set. Some transport providers also forbid zero–length TSDUs and ETSDUs. See Appendix A, ISO Transport Protocol Information for a fuller explanation.

Valid States

T_DATAXFER, **T_INREL**.

Return Values

On successful completion, the **t_snd** subroutine returns the number of bytes accepted by the transport provider. Otherwise, –1 is returned on failure and **t_errno** is set to indicate the error.

Note, that in asynchronous mode, if the number of bytes accepted by the transport provider is less than the number of bytes requested, this may indicate that the transport provider is blocked due to flow control.

Error Codes

On failure, **t_errno** is set to one of the following:

TBADF The specified file descriptor does not refer to a transport endpoint.

TBADDATA Illegal amount of data:

- A single send was attempted specifying a TSDU (ETSDU) or fragment TSDU (ETSDU) greater than that specified by the current values of the TSDU or ETSDU fields in the *info* argument;
- a send of a zero byte TSDU (ETSDU) or zero byte fragment of a TSDU (ETSDU) is not supported by the provider (see Appendix A, ISO Transport Protocol Information) .
- multiple sends were attempted resulting in a TSDU (ETSDU) larger than that specified by the current value of the TSDU or ETSDU fields in the *info* argument – the ability of an XTI implementation to detect such an error case is implementation–dependent. See ”Implementation Specifics”.

TBADFLAG An invalid flag was specified.

TFLOW **O_NONBLOCK** was set, but the flow control mechanism prevented the transport provider from accepting any data at this time.

TNOTSUPPORT This subroutine is not supported by the underlying transport provider.

TLOOK An asynchronous event has occurred on this transport endpoint.

TOUTSTATE The subroutine was issued in the wrong sequence on the transport endpoint referenced by the *fd* parameter.

TSYSERR A system error has occurred during execution of this subroutine.

TPROTO This error indicates that a communication problem has been detected between the X/Open Transport Interface and the transport provider for which there is no other suitable X/Open Transport Interface (**t_errno**).

Implementation Specifics

It is important to remember that the transport provider treats all users of a transport endpoint as a single user. Therefore if several processes issue concurrent **t_snd** calls then the different data may be intermixed.

Multiple sends which exceed the maximum TSDU or ETSDU size may not be discovered by the X/Open Transport Interface. In this case an implementation–dependent error will result (generated by the transport provider) perhaps on a subsequent XTI call. This error may take the form of a connection abort, a **TSYSERR**. a **TBADDATA** or a **TPROTO** error.

If multiple sends which exceed the maximum TSDU or ETSDU size are detected by the X/Open Transport Interface, **t_snd** fails with **TBADDATA**.

Related Information

The **t_getinfo** subroutine, **t_open** subroutine, **t_rcv** subroutine.

t_snddis Subroutine for X/Open Transport Interface

Purpose

Send user–initiated disconnect request.

Library

X/Open Transport Interface Library (**libxti.a**)

Syntax

```
#include <xti.h>

int t_snddis (
    int fd,
    const struct t_call *call)
```

Description

The **t_snddis** subroutine is used to initiate an abortive release on an already established connection, or to reject a connect request.

Parameters

fd Identifies the local transport endpoint of the connection.

call Specifies information associated with the abortive release. The *call* parameter points to a **t_call** structure which contains the following members:

```
struct netbuf addr;
struct netbuf opt;
struct netbuf udata;
int sequence;
```

The values in the *call* parameter have different semantics, depending on the context of the call to the **t_snddis** subroutine. When rejecting a connect request, the *call* parameter must be non–null and contain a valid value of *sequence* to uniquely identify the rejected connect indication to the transport provider. The *sequence* field is only meaningful if the transport connection is in the **T_INCON** state. The *addr* and *opt* fields of the *call* parameter are ignored. In all other cases, the *call* parameter need only be used when data is being sent with the disconnect request. The *addr, opt* and *sequence* fields of the **t_call** structure are ignored. If the user does not wish to send data to the remote user, the value of the *call* parameter may be a null pointer.

The *udata* field specifies the user data to be sent to the remote user. The amount of user data must not exceed the limits supported by the transport provider, as returned in the the **t_open** or **t_getinfo** subroutines *info* parameter *discon* field. If the *len* field of *udata* is zero, no data will be sent to the remote user.

Valid States

T_DATAXFER, **T_OUTCON**, **T_OUTREL**, **T_INREL**, **T_INCON**(*ocnt* > 0).

Return Values

0 Successful completion.

–1 **t_errno** is set to indicate an error.

Error Codes

On failure, **t_errno** is set to one of the following:

TBADDATA	The amount of user data specified was not within the bounds allowed by the transport provider.
TBADF	The specified file descriptor does not refer to a transport endpoint.
TBADSEQ	An invalid sequence number was specified, or a null *call* pointer was specified, when rejecting a connect request.
TLOOK	An asynchronous event, which requires attention has occurred.
TNOTSUPPORT	This subroutine is not supported by the underlying transport provider.
TOUTSTATE	The subroutine was issued in the wrong sequence on the transport endpoint referenced by the *fd* parameter.
TPROTO	This error indicates that a communication problem has been detected between the X/Open Transport Interface and the transport provider for which there is no other suitable X/Open Transport Interface (**t_errno**).
TSYSERR	A system error has occurred during execution of this subroutine.

Implementation Specifics

The **t_snddis** subroutine is an abortive disconnect. Therefore a **t_snddis** call issued on a connection endpoint may cause data previously sent via the **t_snd** subroutine, or data not yet received, to be lost (even if an error is returned).

Related Information

The **t_connect** subroutine, **t_getinfo** subroutine, **t_listen** subroutine, **t_open** subroutine.

t_sndrel Subroutine for X/Open Transport Interface

Purpose

Initiate an orderly release.

Library

X/Open Transport Interface Library (**libxti.a**)

Syntax

```
#include <xti.h>

int t_sndrel (fd)
int fd;
```

Description

The **t_sndrel** subroutine is used to initiate an orderly release of a transport connection and indicates to the transport provider that the transport user has no more data to send.

After calling the **t_sndrel** subroutine, the user may not send any more data over the connection. However, a user may continue to receive data if an orderly release indication has not been received. This subroutine is an optional service of the transport provider and is only supported if the transport provider returned service type **T_COTS_ORD** on the **t_open** or **t_getinfo** subroutines.

Parameter

fd	Identifies the local transport endpoint where the connection exists.

Valid States

T_DATAXFER, **T_INREL**.

Return Values

0	Successful completion.
–1	**t_errno** is set to indicate an error.

Error Codes

On failure, **t_errno** is set to one of the following:

TBADF	The specified file descriptor does not refer to a transport endpoint.
TFLOW	**O_NONBLOCK** was set, but the flow control mechanism prevented the transport provider from accepting the subroutine at this time.
TLOOK	An asynchronous event has occurred on this transport endpoint and requires immediate attention.
TNOTSUPPORT	This subroutine is not supported by the underlying transport provider.
TOUTSTATE	The subroutine was issued in the wrong sequence on the transport endpoint referenced by the *fd* parameter.
TPROTO	This error indicates that a communication problem has been detected between X/Open Transport Interface and the transport provider for which there is no other suitable X/Open Transport Interface (**t_errno**).
TSYSERR	A system error has occurred during execution of this subroutine.

Related Information

The **t_getinfo** subroutine, **t_open** subroutine, **t_rcvrel** subroutine.

t_sndudata Subroutine for X/Open Transport Interface

Purpose

Send a data unit.

Library

X/Open Transport Interface Library (**libxti.a**)

Syntax

```
#include <xti.h>

int t_sndudata (
    int fd,
    const struct t_unitdata *unitdata)
```

Description

The **t_sndudata** subroutine is used in connectionless mode to send a data unit from another transport user.

By default, the **t_sndudata** subroutine operates in synchronous mode and waits if flow control restrictions prevents the data from being accepted by the local transport provider at the time the call is made. However, if **O_NONBLOCK** is set (via the **t_open** subroutine or *fcntl*), the **t_sndudata** subroutine executes in asynchronous mode and fails under such conditions. The process can arrange to be notified of the clearance of a flow control restriction via either the **t_look** subroutine or the Event Management interface.

If the amount of data specified in the *udata* field exceeds the TSDU size as returned in the **t_open** or **t_getinfo** subroutines *info* parameter *tsdu* field, a **TBADDATA** error will be generated. If the **t_sndudata** subroutine is called before the destination user has activated its transport endpoint (see the **t_bind** subroutine), the data unit may be discarded.

If it is not possible for the transport provider to immediately detect the conditions that cause the errors **TBADDADDR** and **TBADOPT**. These errors will alternatively be returned by the **t_rcvuderr** subroutine. Therefore, an application must be prepared to receive these errors in both of these ways.

Parameters

fd Identifies the local transport endpoint through which data will be sent.

unitdata

Points to a **t_unitdata** structure containing the following members:

```
struct netbuf addr;
struct netbuf opt;
struct netbuf udata;
```

In the *unitdata* structure:

addr Specifies the protocol address of the destination user.

opt Identifies options that the user wants associated with this request.The user may choose not to specify what protocol options are associated with the transfer by setting the *len* field of *opt* to zero. In this case, the provider may use default options.

udata Specifies the user data to be sent. If the *len* field of *udata* is zero, and sending of zero octets is not supported by the underlying transport service, the **t_sndudata** subroutine returns –1 with **t_errno** set to **TBADDATA**.

Valid States

T_IDLE

Return Values

0 Successful completion.

–1 **t_errno** is set to indicate an error.

Error Codes

On failure, **t_errno** is set to one of the following:

TBADADDR The specified protocol address was in an incorrect format or contained illegal information.

TBADDATA Illegal amount of data. A single send was attempted specifying a TSDU greater than that specified in the *info* parameter, or a send of a zero byte TSDU is not supported by the provider.

TBADF The specified file descriptor does not refer to a transport endpoint.

TBADOPT The specified options were in an incorrect format or contained illegal information.

TFLOW **O_NONBLOCK** was set, but the flow control mechanism prevented the transport provider from accepting any data at this time.

TLOOK An asynchronous event has occurred on the transport endpoint.

TNOTSUPPORT This subroutine is not supported by the underlying transport provider.

TOUTSTATE The subroutine was issued in the wrong sequence on the transport endpoint referenced by the *fd* parameter.

TPROTO This error indicates that a communication problem has been detected between the X/Open Transport Interface and the transport provider for which there is no other suitable X/Open Transport Interface (**t_errno**).

TSYSERR A system error has occurred during execution of this subroutine.

Related Information

The **fcntl** subroutine, **t_alloc** subroutine, **t_open** subroutine, **t_rcvudata** subroutine, **t_rcvuderr** subroutine.

t_strerror Subroutine for X/Open Transport Interface

Purpose

Produce an error message string.

Library

X/Open Transport Interface Library (**libxti.a**)

Syntax

```
#include <xti.h>

const char *t_strerror (
int errnum)
```

Description

The **t_strerror** subroutine maps the error number to a language–dependent error message string and returns a pointer to the string. The error number specified by the *errnum* parameter corresponds to an X/Open Transport Interface error. The string pointed to is not modified by the program, but may be overwritten by a subsequent call to the **t_strerror** subroutine. The string is not terminated by a newline character. The language for error message strings written by the **t_strerror** subroutine is implementation–defined. If it is English, the error message string describing the value in **t_errno** is identical to the comments following the **t_errno** codes defined in the **xti.h** header file. If an error code is unknown, and the language is English, **t_strerror** returns the string.

```
"<error>: error unknown"
```

where <error> is the error number supplied as input. In other languages, an equivalent text is provided.

Parameter

errnum	Specifies the error number.

Valid States

ALL – except **T_UNINIT**.

Return Values

The **t_strerror** subroutine returns a pointer to the generated message string.

Related Information

The **t_error** subroutine.

t_sync Subroutine for X/Open Transport Interface

Purpose

Synchronize transport library.

Library

X/Open Transport Interface Library (**libxti.a**)

Syntax

```
#include <xti.h>

int t_sync (fd)
int fd;
```

Description

The **t_sync** subroutine synchronizes the data structures managed by the transport library with information from the underlying transport provider. In doing so, if the file descriptor referenced a transport endpoint, the subroutine can convert an uninitialized file descriptor (obtained using the **open** or **dup** subroutines or as a result of a **fork** operation and an **exec** operation) to an initialized transport endpoint, by updating and allocating the necessary library data structures. This subroutine also allows two cooperating processes to synchronize their interaction with a transport provider.

For example, if a process forks a new process and issues an **exec** operation, the new process must issue a **t_sync** to build the private library data structure associated with a transport endpoint and to synchronize the data structure with the relevant provider information.

It is important to remember that the transport provider treats all users of a transport endpoint as a single user. If multiple processes are using the same endpoint, they should coordinate their activities so as not to violate the state of the transport endpoint. The **t_sync** subroutine returns the current state of the transport endpoint to the user, thereby enabling the user to verify the state before taking further action. This coordination is only valid among cooperating processes; it is possible that a process or an incoming event could change the endpoint's state *after* a **t_sync** call is issued.

If the transport endpoint is undergoing a state transition when the **t_sync** subroutine is called, the subroutine will fail.

Parameter

fd	Specifies the transport endpoint.

Valid States

ALL – except **T_UNINIT**.

Return Values

On successful completion, the state of the transport endpoint is returned. Otherwise, a value of –1 is returned and **t_errno** is set to indicate an error. The state returned is one of the following:

T_UNBND	Unbound.
T_IDLE	Idle.
T_OUTCON	Outgoing connection pending.
T_INCON	Incoming connection pending.

T_DATAXFER	Data transfer.
T_OUTREL	Outgoing orderly release (waiting for an orderly release indication).
T_INREL	Incoming orderly release (waiting for an orderly release request).

Error Codes

On failure, **t_errno** is set to one of the following:

TBADF	The specified file descriptor does not refer to a transport endpoint. This error may be returned when the *fd* parameter has been previously closed or an erroneous number may have been passed to the call.
TPROTO	This error indicates that a communication problem has been detected between the X/Open Transport Interface and the transport provider for which there is no other suitable X/Open Transport Interface (**t_errno**).
TSTATECHNG	The transport endpoint is undergoing a state change.
TSYSERR	A system error has occurred during execution of this function.

Related Information

The **dup** subroutine, **exec** subroutine, **fork** subroutine, **open** subroutine.

t_unbind Subroutine for X/Open Transport Interface

Purpose

Disable a transport endpoint.

Library

X/Open Transport Interface Library (**libxti.a**)

Syntax

```
#include <xti.h>

int t_unbind (fd)
int fd;
```

Description

The **t_unbind** subroutine disables the transport endpoint which was previously bound by **t_bind**. On completion of this call, no further data or events destined for this transport endpoint are accepted by the transport provider. An endpoint which is disabled by using the **t_unbind** subroutine can be enabled by a subsequent call to the **t_unbind** subroutine.

Parameter

fd	Specifies the transport endpoint.

Valid States

T_IDLE

Return Values

0	Successful completion.
–1	**t_errno** is set to indicate an error.

Errors

On failure, **t_errno** is set to one of the following:

TBADF	The specified file descriptor does not refer to a transport endpoint.
TOUTSTATE	The subroutine was issued in the wrong sequence.
TLOOK	An asynchronous event has occurred on this transport endpoint.
TSYSERR	A system error has occurred during execution of this subroutine.
TPROTO	This error indicates that a communication problem has been detected between the X/Open Transport Interface and the transport provider for which there is no other suitable X/Open Transport Interface (**t_errno**).

Related Information

The **t_bind** subroutine.

Options for the X/Open Transport Interface

Options are formatted according to the **t_opthdr** structure as described in **"Use of Options for the** X/Open Transport Interface". A transport provider compliant to this specification supports none, all, or any subset of the options defined in the following sections: "TCP/IP–Level Options" to "IP–level Options". An implementation may restrict the use of any of these options by offering them only in the privileged or read–only mode.

TCP–Level Options

The protocol level is **INET_TCP**. For this level, the following table shows the options that are defined.

TCP–Level Options			
Option Name	**Type of Option Value**	**Legal Option Value**	**Meaning**
TCP_KEEPALIVE	struct t_kpalive	see text following table	check if connections are live
TCP_MAXSEG	unsigned long	length in octets	get TCP maximum segment size
TCP_NODELAY	unsigned long	T_YES T_NO	don't delay send to coalesce packets

TCP_KEEPALIVE

If set, a keep–alive timer is activated to monitor idle connections that may no longer exist. If a connection has been idle since the last keep–alive timeout, a keep–alive packet is sent to check if the connection is still alive or broken.

Keep–alive packets are not an explicit feature of TCP, and this practice is not universally accepted. According to **RFC 1122**:

”a keep–alive mechanism should only be invoked in server applications that might otherwise hang indefinitely and consume resources unnecessarily if a client crashes or aborts a connection during a network failure.”

The option value consists of a structure **t_kpalive** declared as:

```
struct t_kpalive {
   long kp_onoff;
   long kp_timeout;
}
```

The **t_kpalive** fields and the possible values are:

kp_onoff Switches option on or off. Legal values for the field are:

T_NO Switch keep–alive timer off.

T_YES Activate keep–alive timer.

T_YES | T_GARBAGE
Activate keep–alive timer and send garbage octet.

Usually, an implementation should send a keep–alive packet with no data (**T_GARBAGE** not set). If **T_GARBAGE** is set, the keep–alive packet contains one garbage octet for compatibility with erroneous TCP implementations.

An implementation is, however, not obliged to support **T_GARBAGE** (see RFC 1122). Since the *kp_onoff* value is an absolute requirement, the request ”**T_YES | T_GARBAGE**” may therefore be rejected.

kp_timeout Specifies the keep–alive timeout in minutes. This field determines the frequency of keep–alive packets being sent, in minutes. The transport user can request the default value by setting the field to **T_UNSPEC**. The default is implementation–dependent, but at least 120 minutes (see RFC 1122). Legal values for this field are **T_UNSPEC** and all positive numbers.

The timeout value is not an absolute requirement. The implementation may pose upper and lower limits to this value. Requests that fall short of the lower limit may be negotiated to the lower limit.

The use of this option might be restricted to privileged users.

TCP_MAXSEG Used to retrieve the maximum TCP segment size. This option is read–only.

TCP_NODELAY

Under most circumstances, TCP sends data as soon as it is presented. When outstanding data has not yet been acknowledged, it gathers small amounts of output to be sent in a single packet once an acknowledgment is received. For a small number of clients, such as window systems (for example, Enhanced AIXwindows) that send a stream of mouse events which receive no replies, this packetization may cause significant delays. **TCP_NODELAY** is used to defeat this algorithm. Legal option values are:

T_YES Do not delay.

T_NO Delay.

These options are not association–related. The options may be negotiated in all X/Open Transport Interface states except **T_UNBIND** and **T_UNINIT**. The options are read–only in the **T_UNBIND** state. See "**The Use of Options for the** X/Open Transport Interface" for the differences between association–related options and those options that are not.

Absolute Requirements

A request for **TCP_NODELAY** and a request to activate **TCP_KEEPALIVE** is an absolute requirement. **TCP_MAXSEG** is a read–only option.

UDP–level Options

The protocol level is **INET_UDP**. The option defined for this level is shown in the following table.

UDP–Level Options			
Option Name	**Type of Option Value**	**Legal Option Value**	**Meaning**
UDP_CHECKSUM	unsigned long	T_YES/T_NO	checksum computation

UDP_CHECKSUM

Allows disabling and enabling of the UDP checksum computation. The legal values are:

T_YES Checksum enabled.

T_NO Checksum disabled.

This option is association–related. It may be negotiated in all XTI states except **T_UNBIND** and **T_UNINI**T. It is read–only in state **T_UNBND**.

If this option is returned with the **t_rcvudata** subroutine, its value indicates whether a checksum was present in the received datagram or not.

Numerous cases of undetected errors have been reported when applications chose to turn off checksums for efficiency. The advisability of ever turning off the checksum check is very controversial.

Absolute Requirements

A request for this option is an absolute requirement.

IP–level Options

The protocol level is **INET_IP**. The options defined for this level are listed in the following table.

IP–Level Options			
Option Name	**Type of Option Value**	**Legal Option Value**	**Meaning**
IP_BROADCAST	unsigned int	T_YES/T_NO	permit sending of broadcast messages
IP_DONTROUTE	unsigned int	T_YES/T_NO	just use interface addresses
IP_OPTIONS	array of unsigned characters	see text	IP per–packet options
IP_REUSEADDR	unsigned int	T_YES/T_NO	allow local address reuse

IP_TOS	unsigned char	see text	IP per–packet type of service
IP_TTL	unsigned char	time in seconds	IP per packet time–to–live

IF_BROADCAST Requests permission to send broadcast datagrams. It was defined to make sure that broadcasts are not generated by mistake. The use of this option is often restricted to privileged users.

IP_DONTROUTE Indicates that outgoing messages should bypass the standard routing facilities. It is mainly used for testing and development.

IP_OPTIONS Sets or retrieves the OPTIONS field of each outgoing (incoming) IP datagram. Its value is a string of octets composed of a number of IP options, whose format matches those defined in the IP specification with one exception: the list of addresses for the source routing options must include the first–hop gateway at the beginning of the list of gateways. The first–hop gateway address will be extracted from the option list and the size adjusted accordingly before use.

The option is disabled if it is specified with ”no value,” for example, with an option header only.

The **t_connect** (in synchronous mode), **t_listen**, **t_rcvconnect** and **t_rcvudata** subroutines return the OPTIONS field, if any, of the received IP datagram associated with this call. The **t_rcvuderr** subroutine returns the OPTIONS field of the data unit previously sent that produced the error. The **t_optmgmt** subroutine with **T_CURRENT** set retrieves the currently effective **IP_OPTIONS** that is sent with outgoing datagrams.

Common applications never need this option. It is mainly used for network debugging and control purposes.

IP_REUSEADDR Many TCP implementations do not allow the user to bind more than one transport endpoint to addresses with identical port numbers. If **IP_REUSEADDR** is set to **T_YES** this restriction is relaxed in the sense that it is now allowed to bind a transport endpoint to an address with a port number and an underspecified internet address (”wild card” address) and further endpoints to addresses with the same port number and (mutually exclusive) fully specified internet addresses.

IP_TOS Sets or retrieves the *type–of–service* field of an outgoing (incoming) IP datagram. This field can be constructed by any OR'ed combination of one of the precedence flags and the type–of–service flags **T_LDELAY**, **T_HITHRPT**, and **T_HIREL**:

- Precedence:

  These flags specify datagram precedence, allowing senders to indicate the importance of each datagram. They are intended for Department of Defense applications. Legal flags are:

  ```
  T_ROUTINE
  T_PRIORITY
  T_IMMEDIATE
  T_FLASH
  T_OVERRIDEFLASH
  T_CRITIC_ECP
  T_INETCONTROL
  T_NETCONTROL
  ```

  Applications using **IP_TOS** but not the precedence level should use the value **T_ROUTINE** for precedence.

- Type of service:

  These flags specify the type of service the IP datagram desires. Legal flags are:

  T_NOTOS requests no distinguished type of service

  T_LDELAY requests low delay

  T_HITHRPT requests high throughput

  T_HIREL requests high reliability

  The option value is set using the macro **SET_TOS(***prec, tos***)** where *prec* is set to one of the precedence flags and *tos* to one or an OR'ed combination of the type–of–service flags. **SET_TOS** returns the option value.

  The **t_connect**, **t_listen**, **t_rcvconnect** and **t_rcvudata** subroutines return the *type–of–service* field of the received IP datagram associated with this call. The **t_rcvuderr** subroutine returns the *type–of–service* field of the data unit previously sent that produced the error.

  The **t_optmgmt** subroutine with **T_CURRENT** set retrieves the currently effective **IP_TOS** value that is sent with outgoing datagrams.

  The requested *type–of–service* cannot be guaranteed. It is a hint to the routing algorithm that helps it choose among various paths to a destination. Note also, that most hosts and gateways in the Internet these days ignore the *type–of–service* field.

IP_TIL This option is used to set the *time–to–live* field in an outgoing IP datagram. It specifies how long, in seconds, the datagram is allowed to remain in the Internet. The *time–to–live* field of an incoming datagram is not returned by any function (since it is not an association–related option).

IP_OPTIONS and **IP_TOS** are both association–related options. All other options are not association–related.

IP_REUSEADDR may be negotiated in all XTI states except **T_UNINIT**. All other options may be negotiated in all other XTI states except **T_UNBND** and **T_UNINIT**; they are read–only in the state **T_UNBND**.

Absolute Requirements

A request for any of these options in an absolute requirement.

Index

Symbols

Numbers

A

B

C

D

E

F

G

H

I

L

M

N

O

P

Q

R

S

T

U

V

W

X

Vos remarques sur ce document / Technical publication remark form

Titre / Title : Bull Technical Reference Communications Volume 2/2

Nº Reférence / Reference Nº : 86 A2 84AP 04

Daté / Dated : February 1999

ERREURS DETECTEES / ERRORS IN PUBLICATION

AMELIORATIONS SUGGEREES / SUGGESTIONS FOR IMPROVEMENT TO PUBLICATION

Vos remarques et suggestions seront examinées attentivement.
Si vous désirez une réponse écrite, veuillez indiquer ci-après votre adresse postale complète.

Your comments will be promptly investigated by qualified technical personnel and action will be taken as required.
If you require a written reply, please furnish your complete mailing address below.

NOM / NAME : _____ Date : _____

SOCIETE / COMPANY : _____

ADRESSE / ADDRESS : _____

_____

Remettez cet imprimé à un responsable BULL ou envoyez-le directement à :

Please give this technical publication remark form to your BULL representative or mail to:

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

Technical Publications Ordering Form

Bon de Commande de Documents Techniques

To order additional publications, please fill up a copy of this form and send it via mail to:
Pour commander des documents techniques, remplissez une copie de ce formulaire et envoyez-la à :

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

Managers / Gestionnaires :
Mrs. / Mme : **C. DUMOULIN** +33 (0) 2 41 73 76 65
Mr. / M : **L. CHERUBIN** +33 (0) 2 41 73 63 96

FAX : +33 (0) 2 41 73 60 19
E–Mail / Courrier Electronique : srv.Cedoc@franp.bull.fr

Or visit our web site at: / Ou visitez notre site web à:
`http://www-frec.bull.com` (PUBLICATIONS, Technical Literature, Ordering Form)

CEDOC Reference # Nº Référence CEDOC	**Qty** Qté	**CEDOC Reference #** Nº Référence CEDOC	**Qty** Qté	**CEDOC Reference #** Nº Référence CEDOC	**Qty** Qté
__ __ ____ _ [__]		__ __ ____ _ [__]		__ __ ____ _ [__]	
__ __ ____ _ [__]		__ __ ____ _ [__]		__ __ ____ _ [__]	
__ __ ____ _ [__]		__ __ ____ _ [__]		__ __ ____ _ [__]	
__ __ ____ _ [__]		__ __ ____ _ [__]		__ __ ____ _ [__]	
__ __ ____ _ [__]		__ __ ____ _ [__]		__ __ ____ _ [__]	
__ __ ____ _ [__]		__ __ ____ _ [__]		__ __ ____ _ [__]	
__ __ ____ _ [__]		__ __ ____ _ [__]		__ __ ____ _ [__]	

[_ _] : **no revision number means latest revision** / pas de numéro de révision signifie révision la plus récente

NOM / NAME : ______________________________ Date : __________

SOCIETE / COMPANY : ______________________________

ADRESSE / ADDRESS : ______________________________

____

PHONE / TELEPHONE : ____________________ FAX : ____________________

E–MAIL : ______________________________

For Bull Subsidiaries / Pour les Filiales Bull :

Identification: ______________________________

For Bull Affiliated Customers / Pour les Clients Affiliés Bull :

Customer Code / Code Client : ______________________________

For Bull Internal Customers / Pour les Clients Internes Bull :

Budgetary Section / Section Budgétaire : ______________________________

For Others / Pour les Autres :

Please ask your Bull representative. / Merci de demander à votre contact Bull.

PLACE BAR CODE IN LOWER LEFT CORNER

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

ORDER REFERENCE
86 A2 84AP 04

Bull

Utiliser les marques de découpe pour obtenir les étiquettes.
Use the cut marks to get the labels.

AIX
Technical
Reference
Communications
Volume 2/2
86 A2 84AP 04

AIX
Technical
Reference
Communications
Volume 2/2
86 A2 84AP 04

AIX
Technical
Reference
Communications
Volume 2/2
86 A2 84AP 04